Developing Non-Playable Character AI in *StarLight*

Team 2

Abstract

The use of artificial intelligence (AI) in video games has become a widespread practice. In particular, the use of AI for non-playable characters (NPCs) has become a popular technique for adding dynamism and depth to game worlds. We have developed an NPC artificial intelligence for the space shooter game, *StarLight*. We claim that a believable NPC can be created in the context of *StarLight*, using a combination of only basic movement AI algorithms.

Keywords: state machines, motor control, movement

Introduction

Artificial intelligence (AI) appeared on the video game scene more than 20 years ago. In the earliest generations of shooters, players were pleasantly surprised if AI characters interacted with them in even basic ways. This set of player expectations and observations is defined as the "perception window". During the early days of gaming, the AI perception window was relatively small and certainly very basic. As game hardware evolved, believable - even photorealistic - environments became possible. Advanced physics, lighting, and animation constructed believable worlds in front of the players eyes at 60 frames a second.

To complement this newfound realism, more dynamic models of AI were required. In 2001, Monolith Productions popular first person shooter (FPS) franchise *No One Lives Forever (NOLF)* demonstrated an important step forward. NOLF endowed its NPCs with the ability to take cover, flip over furniture to create cover, all while tactically utilizing cover positions to their advantage. Six years later, Monolith built on this success with a new title, *First Encounter Assault Recon (F.E.A.R.)*. F.E.A.R. employed cooperative squad movement, suppression fire, and fire and flank teams [7].

Despite advances shown in many AAA titles, progress in game AI has been relatively slow. Developing advanced AI techniques for real time games requires a significant time investment; a cost which most game developers cannot afford [11]. Additionally, as games continue to become more complex, NPCs must take into account an exceedingly large amount of data. Real time environments impose often crippling limits on processing time and storage; making this task potentially prohibitively difficult.

What if we could address these challenges with *only* the most basic AI building block: movement? Not all games are high end first person shooters, especially as low power mobile devices have become increasing popular. Since building a *believable* AI is just as important as building a *smart* AI, can we accomplish the bulk of this task simply?

We have completed a semester project developing an NPC artificial intelligence for the asteroids-inspired game, *StarLight*. We claim that a believable NPC can be created in the context of *StarLight*, using a combination of *only* basic movement AI algorithms. While this approach might not work for the most complex game types, it is applicable to an emerging, and highly popular, mobile gaming environment.

The remainder of this paper is organized as follows: Section 2 details related research work to the topic. Section 3 introduces our game world and associated technical details. Section 4 discusses the specific algorithms in our implementation. Section 5 evaluates the results, followed by limitations, future work, and final thoughts.

Related Work

Artificial intelligence for games produces character actions that mimic human behavior, without the need for a game player's control [1]. One such example is the popular franchise *The Sims* by Maxis Games. *The Sims* applies artificial life (A-life) and fuzzy logic techniques for NPC characters to react in the game world. Similarly, in first person shooter (FPS) games, such as *Unreal Tournament*, even basic flocking movement techniques are employed for coordinated group movement. In Real Time Strategy (RTS) games, including *Warcraft 3*, the number of NPCs can easily exceed hundreds of units. Hierarchical AI is used in this context to reduce unnecessary calculation time [1].

We discovered the two most commonly used game AI techniques are A* and Finite State Machines (FSMs). Nearly all games exhibit some form of AI that implements A* or FSM to plan paths or make decisions [7]. Many of the related work we came across used methods such as Finding, Finite State Machine, Fuzzy State Machine, Script, Flocking, Decision Trees, A-life, Neural Networks and Genetic Algorithms [1].

Adaptation

Adapting to the state of the environment is another important, yet often difficult, feature for a NPC to possess. If a NPC lacks the ability to adapt then in some cases it will appear to behave foolishly; thereby breaking the illusion of an intelligent agent. Research by Hartley et. al provides a solution to this problem by using online tactical learning [5]. Their research demonstrated that tactical online learning technique can be applied to develop successful NPC in games.

Additionally, games use Monte-Carlo methods for artificial game opponents. This produces a desirable level of adaptability according to changing conditions of the environment. One Monte Carlo based implementation by Xiao et. al. was applied to Pac-Man [6]. In order to reduce the computation intensiveness of Monte-Carlo, an Artificial Neural Network (ANN) is used. Consequently, this combination of multiple AI techniques can yield a more effective AI.

