Real-Time Game Design of Pac-Man Using Fuzzy Logic

Adnan Shaout, Brady King, and Luke Reisner
Department of Electrical and Computer Engineering, University of Michigan, USA

Abstract: *This paper presents the design and implementation of a real-time fuzzy-based artificial intelligence system for an interactive game. The chosen game is a remake of Pac-Man in which the opponents are BDI-style intelligent agents. The components of the system and the methods used in fuzzifying the game's rules and variables are discussed. In addition, comparisons are drawn between the proposed fuzzy solution and other crisp and fuzzy approaches.*

Keywords: *Fuzzy logic, artificial intelligence, intelligent agents, real-time, interactive, game.*

Received April 14, 2005; accepted July 12 2005

1. Introduction

Interactive games have become increasingly popular over the years. At the same time, people's expectations for the quality of games have been growing. One of the most famous real-time games of all time is Namco's Pac-Man. It was so popular that it actually caused a Yen shortage in Japan shortly after its release in 1980. In the game, the player controls a character called Pac-Man who navigates a maze populated by four computer-controlled opponents, called ghosts. The player must eat pellets to score points while avoiding the ghosts, whose purpose is to hunt down and eliminate Pac-Man. Initially, it was acceptable for computer opponents to base their behavior on simple, non-adaptive logic. For instance, the enemies in games such as Super Mario Bros. paced back and forth, regardless of the player's movements. In more recent games, players have come to expect the opponents to respond intelligently to the player's actions.

One approach to solving the problem of game intelligence is the application of fuzzy logic. Fuzzy logic is a logical system that works with data that does not have precisely defined values [11]. Fuzzy systems typically employ rules to translate vague terms, such as skill or comfort, into system outputs. In the case of games, fuzzy rules can be used to determine computer-controlled actions based on various player behaviors and system conditions.

For this paper, the classic game Pac-Man was chosen to demonstrate a fuzzy logic based artificial intelligence system. In earlier implementations of this game, the ghosts' logic did not realistically adapt to the user's skill and movements. For instance, ghosts didn't move towards areas where Pac-Man needed to go to complete the level (i. e., areas with many pellets). While this could be done with classical logic, fuzzy logic provides a more elegant way for a system to deal with the often ambiguous data required to implement such behaviors. In addition, this type of system allows rules to be easily added to increase the opponents' intelligence further. For these reasons, fuzzy logic has been chosen as the basis for the intelligent control of the ghosts' behavior.

The following sections of this paper will present the complete design and implementation of a fuzzy rule based version of Pac-Man. In section 2, the design of each of the three main tasks of the proposed system will be explained. The description will follow the flow of execution of the code through each task. Section 3 will present the design details of the rules that govern the ghosts' behavior. Next, section 4 will present the fuzzy linguistic variables for both the inputs and outputs of the fuzzy system. Section 5 will present the method of defuzzification and behavior selection. After that, section 6 will discuss the results observed in the implementation of the game along with a comparison to classical design methodologies. Finally, section 7 will provide a few general comments and suggestions for future work.

1.1. Relations to Existing Work

Clearly, the fuzzy implementation of Pac-Man described in this paper is not the first use of fuzzy logic in a game control application. For a number of reasons, such as increases in microprocessor speed, the use of fuzzy logic (and other artificial intelligence techniques) has become increasingly common in recent years [9]. Thus, it is important to understand the relationships between ideas presented in this paper and existing work.

The ghosts used in this system are examples of intelligent agents. Intelligent agents are autonomous entities that exhibit flexible behavior in pursuit of their objectives [8, 22]. They have been used in a variety of applications, ranging from soccer-playing robots [2] to Internet commerce [23] to weather reports [14].

Several different frameworks exist for the design and implementation of intelligent agents. Three of the most popular frameworks are Beliefs-Desires-Intentions (BDI), Goals, Operators, Methods, and Selection (GOMS) rules, and Soar [10]. Each framework has a unique background and different ways of managing information and behavior. For instance, GOMS is rooted in human-computer interaction, and BDI is based on logical theories of rational behavior.

The framework used in the fuzzy implementation of Pac-Man most closely resembles the BDI model [5]. The "beliefs" or knowledge of the intelligent ghost agents includes the location of other game entities and the performance of the player. In this simple game, the primary "desire" or goal of the ghosts is to intercept the player. Finally, the "intentions" or courses of action of the ghosts include hunting Pac-Man, guarding pellets, and avoiding other ghosts.

The BDI-style framework was chosen for its relative simplicity. The intelligent agents in Pac-Man do not require a large rule base or complex human-like reasoning to perform their task. Attempting to fit Soar or a GOMS model to this small game would have required significantly more work for little or no benefit.

Although many papers have been written regarding fuzzy logic and topics like game theory [7, 17, 21], only a few papers have dealt with fuzzy control in a real-time game. One such paper discusses the implementation of a game called BattleCity.net using intelligent fuzzy agents [13]. Like the fuzzy version of Pac-Man, BattleCity.net uses BDI-style intelligent agents that are governed by fuzzy rule-based logic. However, BattleCity.net lacks any human interaction, which is one of the important aspects of fuzzy game design explored in this paper. Another difference is that BattleCity.net utilizes either an exhaustive rule base or one based on the debated Union-Rule Configuration (URC) [3, 4, 15], whereas the fuzzy rule base in Pac-Man (see section 3) does not have complete coverage of all possible combinations of fuzzy inputs. This was done to simplify the design and reduce the computational requirements, which grow exponentially with the number of fuzzy inputs if exhaustive coverage is used.

