Department of Computer Science and Engineering

The Chinese University of Hong Kong

2007 – 2008 Final Year Project Semester 2 Report

Legendary of 18 Weapons – Motion Capture Analysis for Wii Remote

Group: LYU0702

Supervised by

Prof. Michael R. Lyu

By

Ng Kwok Ho

Tsoi Chi Hung

Prepared by:

Ng Kwok Ho (05669353)

ABSTRACT

Motion capture refers to the technique of recording movements in a digital manner for different purpose in entertainment, sports or medical treatments. At present, most researches regarding motion capture as a use of accelerometers. Those accelerometers are normally attached to certain parts of the body; say the limbs or the hands. However, there are not much researches relating to traditional Chinese martial Art.

Wii Remote also has built-in accelerometers. A player can connect Wii Remote through Bluetooth to the computer, and capture acceleration values through Wii Remote API, i.e. capture player’s motion. It is interesting to investigate on motion capturing and classification with Wii Remote.

In this report, we are going to describe the motivation, background information, what we have done in this semester, as well as problems and limitations we encountered so far. Our objective is to first build a motion classifier to classify motions done with the use of Wii Remote, so that we can further make use of our classifier, either using in games, or for educational purpose.

In the following sections, we would first give some background information as well as the idea of this final year project. Following is different classification algorithms and representations of motion data we have studied. Then we will talk about the implementation details of our motion classifier. After that we will show how the motion classifier can be used in practice on other applications. Finally we will mention problems and limitations we encountered, as well as potential further developments for the project.

TABLE OF CONTENTS

CHAPTER *0*
ACKNOWLEDGEMENT

We would like to express our sincere thanks to our project supervisor, Professor Michael R. Lyu. He has provided us computers and equipments for the project, and given us the many useful advices throughout the two semesters. He has also given us some opinion on our presentation skills so that we can make our presentation better.

We would also like to appreciate the guidance from Mr. Edward Yau, the VIEW Laboratory manager, on our demo. We thank for his innovative ideas and suggestions on our project, too.

Last but not the least, we would like to thank Mr. Un Tsz Lung, the VIEW Laboratory staff member, for giving us suggestions on different classification algorithms, as well as testing and giving opinions on our demo games.

CHAPTER *1* INTRODUCTION

In this chapter we are going to describe some background information about current game industry, the players, the revolution brought by Wii, the Chinese martial arts, well as our objectives and motivation. The development environment of our project will be discussed here as well.

This chapter comprises the following sections:

- 1.1 Introduction to the game industry
- 1.2 What do game players want?
- 1.3 The revolution of Wii
- 1.4 What is lacking in Wii?
- 1.5 What is Legendary of 18 Weapons?
- 1.6 About Chinese Martial Arts?
- 1.7 Our Objective & Motivation
- 1.8 Development Environment

1.1 INTRODUCTION TO THE GAME INDUSTRY

The game industry is mainly composed of two parts – video games, and computer games. Since the birth of game industry in 1972, the industry evolved rapidly, and many new features were introduced with the advancement of hardware and technology from time to time.

However, since the introduction of new consoles using 32-bit technology in 1994, the development of the game industry started to stabilize and tend to be mature. There are several reasons for this:

1. With the invention of 32-bit technology, games could make use of 3D graphics rather than 2D graphics previously. Many of the games since then are some remake of the previous ones, or are originated from them. New ideas are not common. This lacks creativity.

2. The video game market is dominated by proprietary standards from big manufacturers, such as Sony PlayStation® and Microsoft Xbox 360®. In the meantime, to prevent obsolescence of old consoles, the owners of the standards are not willing to release new upgrades frequently, or introduce new features which requires hardware support. This slows down the development.

3. In the computer game market, undoubtedly it is dominated by a single hardware standard – the personal computer [1]. Both players and the manufacturers are limited by the input device – keyboard. This limitation also exists in video game market – the game pad limits everything. This narrows the range of games and limits creativity.