Learning

Layered Learning, first proposed by S. Mondesire et. al, can be used to make decisions in autonomous systems like robotic and other computer controlled agents. Layered Learning decomposes a complex task into simpler tasks and then trains agents to perform each of these sub-task. The motivation behind Layered Learning is to provide an effective model for decision making for computer-based agents and is highly applicable within the context of real time computer games [4].

To create a believable character, learning often needs to be dynamic. AI characters in *F.E.A.R.* path plan in real-time, creating a simulated effect of reaction to various environmental factors [7]. There has also been work done in online learning technique for game AI called 'dynamic scripting', that uses an adaptive rule base for the generation of game AI on the fly. It improves the quality of scripted AI by online learning which can be very effective in complex games. This type of technique succeeds and scales well where AI has to be controlled according human players skill level [8].

Game Environment

We have implemented our AI in the game *StarLight*, a space shooter inspired by the classic arcade game *Asteroids*. The game environment is set in three dimensional space and occupied by the player (a ship) and obstacles (asteroids). The game engine's virtual camera remains in a fixed position, aimed in a "top-down" manner. The stationary nature of the camera effectively reduces the game environment to two dimensions, denoted as (x, y).

Representation

StarLight consists of objects represented with three dimensional polygonal geometry. Each object has an associated (x, y, z) location and rotation (r_x, r_y, r_z). Due to the previously mentioned "top-down" camera setup, the AI engine accepts input as a (x, y) location and angular orientation (z-axis rotation) values.

Structural Design

StarLight is constructed using an object oriented programming approach. Graphics rendering, collision detection, and animation are handled by the game engine, while the AI code is a new self-contained module.

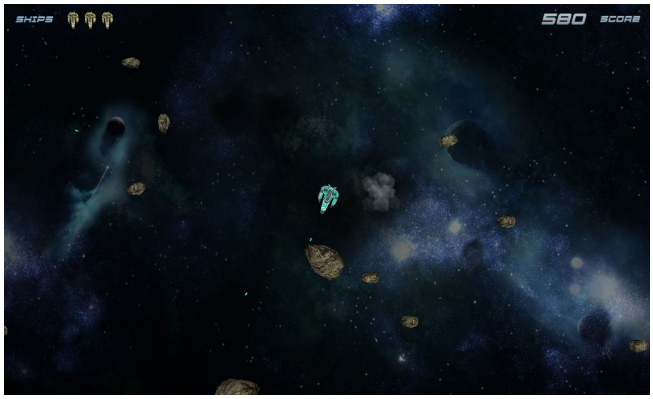

Figure 1: *StarLight* screenshot

The AI module is represented as a C++ namespace and contains four additional namespaces. The "data" namespace contains the Kinematic data structure, the "behaviors" namespace contains the movement algorithms for Seek and Align. The "states" namespace holds the possible states the AI can exist in for use in the priority system and FSM. The "arbitration" namespace holds the Priority manager and FSM classes. Finally, the "utils" namespace holds a vector class used during steering calculations.

The AI module communicates with external game components through a predefined interface. Specifically, the AI engine outputs a kinematic data structure that is then used in the games update loop. Due to the game's original structure, in rare occasions the AI algorithms dispatch events to notify the game to perform an action (i.e. when a laser shot should be created).

Collision Detection

Since collision detection is a key component related to the evaluation of the AI, we provide a brief primer on the engine's collision system. Collision is handled in three dimensional space as a two phase process. First, a broad-phase test compares spherical volumes that contain each mesh object. The formula is expressed as:

$$(r_1 + r_2) < d \tag{1}$$

r_1 = spherical radius of object 1
r_2 = spherical radius of object 2
d = the distance between the centers of each object

The broad phase test provides an efficient rejection condition, increasing the overall performance of the collision system. If the broad phase test evaluates true; however, a more precise check is required. To accomplish this, a narrow-phase test using a single tight fitting oriented bounding box

(OBB) is executed. The test will search for a separating axis (plane) between the objects. It is important to note that this form of collision detection is still approximate and will not always return an exact match to visual expectations.

Additional Details

StarLight is a Windows 7 PC game built for a NCSU CSC562 Graduate Computer Graphics course in the Fall of 2011. The game is built in Microsoft Visual C++ and OpenGL. The game engine is a small scale 'homebrew' engine with which the authors are familiar.