Another paper describes the design of intelligent game characters using principles derived from behavior-based control of mobile robots [19]. The authors present a fuzzy rule-based system for dictating the behavior of enemy and non-enemy characters, similar to the fuzzy implementation of Pac-Man. However, their fuzzy system uses the Takagi-Sugeno model [20], whereas the one detailed in this paper is based on the Mamdani model [11]. Additionally, the authors do not present a method for tuning behavior selection like the weighting system described in section 5 of this paper.

2. Design Overview

Like many real-time systems, the design of the proposed system was broken into several (three) tasks or threads [12]. The first task is the initialization and timing thread. The purpose of this thread is to initialize the game, run the main menu, and then handle the scheduling of the control thread. The second task is the input thread. The sole purpose of this task is to read user input from the keyboard as quickly as possible. The final task is the control thread, which executes most of the game logic. These three tasks run in parallel to produce a fully functioning system.

2.1. Initialization and Timing Thread Design

The first function the initialization thread will execute is the loading of levels. Levels will be read from text files with file name of the format "level*xx*.txt," where "*xx*" is a number between 1 and 99. Consequently, the loading of levels can occur in a loop in which a file open attempt is made on "level*xx*.txt," where "*xx*" starts at 1 and is repeatedly incremented by 1.

To store each level, a level class is defined. The class has variables to store the initial x and y coordinates for the ghosts and Pac-Man, the initial direction for Pac-Man, the x and y level size, the total number of pellets, and the level map. The level map is implemented as a two-dimensional vector of characters. A vector of level objects is used to store all the levels read from the text files.

Once all of the available levels have been loaded (and assuming there were no errors in the process), the next function of the initialization thread is to display the main menu. The main menu allows the player to select a level of difficulty, start a game, or exit the game. The menu is implemented via text input and output in the standard console. The pseudocode shown in Figure 1 demonstrates the flow of the main menu.

The next step of the initialization thread is to initialize the global variables used by the system. The number of lives, number of ghosts, time between game updates, and ghost behavior weights are set based on a selected difficulty level (easy, medium, or hard). Additionally, the score is set to zero and all variables used in the calculation of the fuzzy logic variables are set to their initial values.

Next, the other two system threads (input and control) are created. This is done with two calls to the CreateThread function. The "starting address" parameters of the CreateThread calls are set to the functions for the input and control threads.

At this point, the initialization and timing thread is ready to begin its scheduling duty for the control thread. The control thread must be run at a fixed periodic rate to handle the game logic. Each period is referred to as one "tick" of the game. For the fuzzy implementation of Pac-Man, it was decided that the control thread should be executed every 250 ms, 375 ms, or 500 ms for the easy, medium, and hard difficulty levels, respectively. Since the timing for thread scheduling provided by Windows is only accurate to approximately +/-50 ms, a combination of Windows thread scheduling and busy waiting is used. Busy waiting utilizes the Windows performance counter for highly accurate timing. The pseudocode shown in Figure 2 illustrates the scheduling of the control thread.

Now that the thread has reached its scheduling portion, it does not need to perform any more initialization. Thus, the timing thread remains in the loop shown in Figure 2 until the program exits.

```
Difficulty = MEDIUM;  // Default
While (true)
    Output ("Press 's' [start], 'd' [change difficulty], or
            'q' [quit]");
    Output ("Current difficulty:", Difficulty);
    Read input char from user;
    If (char == 's') then
        Exit Loop;
    Else If (char == 'd') then
        Output ("Press 1 [Easy], 2 [Medium], or 3
        [Hard]");
        Read input char from user;
        Translate char and store in Difficulty;
        Output ("Difficulty changed to:", Difficulty);
        Continue;
    Else If (char == 'q') then
        Exit Game;
    End If;
End While;
```

Figure 1. Pseudocode for the main menu.

2.2. Input Thread

The input thread is responsible for reading input from the keyboard during game execution. Characters from the keyboard are read using a blocking I/O function. Standard C++ requires the "enter" key to be pressed for input to be accepted. This is unsatisfactory because users expect their input to be accepted when any key is pressed. Thus, the input console is configured to accept input after each individual key is pressed. Also, it is desired to prevent the input from being echoed on the screen. This is accomplished by using the SetConsoleMode function [16] with arguments to disable the "line input" and "echo input" options of the standard C++ input console.

Once a key is pressed, the input thread immediately stores the input in a buffer accessible by the control thread. No processing is done on the input within the input thread. Any future input will overwrite the current contents of the buffer.

```
// Get the start time:
QueryPerformanceCounter (&start_time);
While(true)
    // Suspend the timing thread:
    Sleep (tick_period - 60ms);
    Do  // Busy waiting
        QueryPerformanceCounter (&current_time);
        While (current_time - start_time <tick_period);
        ResumeThread (control_thread);
        start_time += tick_period;
End While;
```

Figure 2. Pseudocode for scheduling the control thread.

2.3. Control Thread

The control thread is the heart of the game. All the data-processing and decision-making is done in the control thread, which is a large infinite loop. The first part of the control thread checks if a new level needs to be initialized. This happens when a level ends or a new game has begun. Specifically, a level ends when the number of remaining pellets is equal to zero.

To initialize a level, the level needs to be loaded and configured. To accomplish this, the level is copied out of the level vector and into a global object that represents the current level. In addition, all the characters are placed at their starting locations. Specifically, Pac-Man's starting location (x, y), the ghosts' starting location (x, y), and Pac-Man's starting direction are initialized to the values stored in the level object. Furthermore, the pathfinding algorithm (described below) and the number of remaining pellets on the level is initialized.