1.2 WHAT DO GAME PLAYERS WANT?

There are 2 big aspects that the player wants – variety of inputs, as well as reality. Players want to have different types of inputs, rather than the traditional keyboards / gamepads to control, to play the game in a more realistic way, or have a more realistic feeling, so that they can enjoy the game more.

One common example is the steering wheels for playing vehicle games. Players can play the control the vehicle in a realistic way, say by rotating the steering wheel to control the car movement, or applying brakes with their legs to stop the car.

Fig 1.1 A typical steering wheel specially designed for games

Konami's Dance Revolution is another example making use of different inputs. Players can dance on a specially designed game pad and this is considered as the actual movement inside the game. Undoubtedly players can enjoy this very well.

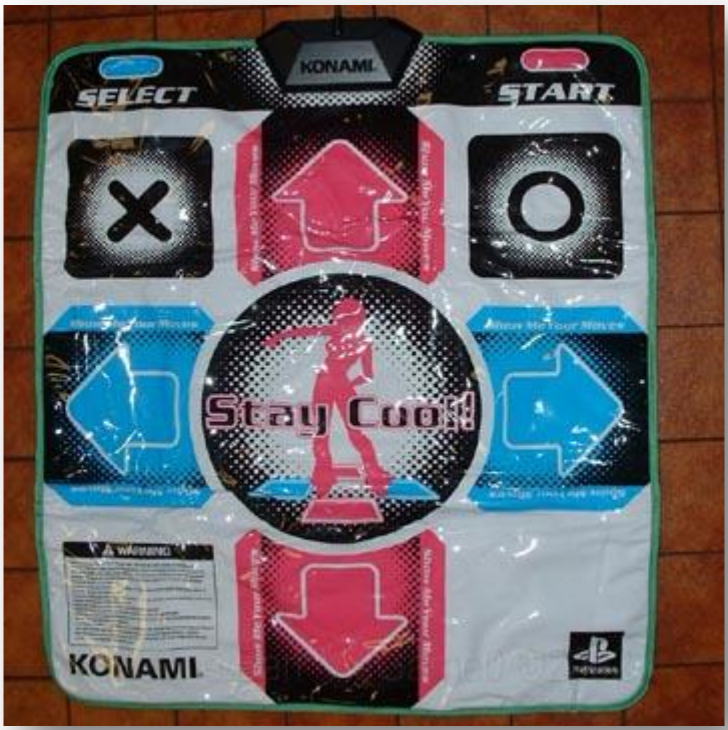


Fig. 1.2 Dance Pad specially designed for Konami's Dance Revolution

1.3 THE REVOLUTION OF WII

Nintendo Wii®, first available in November 19 2006, is considered as a revolution in Video Games.

Nintendo noticed that classical game controllers haven't been changed for more than two decades, and thought this limited creativity and ranges of games. "There are examples of controllers that were made for specific games such as Konami's Dance Revolution. And for a long time, we thought that changing the interface would broaden game design and loosen creative constraints on programmers." said Shigeru Miyamoto from Wii development team [2].

