DATA ANALYSIS FOR GHOST AI CREATION IN COMMERCIAL FIGHTING GAMES

Worapoj Thunputtarakul
Vishnu Kotrajaras
Department of Computer Engineering
Chulalongkorn University Bangkok Thailand
worapoj.t@student.chula.ac.th, vishnu@cp.eng.chula.ac.th

KEYWORDS: Ghost AI, Fighting Game, Case base

ABSTRACT

In this paper we present a simple, rapid and efficient method for creating a ghost AI, an Artificial Intelligence that can imitate playing styles of players in fighting games. The ghost AI created can perform combination actions and make decision about any movement in a similar fashion to a player it is copying. We scan a player's battle data, and then create situation-action pair cases for the corresponding ghost AI to use in actual battles. A ghost AI can be created and run swiftly, using small amounts of memory, making it suitable for console games. Our method is general enough to be used in most 2D and 3D fighting games. We carried out our experiment on Street Fighter Zero 3, one of the most well crafted fighting games, using AI-TEM testbed engine.

1. INTRODUCTION

1.1 Ghost AI

An Artificial Intelligence (AI) that can copy a player's playing style had been used in many games for some time. In many racing games, a semi-transparent car, which mimics the player's controls from the previous race/lap, is usually available for the player to race against himself. That semi-transparent car is known as a ghost car. It is one of the well known ghost AI systems that players recognize. In fighting games there were various attempts at ghost AIs. Virtua Fighter 4 [8] allows players to train computer AIs to fight like them. Such ghosts can then be assigned to fight another player. However, feedback from players was not good at the time the game was released because it was hard to train their ghosts case by case. Also, at that time, means of ghost AI exchange and distribution were limited. Therefore the idea was not popular. But in recent years, ghost AI has again been used, in Tekken5: Dark resurrection [6]. This time many things have been changed. Players do not need to train their ghosts in a training mode, they just play the game normally and the system will mechanically create their ghosts. The created ghosts will be used as computer characters randomly when other players play the game. The ghost owners gain points if their ghosts defeat other players. This method makes fighting games more interesting because there will be many fighting styles for computer controlled opponents. Points gained by winning with their ghosts also motivate players to create more ghosts. Any player can fight with an expert without having to meet or make an online appointment. Furthermore, advancement in network systems allows people to easily exchange their ghosts and form ghost AI communities [7]. Despite the fact that the ghost AI system is being acknowledged as the definitive AI for fighting games, the method for ghost AI creation remains undisclosed. In this paper, we propose a method for ghost AI creation using data obtained from game memory. Our method can be used in most fighting games. It also requires very small amounts of memory and therefore is suitable for console games.

1.2 Street Fighter Zero3 (SFZ3)

Street Fighter Zero3 [5] on Nintendo GameboyAdvance (GBA) [3] is an almost perfectly ported version of the original Street Fighter Zero3 from arcade machines (CPS2 board system). This game is regarded as one of the best fighting games of all times [2].

In a fighting game, a player must select one character from many characters, and fight one by one with an opponent character (another player or computer AI). A character can perform normal action such as move, crouch, jump, guard, punch or kick. There are also special attacks, such as firing bullets or executing a powerful flying punch. These special actions can be performed when a player presses a correct sequence of commands at the right time. A player must choose to perform actions in various situations based on the status of his character and opponent character. Getting into action with SFZ3 requires only a few minutes of tutorial. Nevertheless, the game has many ways to play a single character. For that reason, we have chosen SFZ3 as our game for experimenting with the ghost AI.

1.3 Testbed Environment

For the reliability of experimental results, game researchers may want to test their AI on real commercial game environments (Graepel et al 2004). But such environments are scarcely available. Results obtained from a researcher created game may not be convincing enough to warrant an actual use of discovered techniques in genuine games. Some researchers used mod of a commercial game (Spronck et al 2004), or a clone game (Ponsen et al 2005).

Some developed test games on their own (Demasi and Adriano 2002, Kendall and Kristian 2004) or used a testbed (Bailey and Katchabaw 2005). But none of those methods fit our experimental goal. (Thunputtarakul and Kotrajaras 2006) proposed a system to test AI modules in real commercial games without using any source code. They implemented a testbed from VisualboyAdvance[9], a Nintendo GameboyAdvance emulator. The testbed was called AI-TEM [1]. An overview of AI-TEM is presented in figure 1 and its workflow diagram is presented in figure 2. By accessing the memory pool of the emulator, AI-TEM users are able to know states of the game at any particular moment. For fighting games, a state can consist of characters' positions, current animation frames, health points, etc. Users can insert their AI modules, in the form of C/C++ code or python [4] script, into the testbed to control the game characters by providing controller signals.