Implementation

After determining the structural design of the code, we developed each of the NPC character's behaviors, followed by the arbitration system.

Avoid (Flee)

The primary goal of the NPC ship is to stay alive. The avoid algorithm mimics a human player's actions to evade asteroids. Our NPC includes a radius of evasion originating from the ship's centroid. The radius of evasion acts as a look-ahead giving valuable information to the AI about the state of the game world. The quality of this information; however, is highly dependent on the size of the radius. Increasing the radius provides information earlier, but is also less relevant to actions the ship needs to take in the given moment. Balancing this relationship comes down to mostly trial and error and is highly dependent on the game itself. This is discussed in further detail in the evaluation.

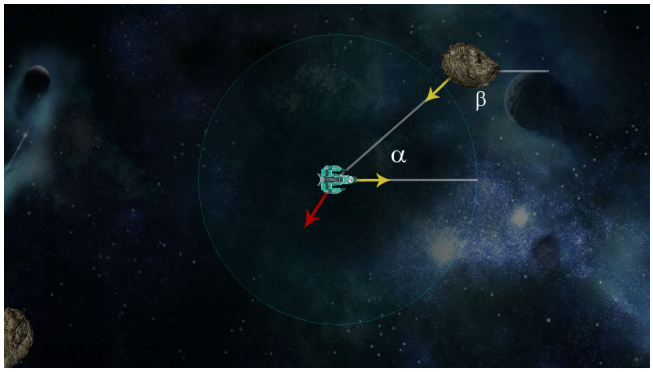

Figure 2: Avoid mechanics

When in danger a fleeing direction is calculated, the ship is oriented, and acceleration is applied. The fleeing direction is calculated in multiple steps. First, the angular relationship between the incoming asteroid and the NPC ship, α, is determined. Next the angular relationship and the asteroid direction, β, are mapped to the smallest available angle. Figure 2 illustrates the values α and β.

After mapping, if the two direction values have opposite signs, the target fleeing direction is set to the direction the asteroid is traveling.

$$if(\alpha/\beta < 0) alpha_{final} = \beta \quad (2)$$

This accounts for all but one possible situation. If the asteroid and ship directions are equal, or relatively close to equal, and the ship is behind the asteroid we must ensure the ship does not accelerate into the asteroid. We determine if the ship direction is within a range of the asteroid direction with Equation 3.

$$cos(\beta - avoidance) < \alpha < cos(\beta + avoidance) \quad (3)$$

In our implementation, avoidance is randomly chosen floating point value between positive or negative 15 and 30 degrees. If the ship direction is within the avoidance range, we set the ships target fleeing direction with Equation 4.

$$alpha_{final} = \alpha + (avoidance) \quad (4)$$

Now that the ship is oriented, it must accelerate away. If the NPC ship boosts away at constant acceleration, the max speed will be attained very quickly - giving the ship little time to react to other possible collisions. By watching human players, we noticed players applying quick amounts of acceleration to avoid game obstacles - without *holding* down the "accelerate" key. To mimic this, we apply a single vector of acceleration in the evade direction. The acceleration is a custom amount, overriding the default acceleration value, that we have found looks good through trial and error. This acceleration is displayed in Figure 2 as the red arrow.

Attack

A secondary goal of the NPC ship is to accumulate points by destroying asteroids. To accomplish this goal, the NPC has a defined radius of attack originating at the ships centroid. When a target (asteroid) is within the radius, the NPC is now in danger and takes actions to destroy the threat. This is a reactive, defense oriented attack model, as opposed to an active "seek-and-destroy" model. The behavior uses the centroid of an asteroid to compute the required alignment (orientation) of the NPC ship in the direction of the target, α.

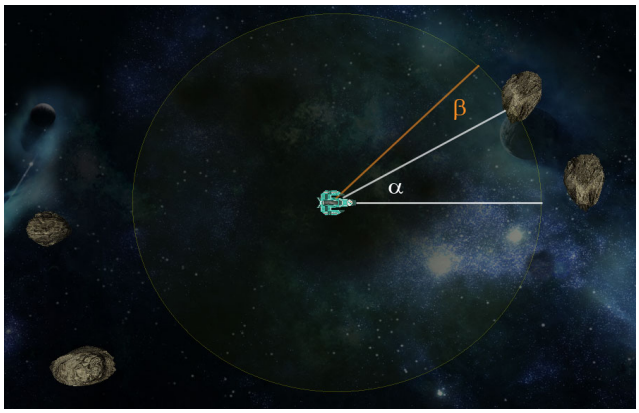

Figure 3: Attack mechanics

A ship that shoots with 100% accuracy doesn't accurately mimic human behavior. A probability, P_a, is introduced to model shot accuracy; adding a level of artificial error to the NPCs behavior. The probability value P_a, which lies in the range [-1, 1] and is multiplied with a predefined error angle β to produce the desired artificial error. Next, the artificial error will be added to angle α, producing the final target orientation α_{final}. Hence, a P_a of zero will produce an error of zero and a 100% accurate shot.