The next part of the control thread processes the input acquired from the player via the input thread. The control thread creates a copy of the current contents of the input buffer to ensure the contents are not changed during processing. If a directional key is pressed ('a', 's', 'd', 'w' for left, down, right, and up, respectively), this direction is translated into the new direction for Pac-Man to travel. If the 'l' key is pressed, the number of remaining pellets is set to zero to force a level change. Finally, if 'q' is pressed or Pac-Man has no remaining lives, the game quits with a message showing player's final score.

After processing the input, the control thread checks if Pac-Man is "powered up." Pac-Man becomes temporarily powered up after eating special "power pellets," which enable him to eat the ghosts. If Pac-Man is powered up, the power time is decremented once for the current control cycle. Otherwise, the number of ghosts eaten during the last "powered up" period is reset.

The next part of the control thread calculates the ghosts' and Pac-Man's new positions on the game grid.

Pac-Man's new position is based on the player input captured during the first part of the thread. For a given direction, the code checks first if the move is valid. A move is invalid if the target location is a wall. If the move is valid, the code sets Pac-Man's new position to the valid location. If the move is invalid, Pac-Man's new position is set to the same position in which he currently resides.

At this point, the ghosts' directions have already been set during the previous game tick based on the fuzzy controller. The code that does this is located near the end of the control thread. During that section of code, which will be discussed later, the validity of the ghosts' movements is checked. Therefore, the positions of the ghosts are updated at the current time without checking for validity.

After the new positions have been calculated, the code checks for a collision between Pac-Man and a ghost. A collision occurs under either of the following conditions:

- Pac-Man and a ghost now occupy the same location on the game grid.
- Pac-Man and a ghost have swapped positions.

If a collision has occurred, flags are set to indicate the collision and which ghost has been hit.

The next block of code handles pellet consumption. The execution of this section is conditional on the fact that Pac-Man has not died (i. e., Pac-Man has not collided with a ghost, or he has but is "powered up"). If Pac-Man's new position contains a regular pellet or power pellet, the pellet is eaten. This requires graphically overwriting the pellet on the current level map with a blank space, decrementing the total number of remaining pellets, increasing Pac-Man's score, and playing the "pellet eaten" sound effect. In addition, if the pellet is a power pellet, Pac-Man's "powered up" time is set to 30 game ticks. The sound effect is played via the PlaySound function of the Windows API [16]. To provide low priority, asynchronous sound generation, the following parameters are passed to the PlaySound function: SND_NODEFAULT, SND_NOSTOP, SND_NOWAIT, and SND_ASYNC.

After handling pellet consumption, the code takes care of ghost consumption. Specifically, if a collision is made while Pac-Man is "powered up," the "ate ghost" sound effect is played, the score is increased, and the ghost that was eaten is reset to its initial location on the game grid. The sound is played in the same fashion as the "pellet eaten" sound described above, but the SND_NOSTOP and SND_NOWAIT options are omitted to increase its priority over the "pellet eaten" sound effect.

At this point, the display is updated. First, the current score and lives are printed to the screen via the console. Next, the current level map is printed using the pseudocode shown in Figure 3. Next, Pac-Man is displayed at his current location provided he has not died during the current iteration of the control thread. The SetConsoleCurrentPosition function is used to set the cursor to the current location of Pac-Man, and the character corresponding to Pac-Man's current direction is printed. Next, each ghost is displayed in the same fashion as Pac-Man, but with their own unique characters. Finally, if Pac-Man has died, a death message is printed at the bottom of the screen.

```
For (each row of the map)
    For (each column of the map)
        Output (current map character);
    End For;
Output (new line);
End For;
```

Figure 3. Pseudocode for the output of the level map.

The next portion of the control thread plays the "game start" sound effect if a new game was just started. This sound is played in the same fashion as the "ate ghost" sound effect. Following the "game start" sound effect, the decision logic for ghost movement is executed. The first step of this process is to determine if any of the ghosts are at an intersection. If a ghost has three or more non-blocked paths, a decision must be made as to which path the ghost should follow. Otherwise, the ghost will continue along the path it was previously following.

The ghost decision logic is conditional upon the state of Pac-Man. If Pac-Man is not "powered up," the fuzzy system (see sections 3 through 5) is employed to select one of four behaviors. The behavior is then used to determine the direction that the ghost will move during the next iteration of the control thread. However, if Pac-Man is powered up (i. e., his "power pellet time" is greater than 0), the ghosts will always attempt to avoid Pac-Man.

If the fuzzy controller selects the "hunting" behavior for a ghost, the A* algorithm is employed to find the direction of the shortest path to Pac-Man [6, 18]. The pseudocode shown below in Figure 4 illustrates the A* shortest path algorithm. Note that the "Source" is the ghost and the "Target" is Pac-Man.

If the fuzzy system selects the "defense" behavior for a ghost, the ghost must move in a direction that will take him towards the area of the map with the highest pellet density. To find this area, the following calculations are done:

- Divide the map into 9 overlapping sections based on combinations of the following fractions of the x size and y size of the level map:

  0 to ½, ¼ to ¾, and ½ to 1

- Sum the total number of pellets in each section.
- Select the section with the highest number of pellets.

- Return the coordinates of the middle pellet in that section:
    - The middle pellet is found by traversing the pellets in that section from left to right, top to bottom and stopping when the number of pellets encountered is half the total number of pellets in that section.

The A* algorithm is then used to determine the direction of the shortest path to the returned pellet. This is the direction the ghost is assigned to take.