An AI-TEM user must first identify game states data that his AI module needs to know. After that, he must find the address of those game states data in the emulator memory address pool. This step can be done by memory searching and comparing techniques (Thunputtarakul and Kotrajaras 2006). It can be argued that for some games, this is a difficult task. However, it is still easier to do than implementing a complete commercial game from scratch. Our work uses AI-TEM as its testbed.

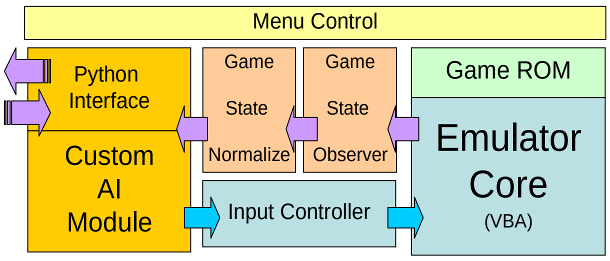
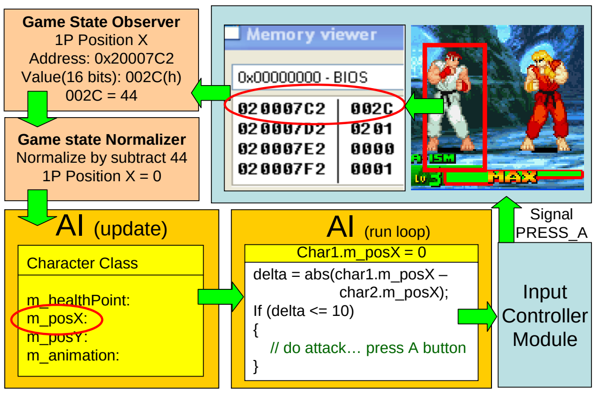


Figure 1: AI-TEM Testbed System Overview.
The Light Blue Modules are VBA Original Modules.



```
delta = abs(char1.m_posX –
            char2.m_posX);
If (delta <= 10)
{
   // do attack… press A button
}
```

Figure 2: Workflow Diagram of AI-TEM System in SFZ3.

2. OUR APPROACH FOR CREATING GHOST AI

The main concept of our ghost AI creation is case based AI construction. We extracted a player character's reaction in various situations from battle log data created while playing, then created situation-action pairs for the ghost of that character. Our experiment was made using SFZ3 training mode with character Ryu versus Ryu. AI-TEM was modified to suit our experiment. The ghost AI creation processes are displayed in figure 3. The following subsections describe each component in the process.

2.1 Obtaining Player Battle Log Data

First, while a player is playing, game states data need to be dumped from memory onto a battle log file. The data are used to identify each case in the case based AI system. The data consist of characters animation, characters positions in x and y axes, characters health points, characters bullet positions in x axes, damage that characters obtain in that frame, player character's facing direction and the corner status of characters. Recorded battle log data is in the following form:

```
Frame Data no: 00001
P1:Ani=002,X=120,Y=40,bullet=0,damage=0,HP=90
P2:Ani=002,X=240,Y=40,bullet=0,damage=0,HP=90
   :
Frame Data no: 00720
P1:Ani=016,X=150,Y=40,bullet=0,damage=0,HP=30
P2:Ani=030,X=560,Y=40,bullet=0,damage=5,HP=20
```

These criteria can change depending on game or user. Creating the ghost AI while the game is running without creating the battle log file is possible if complete information about the game mechanic is known (such as short or shared animation frame, that will be described in section 2.3). For SFZ3 on AI-TEM, we did not have such information. Therefore we had to use the log file.

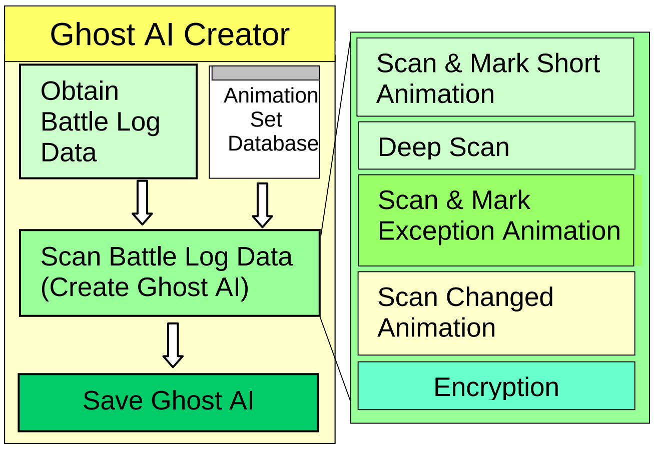


Figure 3: Overall Processes of Ghost AI Creation.