$$\alpha_{final} = \alpha + (P_a * \beta) \tag{5}$$

The above approach is quite effective when a single asteroid exists within the radius of attack. Due to the nature of the game environment; however, there are often many asteroids inside the attack range. Consequently, the NPC needs to decide which asteroid to attack before running the above calculations. This is discussed further in the evaluation.

Idle (Wander)

When the NPC ship is not pursuing its main goals, it will wander around the game environment. This addresses the situation where the NPC ship is not in immediate danger from an obstacle, but also doesn't have a target to attack in range. Wander is a unique behavior in our system, since it does not depend on reacting to the state of the environment. Wandering eventually places the NPC ship into a situation where other behaviors will be required.

This "idle" behavior implementation is composed of a wander steering behavior. A random location in the game world is selected which the ship then accelerates to. This location, (ω_x, ω_y) is calculated by multiplying a random binomial, β, in the range [-1, 1] by the extent of the game world. We use different random binomials for the x and y coordinates.

$$\omega_x = \beta_x * X_{BOUNDARY} \tag{6}$$
$$\omega_y = \beta_y * Y_{BOUNDARY} \tag{7}$$

The ship arrives at the location of interest when the point is within a predefined radius of satisfaction. Due to the implementation of ship movement, steering is performed slightly different than might be expected. Negative acceleration isnt generated by an arrive behavior because the ship is constantly decelerating in the games update loop.

Moving at constant speed to the location of interest would appear unnatural. In order to mimic this human behavior, we added a random delay in how often the NPC ship accelerates. We tweaked this variable through trial and error to find a good looking result. Adding the delay provided the level of randomness we wanted in the ships movement.

Priorities via Finite State Machines (FSM)

Once all the individual NPC behaviors were implemented and tested, they were integrated into a FSM priority system. Based on the current game state, the priority system determines the relative importance of each behavior and allows the most important one to execute. In our system, there are two important values to keep track of:

- Distance to the closest asteroid
- Radius distance for each behavior slice

Processing of asteroid distance is encapsulated in each individual behavior to allow for modularity. This allows for each behavior to potentially employ a different approach when determining which asteroid(s) to react to.

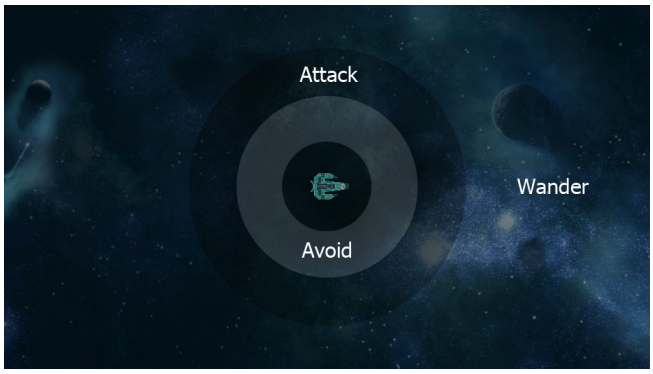

Figure 4: Radius visualization

In our implementation, each behavior takes a simple approach to asteroid importance: react to the closest asteroid. While there are more complex ways to solve this problem, we found this method gave us a reasonable behavior at a very low cost. The game engine collision detection algorithm already calculates the distance between the ship and each asteroid, so we modified the collision code to store this distance in each asteroid. With the distance values saved, all we needed to do was pass a reference to the asteroids collection to the respective behavior algorithm. This removed the need to recalculate distance values, was memory efficient, and worked generically for all behaviors. FSM states are

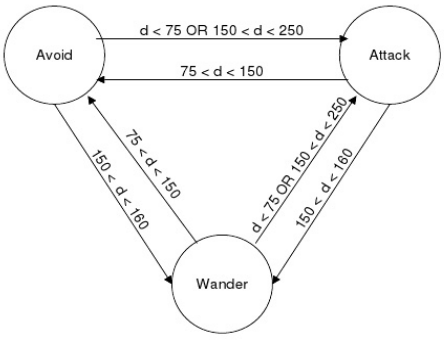

Figure 5: Finite State Machine

represented by radii, originating at the ship centroid. Figure 4 visualizes an example set of radii values for an NPC in *StarLight*. Notice how multiple "slices" are possible for each behavior. The boundaries between state radii represent transitions between states in the FSM. Figure 5 is a diagram

of the state machine resulting from Figure 4s radii configuration.