If the fuzzy algorithm selects the "shy ghost" behavior for a ghost, the ghost needs to move away from the closest ghost. The following calculations are done to determine the required direction:

- Calculate the city-block distance between the source ghost and the other ghosts.
- Select the ghost that is closest to the source ghost.
- Determine the differences in x and y location between the source ghost and the closest ghost.
- If the x difference is greater than the y difference:
    - If the square in the x direction from the source ghost away from the closest ghost is not blocked, return the direction from the source ghost to that square.
    - Else if the square in the y direction from the source ghost away from the closest ghost is not blocked, return the direction from the source ghost to that square.
    - Else return one of the two remaining directions (whichever leads to a path that is not blocked).
- Else the y difference is greater than the x difference:
    - If the square in the y direction from the source ghost away from the closest ghost is not blocked, return the direction from the source ghost to that square.
    - Else if the square in the y direction from the source ghost away from the closest ghost is not blocked, return the direction from the source ghost to that square.
    - Else return one of the two remaining directions (whichever leads to a path that is not blocked).

If the fuzzy controller selects the "random" behavior for a ghost, the ghost is supposed to move about the level randomly. This is accomplished by using a random number generator to select one of the four directions randomly. If that direction is blocked, the other directions are iterated through until a non-blocked path is found.

If Pac-Man is "powered up," the ghost decision logic must move the ghost away from Pac-Man. The same algorithm that is used for the "shy ghost" behavior is used to accomplish this task. The only differences are that there is no need to find the "closest ghost," and the "closest ghost" is replaced with Pac-Man.

If the fuzzy controller selects the "random" behavior for a ghost, the ghost is supposed to move about the level randomly. This is accomplished by using a random number generator to select one of the four directions randomly. If that direction is blocked, the other directions are iterated through until a non-blocked path is found.

If Pac-Man is "powered up," the ghost decision logic must move the ghost away from Pac-Man. The same algorithm that is used for the "shy ghost" behavior is used to accomplish this task. The only differences are that there is no need to find the "closest ghost," and the "closest ghost" is replaced with Pac-Man.

```
If (Target == Source) then
   Return INVALID;
   Add Source to Open List (GCost = 0, HCost = cityblock
    dist from Source to Target,
   Parent = Null)
   While (Open List is not empty)
      Current = Item in Open List with lowest (GCost +
                   HCost)
      If (Current == Target) then
        Break;
      End If;
      For (each of 4 squares adjacent to Current)
         If (square is not a wall) then
           If (square not already on Open List) then
              Add square to Open List (
              GCost = Current GCost + 1,
              HCost = cityblock dist from square to Target,
              Parent = Current)
           Else
             If (Current GCost + 1 <
                GCost of square on Open List) then
                GCost of square on Open List = Current GCost
                 + 1;
             End If;
           End If;
         End If;
      End For;
         Move Current from Open List to Closed List;
   End While;
If (Open List is empty) then
   Return INVALID;
End If;
Retrace Path from Current to Source;
Return direction from Source to the best path to Current;
```

Figure 4. Pseudocode for the A* shortest path algorithm.

If the fuzzy controller selects the "random" behavior for a ghost, the ghost is supposed to move about the level randomly. This is accomplished by using a random number generator to select one of the four directions randomly. If that direction is blocked, the

other directions are iterated through until a non-blocked path is found.

If Pac-Man is "powered up," the ghost decision logic must move the ghost away from Pac-Man. The same algorithm that is used for the "shy ghost" behavior is used to accomplish this task. The only differences are that there is no need to find the "closest ghost," and the "closest ghost" is replaced with Pac-Man.

3. Behaviors and Fuzzy Rules

In order for the ghosts to act more intelligently, four different behaviors for the ghosts were chosen. Each individual ghost chooses his behavior based upon a fuzzy logic model that is similar to the popular Mamdani model [11]. The model will be described below.

The first behavior implemented is the "hunting" approach. In this behavior, the ghost will actively seek Pac-Man using the A* path finding algorithm [6, 18]. The second behavior is the "defense" approach. For this behavior, the ghost will defend the area that has the most pellets, thereby ensuring that Pac-Man must come near the ghost to complete a level. The third behavior is the "shy ghost" approach. In this behavior, a ghost will move in a direction that will take it away from a nearby ghost. This behavior ensures that ghosts will spread out and cover the entire level. Finally, the "random" approach is chosen when no other method is a preferred choice. In this approach, a ghost will randomly move about the level.

Each of these behaviors has a set of fuzzy rules that contribute to the likelihood of choosing that behavior. The inputs to the rules are a number of linguistic terms that are derived from player performance and game conditions. The definitions of the linguistic terms will be given in section 4.

The following list of rules contributes to the "hunting" behavior:

- If (pacman_near AND skill_good) then hunting_behavior
- If (pacman_near AND skill_med AND pellet_med) then hunting_behavior
- If (pacman_near AND skill_med AND pellet_long) then hunting_behavior
- If (pacman_med AND skill_good AND pellet_long) then hunting_behavior
- If (pacman_med AND skill_med AND pellet_long) then hunting_behavior
- If (pacman_far AND skill_good AND pellet_long) then hunting_behavior

The following rules relate to the "defense" behavior:

- If (pacman_far AND skill_bad AND ghost_far AND pellet_short) then defense_behavior
- If (pacman_far AND skill_bad AND ghost_far AND pellet_med) then defense_behavior
- If (pacman_far AND skill_bad AND ghost_med AND pellet_short) then defense_behavior
- If (pacman_far AND skill_bad AND ghost_med AND pellet_med) then defense_behavior
- If (pacman_far AND skill_med AND ghost_far AND pellet_short) then defense_behavior
- If (pacman_med AND skill_bad AND ghost_far AND pellet_short) then defense_behavior

The following rules are associated with the "shy ghost" behavior:

- If (pacman_far AND skill_bad AND ghost_near AND pellet_short) then shy_ghost_behavior
- If (pacman_far AND skill_bad AND ghost_near AND pellet_med) then shy_ghost_behavior
- If (pacman_far AND skill_bad AND ghost_med AND pellet_short) then shy_ghost_behavior
- If (pacman_far AND skill_bad AND ghost_med AND pellet_med) then shy_ghost_behavior
- If (pacman_far AND skill_med AND ghost_near AND pellet_short) then shy_ghost_behavior
- If (pacman_med AND skill_bad AND ghost_near AND pellet_short) then shy_ghost_behavior

The following rule is related to the "random" behavior:

If NOT (hunting_behavior) AND NOT (shy_ghost_ behavior) AND NOT (defense_behavior) then random_behavior

Note that all of the previous rules take into account (directly or indirectly) a "skill" term. This variable refers to the player's skill, which is defined in terms of good skill, medium skill, and bad skill. The three skill levels are actually formed from another set of fuzzy rules. These intermediate rules must be calculated before the ghosts' behavior can be calculated. The following fuzzy rules are used to define the player skill variables:

- If (time_life_short OR pellet_rate_bad) then skill_bad
- If (time_life_medium OR pellet_rate_medium) then skill_medium
- If (time_life_long AND pellet_rate_good) then skill_good

For all of the fuzzy rules, the AND term refers to fuzzy intersection [11]. This is often implemented as the minimum or product operator. For this system, the minimum operator was chosen because it produces larger outputs for inputs within the range [0, 1]. The OR term used in the fuzzy rules refers to fuzzy disjunction. As in many fuzzy systems, the maximum operator was chosen for fuzzy disjunction.

4. Fuzzy Linguistic Terms

As seen in the rules listed in the previous section, the system makes use of many linguistic variables. The linguistic variables can be divided into three classes: Distance, time, and rate. Note that the player skill terms are formed from rules using linguistic variables from one of the three classes.

The levels in the implementation of the game are based on a two-dimensional grid. Some of the linguistic terms are based on the size of the current level map, called “level_size.” This quantity is defined as the maximum x dimension of the level plus the maximum y dimension of the level.

4.1. Distance Variables

There are two types of distance variables. The first type is the distance between Pac-Man and each of the ghosts, while the second is the distance between every possible pair of ghosts. The distance metric used in the system is the Manhattan or city-block distance scheme. This metric was chosen because its calculation is not computationally complex and it works well with two-dimensional grid-based structures.

There are three linguistic variable types for distance: Near, medium, and far. All of these variables are related to the size of the current level map. Membership functions are used to map the city-block distances into these fuzzy variables. These membership functions are simple linear functions (e. g., triangle and ramped-step) of the city-block distances. The same membership functions are used for the Pac-Man-to-ghost and ghost-to-ghost distances, which are defined as shown in Figure 5.

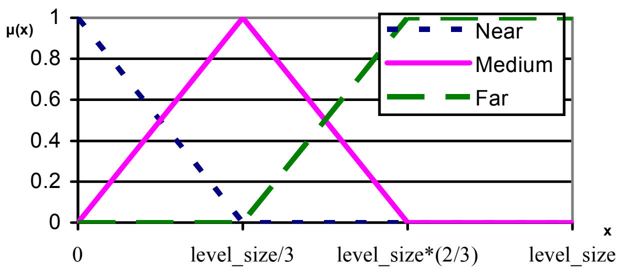

Figure 5. Membership functions for distance.

4.2. Time Variables

Two types of time variables are used in the proposed system. The first type is a measurement of the amount of time since Pac-Man has eaten a pellet and the second is the average period of time that Pac-Man has gone without losing a life. The unit of time used by the system is a fixed-length “tick.” Each tick represents one cycle of player and ghost movements. The actual amount of time represented by a tick is between 250 ms and 500 ms, depending on the selected level of difficulty. The tick was chosen as the time unit rather than real-world time because all game logic (fuzzy and crisp) occurs on discrete ticks.

As with distance, there are three linguistic variable types for time: Short, medium, and long. The membership functions for these time types are based on simple triangle and ramped-step functions. The thresholds for the functions are derived from the base time, which is the shortest possible time it would take Pac-Man to travel from a corner of the level to the opposite corner. Since Pac-Man can move one square per tick, the base time is equal to the level size. Using these facts, the membership functions for pellet time and average lifetime are defined as shown in Figure 6 and Figure 7.

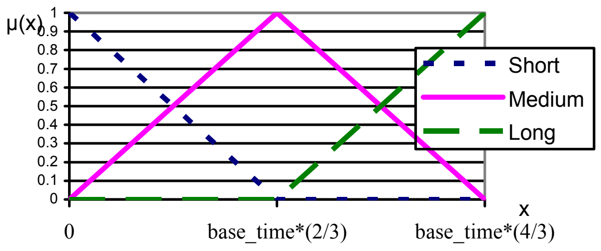

Figure 6. Membership functions for pellet time.

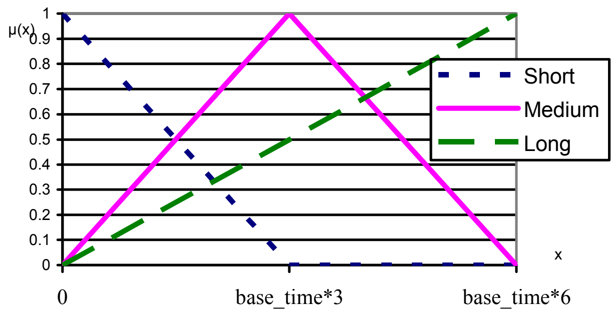

Figure 7. Membership functions for average lifetime.