2.2 Animation Set Database

An animation set database is used for identifying whether a character animation frame belongs to an animation set. An example is illustrated in Figure 4. Ryu animation frame number 0 to 6 belong to animation set ID 0, which represents Ryu's standing animation, while frame number 707 to 713 belong to Ryu's medium punch action, set ID 150. Animation sets play an important role when processing the battle log file. Together with the battle log file, the animation sets are used to create situation-action pair cases. In our experiment, we manually defined this database. There are totally 912 frames for character Ryu. This seems daunting. However, it is relatively easy for a game company to do because any game development team usually has access to animation data.

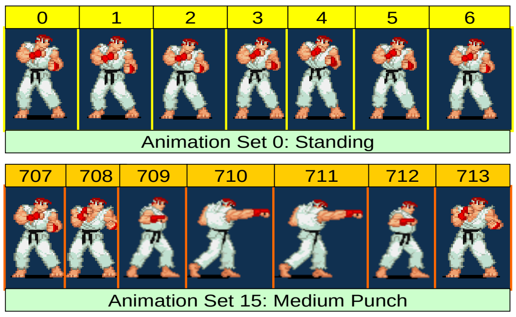


Figure 4: Example of Animation Set Database.

2.3 Scanning Battle Log Data

This process scans through every frame of a player's battle log data, trying to find which situation the player decided to begin his new animation set. For example, in situation *A* player1 is standing on the ground at position x=120 and player2 approaches player1 by jumping in the air at position x=150, both characters have full health bars and no bullets. Player1 decides to perform the special anti-air attack called Shoryuken punch. In short, the following situation-action pair will eventually be created:

```
if (Situation == A) do SHORYUKEN;
```

Now we look at this process in more detail. The process contains the following subtasks:

2.3.1 Finding short animation

Short animation means any animation that occurs for a very short period of time. It takes place mostly when a character is changing over from any standing animation loop to crouching animation loop. See an example animation time frame in figure 5. In figure 5, our character is standing then intends to do a crouching kick, but the crouching kick is not performed immediately. Before the crouching kick is carried out, a short period of moving forward and crouching animation is performed. This can happen due to the player not inputting the right command. For a crouching kick to be performed correctly without any prefix animation, the player needs to press down and kick at the same time on his control pad. In figure 5, the player presses down before kick and also unintentionally presses forward at the same time as down. Therefore extra animation is triggered. Nevertheless, the crouching kick is eventually performed and the prefix animation is so fast a human eye cannot see. We cannot avoid such minor mistakes made by players.

In our ghost AI model, detected animation frames tell us about a player's intention. Therefore, having the short animation taking place before the intended animation can misinform us. We must either identify a player's intention from the overall animation or get rid of the short animation before processing. In our experiment, we chose to do the latter.

All battle log data need to be scanned to find which animation set appears unusually brief, then that set is marked. Marked animation will not be considered when creating the AI. For a set of animation to be considered short, it depends on the set. In our experiment with SFZ3, short animation was no longer than 6 frames for most of the animation sets. The only exception was the crouching animation, of which short animation was no longer than 14 frames because changing from standing to crouching already took 8 frames.

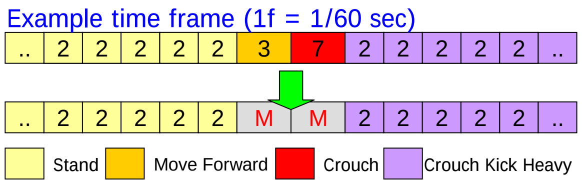


Figure 5: Short Animation Marking.

2.3.2 Deep scanning

Some animation frames are shared between many animation sets. In such case, scanning ahead becomes necessary in order to identify the correct animation set. For example, jump straight, jump forward and jump backward begin with the same animation frames at the beginning. With the first frame obtained, we can only conclude that the character is doing an anonymous jump. With further scanning, we then know which jump the player intends to do and can go back to change from an anonymous jump to a specific jump. This step can be omitted if the controller signal can be completely analyzed. But this is not always the case.

2.3.3 Exception Animation Sets

Some animation sets should be omitted from our case base because they do not take place under players' control. Obvious examples are various damage animation sets. They occur as the results of opponent attacks. This type of animation that appears in the battle log data will be marked here.

2.3.4 Scanning Changed Animation

This step is the core of our ghost AI creation. After matching all animation frames to their corresponding animation sets and marking useless animation, it is time to scan the battle log data once more to find the situation that causes the player character to change its animation. Such situation and the changed animation set that it causes will be paired to create a situation-action case.

An example is shown in figure 6, where a player executes a crouching heavy kick. In 7th-8th frame, our character changes its animation set from standing to moving forward. But moving forward lasts only 2 frames so it is a short animation. It is marked useless and the next animation to consider will be crouching. However, this crouching is also a short animation and therefore marked useless (a proper crouching must last 14 frames or more). As a result, the next animation (crouching heavy kick) will be taken into account. The crouching heavy kick does not fit useless animation category, so it is regarded as the changed animation set. Therefore the (situation at 7th frame, crouching heavy kick) is added to the case based AI.