In our implementation, radius values are set statically during author time; however, recomputing these values at runtime based on game state is also possible. Since the behavior of our NPC character is relatively simple, the finite state machine provides a clean way to decompose the character's behavior into modules. The FSM limits the behaviors the NPC can execute to one at any given time, which makes the behavior easy to debug.

Evaluation

Intelligence is an inherently difficult quantity to measure. Standardized tests, such as the Graduate Record Exam (GRE), have been devised to quantify the relative intelligence of humans; however, they ultimately are incapable of accurately capturing every aspect of the intelligence one possesses. In games, the primary concern is the appearance of intelligence, as opposed to the actual intelligence of NPCs. This insight proves useful when developing an approach to evaluate how the AI is perceived by players.

To evaluate the NPC artificial intelligence, we have measured game success conditions including time survived and score obtained. Additionally, we are concerned with the believability of the behavior the AI exhibits. Specifically, we are focusing on the following five traits [9]:

- Displays emergent behavior, minimal designer input
- Decides to act autonomously
- Motivated by both internal and external situational factors
- Capable of altering, and responding to changes in, the properties of other game objects
- Performs efficiently so that gameplay is not affected

We begin our evaluation with quantitative data, followed by a qualitative evaluation of the NPC's believability. Table 1 below displays the result of 10 trial runs of the AI where all behaviors are active in the priority system.

Table 1: Data

Trial	Time (Minutes)	Score	Level
1	7:23	4380	3
2	3:54	2930	3
3	3:28	2640	3
4	1:06	590	1
5	8:47	2940	3
6	4:33	3580	3
7	12:49	12440	6
8	13:02	4910	4
9	11:13	6830	4
10	3:26	1280	2

Time Survived

In order to evaluate the effectiveness of the FSM Priority system, we measured the survival time of individual behaviors versus the FSM Priority system running all behaviors. Table 1 and Figure 5 show the results of running only the avoid behavior. Time is measured in seconds. Figure 6 shows the time survived, in minutes, when all behaviors are running and prioritized. When running just the attack behavior, the ship survives for an upwards of 15 minutes. In this situation, the ship is stationary and constantly attacking; therefore in danger much less often.

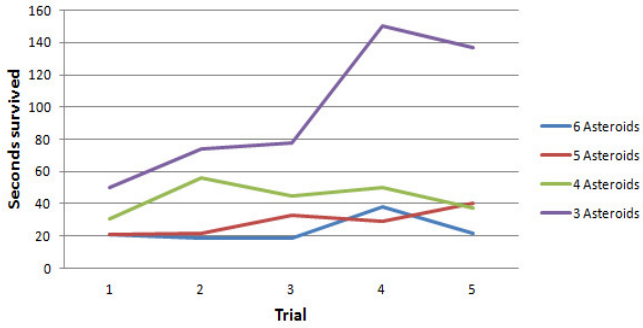

Figure 6: Time Survived, only Avoid

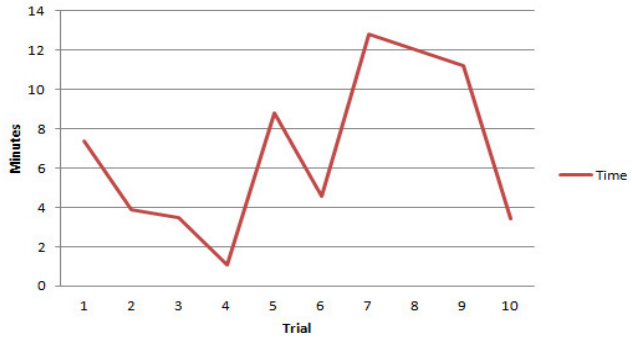

Figure 7: Time Survived, Priorities via FSM

From purely a time perspective, we can conclude that the FSM Priority System blends the behaviors together well. The AI neither stays alive too long, nor does it die quickly. Using the FSM Priority System, the average time survived is 6 minutes 48 seconds.