4.3. Rate Variables

There is only one type of rate variables used in the proposed system, called the pellet rate. The pellet rate represents the ratio of the number of pellets eaten to the number of ticks that have passed since the game started. Since Pac-Man can only eat one pellet per tick, one would assume that the pellet rate would be a monotonic decreasing value. However, the system provides additional “bonus” pellets when a large number of pellets are eaten in a row, which makes it possible for the pellet rate to increase over time.

Similar to the other variables, there are three linguistic variable types defined for the pellet rate: Bad, medium, and good. However, the membership functions for each of the three types are defined a bit differently from the other variables in the system. The “good” pellet rate’s membership function is directly calculated as the number of pellets eaten divided by the number of ticks that have passed since the game started. To prevent the pellet rate from exceeding the range [0, 1], the maximum value is always clipped to

1. The "bad" pellet rate's membership function is the fuzzy complement of the good pellet rate, i. e., (1-good pellet rate). The "medium" pellet rate's membership function is essentially a triangle function with its peak at a pellet rate of 0.5. Figure 8 provides a graphical representation of the membership functions.

5. Defuzzification, Weighting, and Behavior Selection

In order to select a single behavior from the fuzzy calculations, the outputs from the fuzzy rules must be defuzzified. The defuzzified value for each behavior is calculated as follows. First, the individual fuzzy outputs of each fuzzy rule are calculated. Then the outputs of each set of fuzzy rules that relate to a specific behavior are combined via the disjunction operator (maximum). The result is a defuzzified output value for each behavior that is in the range [0, 1]. This method was chosen because it produces acceptable results and is less computationally complex than other defuzzification methods, such as centroid or average maximum.

Once the output value is determined for each behavior, it is multiplied by a weighting factor. The weighting factor is chosen by the programmer and allows certain behaviors to be given preference over others. For instance, if it is desired that a ghost act more aggressively, the "hunting" approach's output can be multiplied by a bigger weight. Alternatively, if the goal is to make the game easier, a ghost's tendency to move randomly can be increased by multiplying the defuzzified output of the "random" behavior by a larger weight.

For the fuzzy implementation of Pac-Man, three sets of weighting factors were developed. Each set is associated with a level of difficulty that the player can select. On the easy difficulty level, less weight is given to hunting Pac-Man. On the harder difficulty levels, the weights are configured so the ghosts are more likely to hunt Pac-Man than to choose other behaviors.

After the outputs have been defuzzified and weighted, it is necessary to choose a particular behavior. For proposed system, the maximum of the final outputs of the four behaviors is selected as the behavior a ghost will follow. For example, given the conditions shown in Table 1, the "hunting" approach would be chosen.

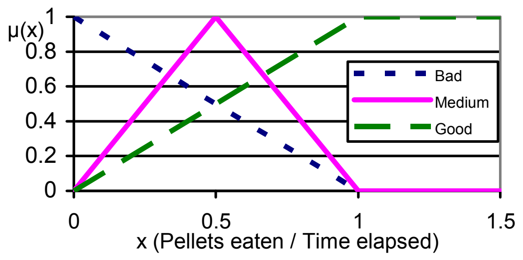

Figure 8. Membership functions for pellet consumption rate.

6. Results

The timing was excellent on a 500 MHz Pentium 3 PC that was used to run the system. Most game cycles occurred within +/-2 ms of the desired cycle time, and almost all occurred within +/-10 ms. On occasion, the game cycle time would be up to +/-60 ms from the desired time, but this is mostly unavoidable. Windows is not a hard-real time operating system, and the game has to share CPU cycles with numerous background tasks of equal priority. For this reason, the proposed system can be classified as a soft real-time rather than firm or hard real-time [12]. Even though the timings were occasionally overshot, there were no perceptible changes in game speed.

The execution time for one cycle of the control thread (which includes all the fuzzy calculations) was measured on a 500 MHz Pentium 3 PC. The average value was approximately 40 ms +/-5 ms. Given that the control threads runs once every 250 ms, 375 ms, or 500 ms depending on the difficulty level, it is possible to calculate the system's CPU utilization. This is shown below in Table 2.

Table 1. Behavior selection example.

Behavior	**Defuzzified Output**	**Weight**	**Final Output**
Hunting Approach	0.5	4	2
Defense Approach	1	1	1
Shy Ghost Approach	0.2	1	0.2
Random Approach	0.1	2	0.2

Table 2. CPU utilization on a 500 MHz Pentium 3 PC.

Difficulty Level	**CPU Utilization**
Easy	8 %
Medium	10.7 %
Hard	16 %

During the initial testing, some minor issues with the adaptation of the ghosts to the player's performance were found. Fortunately, it was trivial to revise some of the rules, membership functions, and weights to achieve the desired reactions. For example, in early testing it was found that the definition for poor player skill was far too strict. This caused the ghosts to hunt Pac-Man more than desired, even on the easier difficulty levels. By changing the definition of the fuzzy rule for skill_bad, it was possible to bring the ghosts to a level of difficulty that better matched the player's performance and selected difficulty. Tweaks such as these are common in game design and do not indicate a weakness in the fuzzy logic system.

The final implemented system was tested with 10 different players of various skill levels who provided their feedback on the experience. While this is a subjective method of testing, it is the only way to gauge the performance of a system that depends solely on player perception. In other words, the main goal of testing is to investigate if the players "feel" that the ghosts act and respond in an intelligent fashion.