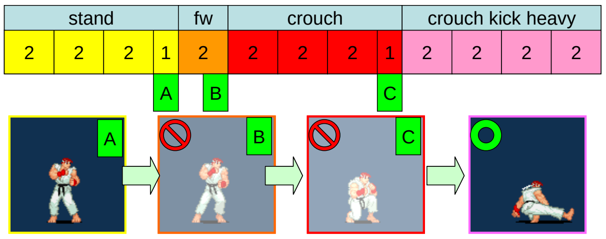


Figure 6: Scan Animation Change.

2.3.5 Situation Encryption

If the game needs to compare ten or more criteria (animation, position, bullet, etc.) to judge whether the current situation in the game is the same as any existing condition in our situation-action database, it will be a waste of processing power. Any game situation should be defined in simple form for easy comparison and discovery. We propose a method to encrypt a fighting game state situation into a 32-bit integer (capable of holding 4,294,967,296 values). The bits can be divided into small 1-8bits sections as shown in table 1.

Table 1: Detail of Situation Encryption.

Bit no.	nBits	nValues	Meanings
1-8	8	256	Character animation set ID.
9-12	4	16	Delta position in X axis.
13-14	2	4	Delta position in Y axis.
15-18	4	16	Enemy character state.
19	1	2	Character's bullet state.
20-22	3	8	Enemy's bullet state.
23-29	7	128	Enemy damage.
30	1	2	Character side, left right.
31	1	2	Is Player at corner.
32	1	2	Is enemy at corner.

• Bit 1 to 8 store the animation set ID of the action that the player character performs in that frame situation. The animation set value comes from the animation set database described in section 2.2. The number of animation sets for one character in a typical fighting game is between 60 to 100 sets. Our 8 bit can store up to 256 animation set ID, therefore it is capable of dealing with future games or games with extra animation sets.

• Bit 9 to 12 store the distance (in the x axis) between two characters. First, the distance is calculated and stored in variable *absDeltaX*. Then the distance table (shown in figure 7) is checked and the bits are set accordingly. A character position is at the middle of its sprite. For example, the distance between the two characters in figure 7 falls into range 8. Ranges do not separate equally. A close range will be more precise than a far range, because it plays a critical role in combination attacks. A table is specific for each character, but can easily be modified for other characters, including characters in different games.

• Bit 13-14 keep the y axis distance between characters. The meaning of distance in the y axis is different. Relative distance is not important. Players are only interested in whether a character is in the air. If a player character is in the air, the player mostly will perform an air attack. On the other hand, if the player character is on the ground and an opponent is in the air, the player may decide to perform an anti-air attack. Hence, the state of the y coordinate can be simplified. In our experiment, this state was given 4 possible values. They are 0: both characters are on the ground, 1: a player character on the ground and an opponent in the air, 2: the player character in the air and the opponent on the ground and 3: both characters are in the air.

Distance (dot unit)	Range ID and Range Meanings
10-45	[1] Close attack.
46-62	[2] Throw distance.
63-75	[3] Normal attack.
76-95	[4] 25% of screen.
96-140	[5] 30% of screen.
141-190	[6] 40% of screen.
191-240	[7] 50% of screen.
241-310	[8] 62% of screen.
311-380	[9] More than 75%.

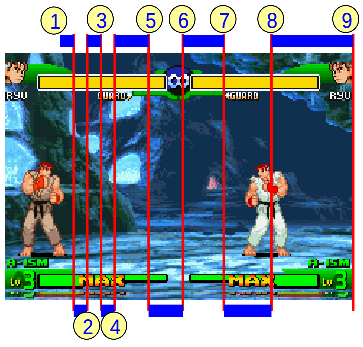


Figure 7: Range of Distance in the x Axis.

• Bit 15-18 store an opponent state. Keeping all of the opponent states is not necessary. There are only two things that an AI needs to know about its opponent. The first is whether the opponent is attacking. The second is whether the opponent is in a state that can be attacked. In our experiment, an opponent state had 6 possible values. They are, 0: any state that does not match other states. 1: the opponent is in damaged animation. The player can attack the opponent in this state. 2: the opponent is in a wake-up-from-knocked down state, which makes him invulnerable to any attack (most games use this mechanism to prevent unfair advantages). 3: the opponent is dizzy (some games do not have this), and cannot do anything. This state is a very nice opportunity for the player to perform his best attack combos. 4: the opponent is in an attacking motion. 5: the opponent is doing a block action.

Since our opponent state contained 4 bits. There were many possible values left unused. Other games can utilize such values as appropriate.

• Bit 19 tells us whether there is a player character's bullet on the screen. The bullet is mostly used for keeping distance form the opponent, or making an anti-air trap. Therefore, when combining with other bits, only one bit should be enough for this data.