Score Obtained

In addition to staying alive, the AI must accumulate points by destroying asteroids. Similar to the time survived metric, similar tests for score obtained were performed.

Figure 8 shows the scores obtained by only the attack behavior over multiple trials. Figure 9 displays the the score

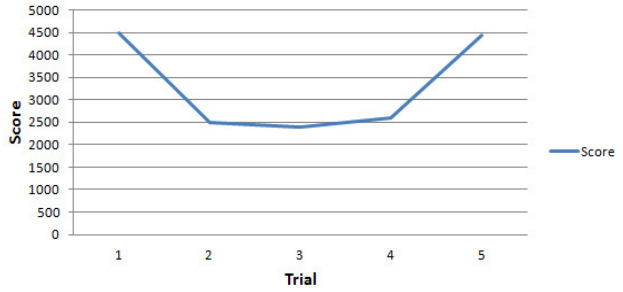

Figure 8: Score Obtained, only Attack

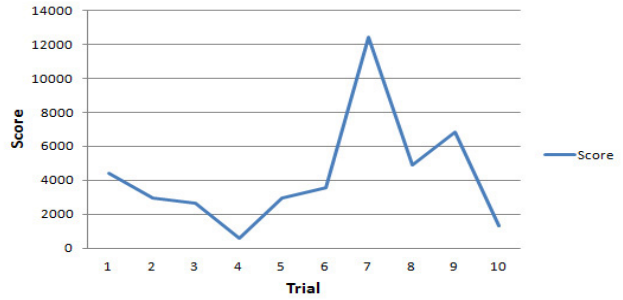

Figure 9: Score Obtained, Priorities via FSM

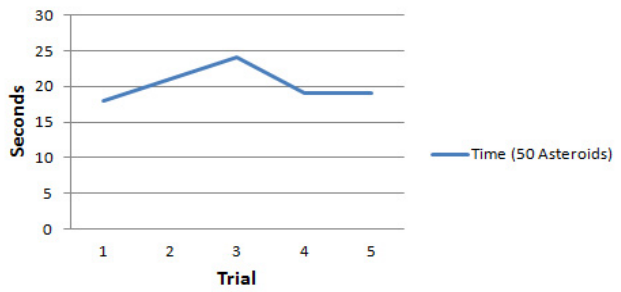

Figure 10: Time Survived, 50 Asteroids

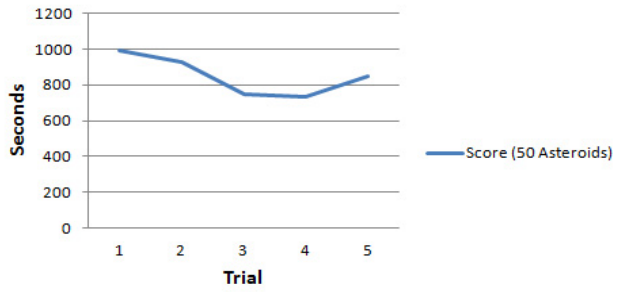

Figure 11: Score, 50 Asteroids

obtained with the FSM Priority System over multiple test trials. The average score with the priority system is 4,252 points.

Complexity and Performance

At the beginning of the project, we anticipated the runtime complexity of the AI implementation would be computationally significant. While running the above test trials; however, we noticed almost no change in the frame time, frames per second, and system memory used. We attribute this to the fact that the AI system is not recomputing asteroid-ship distance, there are no nested loops, and the AI system itself creates very few objects.

Stress Tests

In the feedback we received, a common test suggestion was to place a large number of asteroids on screen and observe the NPCs behavior. The results of these stress tests are below:

Evaluating Believability

In *StarLight*, there are various strategies a player can employ to accomplish the game's goals. We generalize these strategies into defensive and offensive movements. A defensive player moves slowly and purposefully. This strategy values precise control over the ship and the ability to quickly evade. Offensive player movements are faster, place the ship in a position to attack, but affords less control over the ship in dangerous situations.

Our AI implementation employs the defensive strategy in order to maximize survival time and points scored. When an increasing number of asteroids fill the screen, this strategy keeps the ship alive and works well. Despite this, the strategy can also lead to situations where the ship appears to be overly cautious. While this effectively accomplishes the success conditions of the game, it can make the NPC appear to be less realistic.