The testing procedure consisted of having each person play several rounds of both the fuzzy implementation of Pac-Man and the original version of Pac-Man. For the fuzzy version of the game, all three difficulty levels were tested to evaluate how changing the weighting parameters of the fuzzy behavioral logic would impact the performance and perceived intelligence of the ghosts. At the end, each player was asked to rate each game in several categories on a scale of 1 to 10, where 1 is the lowest and 10 is highest. The average score of each game in each category is presented below in Table 3. Please note that because the original version of Pac-Man has only one difficulty level, only one score is given for difficulty (under "Medium").

Table 3. Player ratings of fuzzy vs. original Pac-Man.

Criterion	Fuzzy Pac-Man	Original Pac-Man
Difficulty on "Easy"	2.3	N/A
Difficulty on "Medium"	5.4	7.8
Difficulty on "Hard"	9.1	N/A
Ghost Predictability	8.2	6.4
Ghost Adaptability	9.0	2.8
Ghosts Feel Human?	6.3	3.7
Fun	7.1	6.5
Overall Impression	7.8	6.6

It can be seen from Table 3 that for most categories, the fuzzy version of Pac-Man game compares favorably to the original. In particular, notice that players rated the ghosts in the fuzzy version as more adaptable and more human-like. It is also interesting to note that players felt the ghosts in the fuzzy system were more predictable. This is due to the fact the ghosts were designed as intelligent, rational entities, meaning they tend to demonstrate logical behaviors that humans would expect.

As the game was being tested, the ghosts' actions and defuzzified output values for behavior were observed. In general, the ghosts seemed to respond appropriately to the current conditions and adapt to player trends. The players also responded positively to the game, stating that they felt the ghosts adapted and acted intelligently while maintaining a reasonable level of difficulty.

One example of the adaptation of the ghosts was seen when a player who had chosen the medium difficulty level was doing very well. The ghosts first responded by more actively hunting the player. Once the ghosts had succeeded in eliminating the player twice in rapid succession, the ghosts chose other behaviors that were not as aggressive towards the player. In another example on the highest difficulty level, the ghosts chose to guard areas where the pellet density was greatest. Once Pac-Man moved close enough to a ghost, the ghost began to hunt the player, demonstrating a believable intelligence.

6.1. Comparison with Non-Fuzzy Techniques

Game artificial intelligence can be designed without the use of fuzzy techniques. For example, most versions of Pac-Man, including the original, used crisp control mechanisms [1]. The deterministic logic that controlled the ghosts caused them to react consistently in predictable ways to player action. They did not learn from previous player performance and consequently adjust the level of difficulty. To create the illusion of intelligence, the original implementation of Pac-Man had special (crisp) rules for each of the four ghosts. For instance, certain ghosts would try to approach Pac-Man from different sides. Although this allowed the ghosts to behave more "realistically," it undoubtedly added to the size and complexity of the code.

Using fuzzy logic can rectify many of the deficiencies in the original Pac-Man while maintaining code simplicity. One such improvement that fuzzy logic provides is adaptability to player performance. In the proposed system, fuzzy linguistic variables are used to capture information about player trends. For example, the game records the pellet consumption rate and the average time between player deaths. Since this information can be used when determining the ghosts' behavior, the decisions that ghosts make are based on player actions over a period of time rather than at a specific instant. This means that even if an identical setup of ghosts and Pac-Man occurs in two different games, the ghosts will most likely react differently due to differences in past player performance.

Another benefit of a fuzzy control system is that it is easy to change or add rules that govern ghost behavior. The presented fuzzy framework allows rules to be altered or created and then integrated into the control logic with a change in only a single line of code. New fuzzy variables can also be created with relatively little work. It is even possible to define completely new behaviors for the ghosts, complete with their own sets of fuzzy rules. For example, a "huddle" behavior that would cause a ghost to move towards other ghosts could easily be created. Furthermore, this could be added without affecting the code and rules for the previously defined behaviors.

Fuzzy logic provides the proposed system with a simplicity that would be difficult to achieve with crisp logic. Any change to a crisp system could require the re-ordering or replacement of conditional statements or the re-scaling of variables. In the presented fuzzy system, rules can be changed independently, and all variables are always scaled to a common range (i. e., [0, 1]). An example of this simplicity can be seen in the behavior weighting. If it is desired to make the ghosts prefer a certain behavior to others, it is only necessary to modify the single weight value for that behavior. To change the behavior preferences in a crisp system, conditional statements would probably need to be changed, and entire sections of code might need to be moved.

7. Conclusion

Overall, the performance of the system is satisfying. In fact, the fuzzy control of the ghosts worked better than expected. The results indicated that the ghosts were successful in conveying a believable intelligence to the player. Their adaptation to the player's level of skill kept the game challenging without overwhelming the player. In addition, once the system was created, it was relatively simple to tweak the fuzzy rules and parameters to modify the ghosts' behaviors and levels of difficulty.

There are a number of ways the system could be improved in the future. For example, to enhance the artificial intelligence of the ghosts, new behaviors along with fuzzy rules and variables could be added. To simplify dynamic modification of the ghosts' preferences for certain behaviors, the weights could be read from a text file during execution. There are also more complex enhancements that could be made to the system. For instance, genetic algorithms or neural networks could be used to learn from the performance of each player and modify the rule base in response. In addition, an effort could be made to extract the core of the presented fuzzy system and make it generic enough to work with other types of games.

References

[1] Blair I., "Pac-Man," *Wikipedia*, http://en.wikipedia.org/wiki/Pac-Man, 2005.

[2] Bonarini A., "Evolutionary Learning, Reinforcement Learning, and Fuzzy Rules for Knowledge Acquisition in Agent-Based Systems," *in Proceedings of the IEEE*, vol. 89, no. 9, pp. 1334-1346, 2001.