• Bit 20-22. We used 3 bits to keep an opponent's bullet information. In our experiment, 6 possible distance values between a player character and a bullet were defined. Depending on the distance, a player can choose to perform various actions to react to an opponent's bullet, such as countering with his own bullet or jumping out of the way. Our bullet distance also included a 'safe' state, triggered when the opponent's bullet has already passed the player character.

• Bit 23 to 29 store the damage that an opponent character receives at that animation frame. We designed this to be able to distinguish between situations with the same state but come from different actions, such as distinguishing between two combination attacks.

Imagine two situations where a player character is crouch-medium-punching and his opponent is sustaining damage, the x and y distance of the two situations are the same, and there is no bullet on the screen (see the middle frame of figure 8). The first situation comes from a jump-kick, and will end with a tornado kick as a 3-hit combination attack. The other situation comes from a crouch-kick, and will end with a fire Hadoken bullet as a 3-hit combination attack.

If we only look at the second frame, you can see that the two sequences are exactly the same. Therefore, the tornado kick and the Hadoken may have the same probability of occurrence. This is wrong, since any whole sequence of combination attack should be remembered in its entirety in order for the ghost AI to perfectly reproduce the combination attack frequently used by the player. Any mixed sequence is considered unacceptable because it does not match the player's style of using combination attacks.

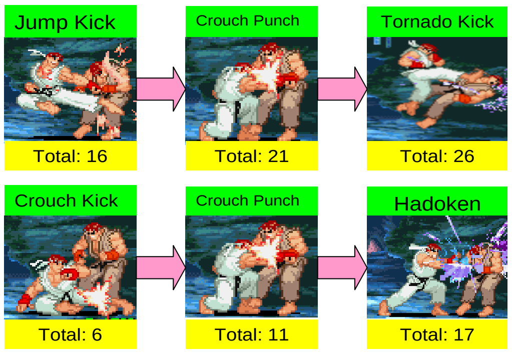


Figure 8: Different Combos Attack Situation.

To differentiate between combinations, we use damages that the opponent receives. In the first sequence of figure 8, the jump-kick causes 16 damage, then the crouch-punch causes 5 damage. Therefore the overall damage is 21. In the other sequence, the total damage that the opponent receives in the second frame is only 11. Now we can imitate the whole first sequence correctly. There is a chance that total damage value remains equal but that is very rare. Using damage value helps save memory space. Otherwise, our system will have to remember a sequence of animation sets, which is more memory consuming.

In many fighting games, an attack's damage value is not constant. For example, a heavy punch receives a damage bonus if used to counterattack. In order to use our method correctly, damage bonus and penalty need to be considered.

• Bit 30 tells us whether a player character is facing left or right. Players, especially novices, play the game differently depending on the direction their characters are facing. Some can only execute a special move when facing right. Some cannot perform a combination attack when facing an unfamiliar side.

• Bit 31 tells us whether a player character is at a corner or not. When a character is at corner, it is easier to be damaged by combination attacks because there is no space for evasion. Players usually try to escape from a corner.

• Bit 32 tells us whether an opponent character is at a corner.

As an example, the situation in figure 7 is saved as follows: Bit 1-8 are set to *0* (player character is standing). Bit 9-12 become *0111* (range 8 but begin with 0). Bit 13-14 are set to *0* (both on the ground). Bit 15-18 become *0* (standing normal). Bit 19-22 are set to *0* (no bullet). Bit 23-29 are set to *0* (no damage occurs). Bit 30 is *1* (player facing left). Bit 31 is set to *0* (player is not at any corner). Bit 32 becomes *1* (opponent is at a corner). When we bring all 32bits together the result will be 2,684,356,352 (in decimal). Details can be changed to match other games or other platforms. We will discuss this topic again in section 5.1.

2.4 Creating Ghost AI File

When the scanning process discovers that animation set change takes place, the situation in the frame before that discovered frame is encrypted into 32-bit data (integer) by the process in section 2.3.5. Its corresponding case base can now be created by combining the situation ID (32-bit situation encryption result) with its response action list. An example of our case base is shown below.

```
SituationID: 0000000000
TotalRatio: 03 TotalNextAni: 02
    NextAni: Punch-Light-Close Ratio 2
    NextAni: Kick-Heavy-Close  Ratio 1
:
SituationID: 2684356352
TotalRatio: 01 TotalNextAni: 01
    NextAni: Hadouken          Ratio 1
```

Each case will have `situationID` for representing each game situation. `TotalRatio` is the number of incidents the player encounters that situation. `TotalNextAni` is the number of different animation sets that the player performs when facing that situation. It is followed by the list of those animation sets and the number of times the player performs each animation set. The ratio of each animation set and the total number of sets will be used in response selection while the ghost AI is actually running.