For example, an edge situation where the defensive strategy breaks down is when there is a small number (1-2) of asteroids left on the screen. The AI still actively avoids asteroids, despite there being little actual danger. Human players; however, are very good at seamlessly transitioning between strategies of play. In this instance, a human player quickly recognizes they hold an advantage and transition to an offensive strategy. A possible remedy for this behavior is the aforementioned dynamic computation of radii values based on game state. This tactic could effectively approximate a human players adaptation.

Limitations

The NPC ship is inherently limited to the behaviors it can perform. It is able to wander, avoid, and attack, but cannot perform any other actions. For a small to medium number

of asteroids, the ship AI works effectively. As the number of asteroids increases; however, it becomes increasingly difficult for the AI to accomplish its tasks. This is also true for the human player; however, a human player can much more dynamically adapt the the game environment.

Coordination with a human player, other AI agents, and learning are all areas left for future work. While our goal was simplicity, more complex behaviors such as target leading (asteroid prediction) and behavior layering could yield more interesting results.

Future Work

While the behavior exhibited by the NPC is successful, there are areas of improvements as noted in the limitations. The first improvement we plan on is dynamically recomputing the radii sizes associated with avoid, attack, and wander. This will give the ship AI more appropriate behaviors during edge conditions where there are either many or very few asteroids on the screen. Additionally, implementing advanced behaviors like pursue when the number of asteroids fall below a certain threshold can improve these edge conditions.

Conclusions

We have implemented an effective yet simple NPC using basic movement algorithms. An intuitive finite state machine arbitration system blends avoid, attack, and wander behaviors together to produce a workable NPC AI. Our approach is simple to implement and is computationally lightweight enough for low-power mobile devices. The behaviors in the system can be tweaked and adapted both at author and runtime. Additionally, the AI can be applied in multiple contexts. This could include a co-op NPC that plays with a human player or against the player as alien ship opponents.

While there is certainly room for improvement in the available behaviors, the basic movement algorithms provide a surprising amount of behavior relative to their cost. By successfully integrating the NPC into *StarLight*, we demonstrate that the illusion of sophisticated behavior can be added to a game with only basic movement algorithms.

References

[1] Bong-Keun Lee; Choong-Shik Park; Jae-Hong Kim; Sang-Jo Youk; Ryu, K.H.; , An Intelligent NPC Framework for Context Awareness in MMORPG, International Conference on Convergence and Hybrid Information Technology, Aug. 2008, pp.190-195, 28-30.

[2] Brian Schwab. *AI Game Engine Programming*, Charles River Media, 2004.

[3] S.J. Russel, P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice Hall, 2002.

[4] Mondesire, S. and Wiegand, R.P., Evolving a Non-playable Character team with Layered Learning, IEEE Symposium on Computational Intelligence in Multicriteria Decision-Making, April 2011, pp.52-59, 11-15.

[5] Hartley, T.P. and Mehdi, Q.H., In-game tactic adaptation for interactive computer games, 16th International Conference on Computer Games, July 2011, pp.41-49, 27-30.

[6] Xiao Liu; Yao Li; Suoju He; Yiwen Fu; Jiajian Yang; Donglin Ji; Yang Chen; , To Create Intelligent Adaptive Game Opponent by Using Monte-Carlo for the Game of Pac-Man, Fifth International Conference on Natural Computation, 2009.

[7] Orkin, Jeff. Three States and a Plan: The A.I. of F.E.A.R., Cambridge, MA: Monolith Productions / M.I.T. Media Lab, Cognitive Machines Group, 2006.

[8] Spronck, Pieter, Marc Ponsen, Ida Sprinkhuizen-Kuyper, and Eric Postma. Adaptive Game AI with Dynamic Scripting. Machine Learning 63.3 (2006): 217-248, Mar. 2012.

[9] Reed, Kevin. Educating the Masses: Improving the Intelligence of Non-Playable Support Characters in Video Games, Ipswich, UK: UNIVERSITY CAMPUS SUFFOLK, 2011.

[10] Van Lent, M., *Game Smarts*, Computer, vol.40, no.4, April 2007, pp.99-101.

[11] D. Conroy, P. Wyeth, and D. Johnson. Modeling player-like behavior for game AI design, Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology (ACE '11), New York, NY, USA, , Article 9 , 8 pages. 2011.