[3] Combs W. and Andrews J., "Combinatorial Rule Explosion Eliminated by a Fuzzy Rule Configuration," *IEEE Transactions on Fuzzy Systems*, vol. 6, no. 1, pp. 1-11, 1998.

[4] Dick S. and Kandel A., and Combs W., "Comment on 'Combinatorial Rule Explosion Eliminated by a Fuzzy Rule Configuration' [and Reply]," *IEEE Transactions on Fuzzy Systems*, vol. 7, no. 4, pp. 475-478, 1999.

[5] Georgeff M., Pell B., Pollack M., Tambe M., and Wooldridge M., "The Belief-Desire-Intention Model of Agency," *in Proceedings of the 5th International Workshop on Intelligent Agents V*, Paris, France, pp. 1-10, 1998.

[6] Hart P., Nilsson N., and Raphael B., "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions on Systems, Science, and Cybernetics*, vol. 4, no. 2, pp. 100-107, 1968.

[7] Ishibuchi H., Sakamoto R., and Nakashima T., "Learning Fuzzy Rules from Iterative Execution of Games," *Fuzzy Sets and Systems*, vol. 135, no. 2, pp. 213-240, 2003.

[8] Jennings N., "On Agent-Based Software Engineering," *Artificial Intelligence*, vol. 117, no. 2, pp. 277-296, 2000.

[9] Johnson D. and Wiles J., "Computer Games with Intelligence," *in Proceedings of the IEEE International Conference on Fuzzy Systems*, Melbourne, Australia, vol. 3, pp. 1355-1358, 2001.

[10] Jones R. and Wray R., "Comparative Analysis of Frameworks for Knowledge-Intensive Intelligent Agents," *AAAI Fall Symposium on Achieving Human-Level Intelligence Through Integrated Systems and Research*, Arlington, Virginia, pp. 47-53, 2004.

[11] Langari R. and Yen J., *Fuzzy Logic: Intelligence, Control, and Information*, Prentice Hall, New Jersey, 1998.

[12] Laplante P., *Real-Time Systems Design and Analysis*, Wiley-IEEE Press, New Jersey, 2004.

[13] Li Y., Musilek P., and Wyard-Scott L., "Fuzzy Logic in Agent-Based Game Design," *in Proceedings of the 2004 Annual Meeting of the North American Fuzzy Information Processing Society*, Banff, Canada, vol. 2, pp. 734-739, 2004.

[14] Mathieson I., DanceS. , Padgham L., Gorman M., and Winkoff M., "An Open Meteorological Alerting System: Issues and Solutions," *in Proceedings of the 27th Australasian Computer Science Conference*, Dunedin, New Zealand, vol. 26, pp. 351-358, 2004.

[15] Mendel J., Liang Q., and Combs W., "Comments on 'Combinatorial Rule Explosion Eliminated by a Fuzzy Rule Configuration' [and Reply]," *IEEE Transactions on Fuzzy Systems*, vol. 7, no. 3, pp. 369-373, 1999.

[16] Microsoft Corporation, "MSDN Home Page", available at: http://www.msdn.microsoft.com/, July 19, 2004.

[17] Nishizaki I. and Sakawa M., "Fuzzy Cooperative Games Arising from Linear Production Programming Problems with Fuzzy Parameters," *Fuzzy Sets and Systems*, vol. 114, no. 1, pp. 11-21, 2000.

[18] Patel A., "Amit's Thoughts on Path-Finding and A-Star," available at: http://theory.stanford.edu/~amitp/GameProgramming/, 2004.

[19] Sanornoi N. and Sooraksa P., "Artificial Intelligence Based on Fuzzy Behavior for Game Programming," *in Proceedings of the 2004 ACM SIGCHI International Conference on Advances*

in Computer Entertainment Technology, Singapore, pp. 277-279, 2004.

[20] Takagi T. and Sugeno M., "Fuzzy Identification of Systems and Its Application to Modeling and Control," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 15, no. 1, pp. 116-132, 1985.

[21] Vijay V., Chandra S., and Bector C., "Matrix Games with Fuzzy Goals and Fuzzy Payoffs," *Omega*, vol. 33, no. 5, pp. 425-429, 2005.

[22] Wooldridge M. and Jennings N., "Intelligent Agents: Theory and Practice," *Knowledge Engineering Review*, vol. 10, no. 2, pp. 115-152, 1995.

[23] Yager R., "Targeted E-commerce Marketing Using Fuzzy Intelligent Agents," *IEEE Intelligent Systems*, vol. 15, no. 6, pp. 42-45, 2000.

Adnan Shaout obtained his BSc, MSc and PhD in computer engineering from Syracuse University, Syracuse, NY, in 1982, 1983, 1987, respectively. He is a full professor in Electrical and Computer Engineering Department at the University of Michigan, Dearborn. He has more than 20 years of experience in teaching and conducting research in the electrical and computer engineering fields at Syracuse University and the University of Michigan, Dearborn. He has published over 90 papers in topics related to electrical and computer engineering fields. His research interests include applications of fuzzy set theory, computer architecture, computer arithmetic's, real time systems and artificial intelligence.

Brady King received his BSc in computer engineering from Lawrence Technological University and MS in computer engineering from the University of Michigan–Dearborn. He is currently working towards his PhD in the same field at Wayne State University. His research interests include intelligent systems, embedded systems, and biomedical applications.

Luke Reisner is a lecturer at the University of Michigan, Dearborn. He received his BSc in computer engineering, BSc in electrical engineering, and his MSc in computer engineering from the University of Michigan, Dearborn. Currently, he is preparing to enter the PhD program in electrical and computer engineering at Wayne State University with a graduate research assistantship. His research interests include intelligent systems, digital signal processing, and embedded systems.