From above example cases, the player encountered situation 0 three times and decided to do a light-punch twice and a heavy kick once. These cases should be kept in a data structure that is convenient and fast to insert and find because we need to know whether the situation is a new situation that player never encounters (so we can add new data from scratch), or an old situation that updates the response action list. In our experiment we chose *map* of standard template library (STL), which is a balanced binary search tree, to store the cases. The tree was written into our ghost AI file. Using file allows for future modifications of the knowledge base.

3. USING GHOST AI

To run the ghost AI, first, the game needs to load any required database such as the animation set database. Then it needs to load the ghost AI case base into some data structure that allows quick finding and matching. A new case is never inserted while running the ghost AI.

From the data in section 2.4, the game first loads all cases into the *map*. When the `situationID` 0 takes place, the case that has `situationID` 0 in the *map* is searched. It will be found and returned. That case has a total ratio of 3 and has two next animations (light-punch with ratio 2 and heavy-kick with ratio 1). The game then randomly selects one of these actions corresponding to the ratio value and sends a command to perform that action.

A command is a controller press function. For example, if the ghost AI decides to fire Hadoken bullet, the controller press function will create a signal buffer like:

```
PRESS_DOWN                6 frames,
PRESS_DOWN|PRESS_FORWARD 6 frames,
PRESS_FORWARD             6 frames,
PRESS_PUNCH               6 frames
```

That will make the controller signal buffer not empty for the next 24 frames. While the buffer is not empty, our ghost AI will not encrypt game state situation or find any case base from the *map*, but will immediately return the next signal in the buffer to the emulator.

4. VERIFYING METHOD AND RESULTS

The best way to evaluate a ghost AI's similarity to its creator should be: letting its creator verify with his own eyes. But sometimes, people can make incorrect judgments, forgetting even their own playing styles. Therefore we designed a measurable method for evaluating the ghost AI.

4.1 The Experiment

We appointed thirty two SFZ3 players and let them play the game for approximately 2 to 10 minutes. We recorded their game events in VMV file format (recording the beginning game state and controller sequence) and created their ghost AI. After that, we let the player semi-play the game again two more times, while their ghost AI was playing and while their own playing movie was playing. The term semi-play means players see their ghosts or their own movies playing while pressing the controller, imagining that they are controlling their characters in that situation. We wanted to compare the controller signals of the ghosts with the players' signals. We also wanted to compare the players against their video.

Controller signals should not be compared frame-by-frame, because only 1 frame delay (1/60 second) will cause the rest of the matching process to fail.

Therefore the controller signals need to be normalized before any comparison can be done. In our approach, we normalized the signals by splitting the signals into parts. Each part contained approximately 5 to 15 signals. After that, we combined all the same signals that appear continuous into one signal (when a player presses one button normally, it takes approximately 6-8 frame, so it gives out 6-8 continuous signals). For example, if the signals are as follows:

```
Raw ghost signal:
16,16,16,32,32,32,64,64,64,64,128,128,256,256,256
Raw player signal:
16,16,16,16,16,16,32,32,32,32,64,64,128,128,128
```

After normalized they will be like these.

```
Normalized ghost signal: 16,32,64,128,256
Normalized player signal:16,32,64,128,0
```

It can be seen from the example that if we compare raw signals directly the result will be 3 of 15 signals match. The matching result is not correct because identical commands that are pressed for slightly different amount of time will be regarded as being different. However, if we compare the two signals after our normalization, the match is 4 out of 5.

We had two methods for slicing controller signals. In the first method, we sliced every 15 frames. We had tried several values and this value gave the best result. Too small values made the normalization meaningless, while too large values put more than one signals in the same frame, making the result unreliable. In the second method, we performed the slicing every time the signal of the ghost AI or the player movie changed values, based on the assumption that matching signals should occur in the same frame time period as its counterpart. With the second method, we always had one signal per slicing window. We also gave score if there were some similarity between controller signals. For example, if the ghost AI was pressing down-forward and the player was pressing forward only, we gave similarity score of 0.5 (50%) to the ghost AI.

4.2 Result

The result of our experiment is illustrated in figure 9 and table 2. *Player_Player%* is the similarity (in percentage) between each player's own movie and his actual control when re-playing the situation in the movie. *Ghost AI_Player%* compares each ghost AI with its corresponding player's re-play. *Delta*% is the difference between the two comparisons. Table 2 displays the overall statistical summary. *Delta A* and *Delta B* indicate delta percentage points between the result of [player's own movie vs. player] and [ghost AI vs. player]. *Score* is the score that the players evaluate their ghosts' similarity to their fighting styles based on their feelings.

Both signal slicing methods gave similar results. But the second method gave less matching percentage points. This is likely because the number of signals after the normalization was less than in the first method. With many long signals in play, such as idle signals, the first method scored better because it did not compress long signals into one signal.

For the first method, the average similarity between ghosts and the players is 26.33%. This may seem small. But if we look at the comparison between the players and their own movies, the similarity is only 34.96%. The ghosts' performances were therefore very close to players' performances (75.31% close). Some ghosts even scored better than their corresponding players.

An average satisfactory score given by players is 72.2%, which is good. The players thought that the ghosts sometimes performed more attacks and fewer defenses than their creators. Some players could not distinguish between their ghosts and their own movies while semi-playing. (We did not tell the players which engine was really controlling the characters).

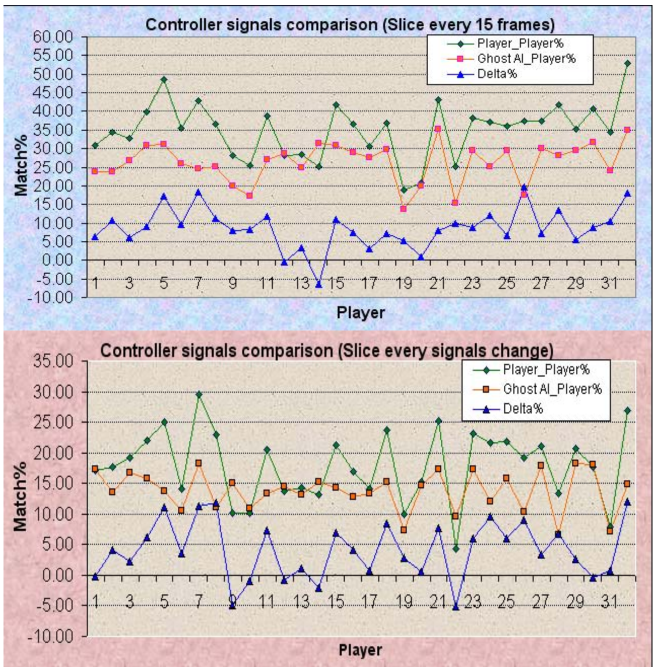


Figure 9: Players vs. Movies and Players vs. Ghosts.

Table 2: Summary Result of Experiment. A: Slice Every 15 Frames, B: Slice Every Time When Signal Change.

Summary	Player_ Player A	Ghost AI_ Player A	Delta A	Player_ Player B	Ghost AI_ Player B	Delta B	Score
Min	18.81	13.6	-6.45	4.37	6.54	-5.18	44
Max	52.84	35.18	19.82	29.51	18.27	12.03	90
Average	34.96	26.33	8.62	17.93	13.81	4.12	72.2

5. DISCUSSION

5.1 About Situation Encryption

The situation encryption method is very important. Our proposed integer representation is only intended as a guide, not an exact solution. Details can be changed to suit other games or characters. For example, a character with long arms will have greater throwing and attacking ranges.

Our situation information did not consider heath points and super move points because these bars are always full in training mode, the game mode that we used. In actual play, these parameters can influence players' decision. For example, some players always use their super move when their opponents' health bar is nearly exhausted. This is because a special move deals damage even when guarded. Without these two parameters, our guideline encryption criteria still gave satisfactory results.

5.2 Pitfalls and solutions

Creating a ghost AI by scanning changes in animation ignores the fact that "not doing anything" is also a player's style. For example, when a character is damaged while in the air, its player can choose not to do anything and let the character fall to the ground normally (animation set does not change), or perform an air recovery (animation set changes). Our method only regards changes in animation as actions intended by players. Therefore "not doing anything" is not recorded and counted as one of possible player actions. This means even though the player may choose to do an air recovery only one out of ten times, his ghost AI will always do the air recovery when facing the same situation. The solution to this problem involves identifying situations that allow for both reactive and "do nothing" actions, then adding "do nothing" as the intended action appropriately if the system does not detect any animation change when running the situation.

Other pitfalls are caused by players changing fighting styles. Ghosts do not unlearn things easily. If a player changes his playing styles, his ghost AI will use mixed-up styles learned from all his fights. Therefore it will not act like him. To use more than one playing styles, a player will have to create another ghost AI.

5.3 Performance of Creating and Using Ghost AI

To create a ghost AI, we need to store a player battle record into a file and scan it. This process does not use heavy CPU power or large amount of memory. The amount of data recorded depends on the length of the recording session. In our experiment, we recorded about 4KB of data per minute. When a ghost AI is running, if used with a suitable data structure such as a balanced binary search tree, searching any case is guaranteed to use *O(log n)* amount of time (when n is the number of cases). A ghost AI with one thousand cases should find a result in the tenth search. Each case based data uses approximately 40 bytes of memory. Therefore, a thousand-case ghost requires only 40KB of memory. In short, creating and running our ghost AI does not slow down the game or consume much memory at all.

5.4 Other Findings

We also tried to verify our results by using methods other than in section 4. These methods used human judgment.

Some did not measure the similarity between human and his ghost AI counterpart directly, so we did not include these methods in our main experimental result in section 4.

Ghost AI vs. its creator: We invited players to see many matches of AI against AI. The AIs were the mixture of ghost AIs, player recorded movies and original in-game AIs. We asked them if they could tell which one was a ghost AI and which one was a player recorded movie that was the source of that ghost AI. Most of them could answer correctly about the movies and their ghost counterparts. From each movie-ghost pair, only some people could tell which one was a ghost. Some even believed that a ghost was a human player.

Statistical performance of Ghost AI: We counted the number of each movement that each ghost performed then compared with its creator's recorded movie. It was found that each movement was executed with a similar rate for both. This verified our implementation.

We also counted the hit rate and the damage rate of each ghost against its creator. Each ghost achieved similar results compared to its creator.

The effect of the amount of time used in training: The average number of cases generated was 180 in the first two minutes. It gradually increased to 350 cases in ten minutes (this statistic varied depending on players, some player could create 450 cases in the first two minutes). The longer the training period, the harder new cases will come up. Therefore we believe that if a player and his opponent do not change their style, his training period does not have to be long to obtain an effective ghost.

We also selected some player that has a solid playing style to play for a longer period (30 minutes). The resulting ghost was not noticeably different from the resulting ghost created by a 10-minute play.

6. CONCLUSION AND FUTURE WORK

We propose a method and concept for creating ghost AI without having to know game source code. We used AI-TEM, an emulator based testbed to provide a commercial game testing environment. Our concept for ghost AI creation is general for all fighting games. Using SFZ3, which is a very well respected commercial game. Its basic systems are use in almost real fighting games so our findings are guaranteed to be applicable to real games.

Our method produces good results. Ghost AIs display their creators' playing styles even when the training time is short. The two-minute average training time we used is equal to a match time in an average fighting game.

For future experiment we are interested in exploring techniques for ghost AI in team based fighting games, where characters can cooperate. Another interesting future work is developing AI that can adapt and counter an opponent's play style.

REFERENCES

Bailey, C. and M. J. Katchabaw. 2005. An Experimental Testbed to Enable Auto-Dynamic Difficulty in Modern Video Games. *Proceedings of the 2005 GameOn North America Conference.* Montreal, Canada.

Demasi Pedro and Adriano J. de O. Cruz. 2002. Online CoEvolution for Action Games. *GAME-ON 2002 3rd International Conference on Intelligent Games and Simulation*, SCS Europe Bvba, pp. 113–120.

Graepel Thore, Ralf Herbrich, Julian Gold. 2004. Learning to fight. *International Conference on Computer Games: Artificial Intelligence, Design and Education*

Kendall Graham, Kristian Spoerer. 2004. Scripting the Game of Lemmings with a Genetic Algorithm. *Proceedings of the 2004 Congress on Evolutionary Computation,* IEEE Press, Piscataway, NJ, pp. 117-124

Ponsen Marc J.V., Hector Munoz-Avila, Pieter Spronck, and David W. Aha. 2005. Automatically Acquiring Domain Knowledge For Adaptive Game AI Using Evolutionary Learning. *Proceedings The Twentieth National Conference on Artificial Intelligence.*

Spronck Pieter, Ida Sprinkhuizen-Juyper, Eric Postma. 2004. Online Adaptation Of Game Opponent AI With Dynamic Scripting. *International Journal of Intelligent Games and Simulation,* Vol. 3, No. 1, University of Wolverhampton and EUROSIS, pp. 45–53.

Thunputtarakul Worapoj and Kotrajaras Vishnu. 2006. AI-TEM: Testing Artificial Intelligence in Commercial Game using Emulator. *8th CGAMES International Conference on Computer Games: AI, Animation, Mobile, Educational & Serious Games.* Louisville Kentucky, USA.

[1] AI-TEM: Artificial Intelligence Testbed in Emulator (2007). http://www.cp.eng.chula.ac.th/~g48wth/aitem.htm
[2] Gamespot review: Street Fighter Alpha3 (2002). http://www.gamespot.com/gba/action/streetfighteralpha3/index.html
[3] Nintendo GameboyAdvance (2007). http://www.gameboy.com
[4] Python (2007). http://www.python.org
[5] Street Fighter Zero3, Capcom game (2006). http://www3.capcom.co.jp
[6] Tekken 5: Dark Resurrection, Namco game (2005). http://www.tekken-official.jp/tk5dr/index.html
[7] Tekken Zaibatsu , Tekken ghost AI community (2007). http://www.tekkenzaibatsu.com/forums/ghostlist.php
[8] Virtua Fighter 4, Sega game (2002). http://www.virtua-fighter-4.com
[9] VisualboyAdvance, GameboyAdvance Emulator (2005). http://vba.ngemu.com
[10] Street Fighter Alpha Anthology: Bradygames official strategy guide (2006)
[11] All About Street Fighter Zero3: All About series vol.21, Studio BENT STUFF (1998)