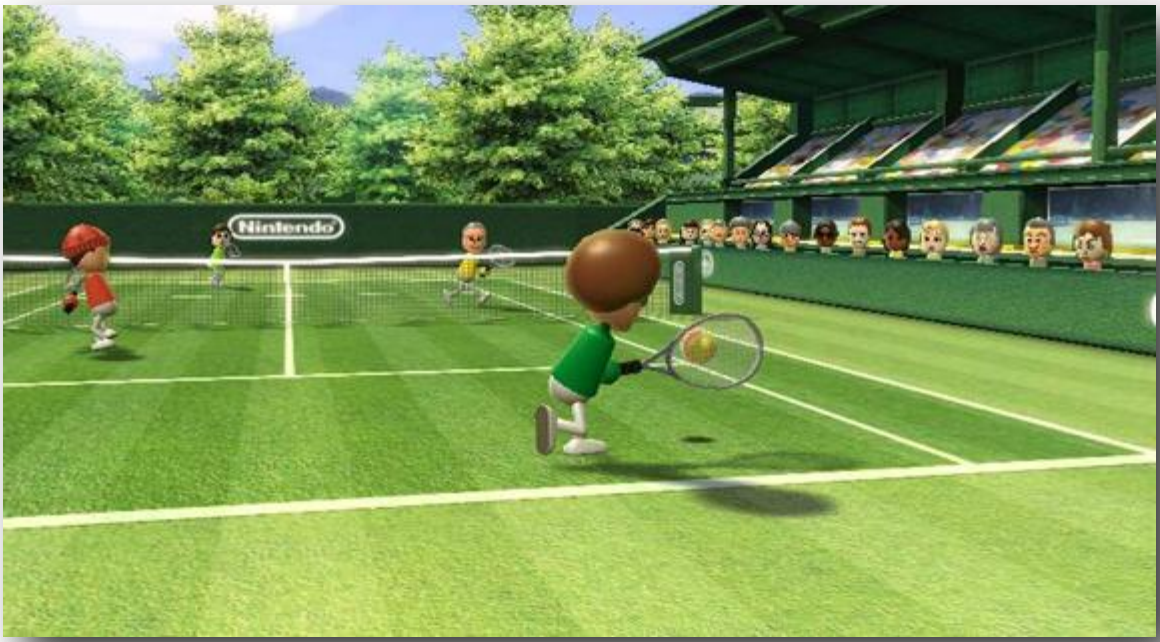


Fig. 1.3 Screen shot of Wii Sports – the tennis

So Nintendo started developing a new console with codename "revolution", Wii is the final name for this console. One of the most distinctive features is the design of the game controller. Unlike controllers of other consoles, Wii controller is a remote – *Wii Remote*. This remote has several important features:

1. Motion sensors are installed inside the remote to sense movements from players, which is treated as the input to the Wii machine. Instead of holding the gamepads and press buttons to control, Wii Remote allows user to control by just waving the remote, which is more realistic.
2. The remote is designed in such a way that motion detection is more accurate and intuitive [4].
3. The remote communicates with the console through Bluetooth. This allows maximum flexibility to the player when motion is performed.

With the Wii Remote, players can play the game in a more realistic way; say you can play tennis or golf games in Wii by performing actual tennis or golf-like motions. This was never been possible before the invention of Wii.

1.4 WHAT IS LACKING IN WII?

Although Wii Remote has built-in motion sensors, still in most of the games, they are used to detect movements only. Only a few games make use of Wii Remote to perform motion classification. This does not make full use of Wii Remote. In the mean time, even for those games with motion classification, the performances are not very good still.

In other words, Wii is still lacking games with accurate motion classification.

Fig. 1.4 Victorious Boxers: Revolution, one of the Wii game which requires motion recognition

On the other hand, the majority of Wii games are manufactured by game companies from Japan and Western countries, in which the ideas are originated from their home countries. The game market is still lacking games originated from our country – the mainland China; there are rooms to develop games in which the idea comes from our traditional cultures.

Furthermore, being inspired by Nintendogs, a Nintendo DS game featuring dog raising, player should actually be able to define their own motions so that they can have fun in the game using their own style of motions.

Fig 1.5 In Nintendogs, you'll be able to name your pooch using the game's voice-recognition software, repeating it a few times until your pooch "learns" his or her name.

1.5 WHAT IS IN LEGENDARY OF 18 WEAPONS?

In Chinese martial art, one of the well-known components is the "Eighteen Arms". In fact "Eighteen Arms" is the list of the eighteen major weapons used in traditional martial art. There were rumors saying that these eighteen weapons first appeared in a book eight hundred years before. Although there are inconsistencies on the correct list of eighteen weapons, still all lists contains most of the following weapons:

- Dao 刀, Qiang 槍, Jian 劍, Halberd 戟
- Axe 斧, Battle axe 鉞, Hook sword 鉤, Fork 叉
- Trident-halberd 镗, Cudge 棍, Lance 槊, Short cudgel 棒
- Chain whip 鞭, Mace 鐧, Hammer 錘, Talon 抓
- Stick 拐, Meteor Hammer 流星錘

1.6 About Chinese Martial Arts

Along with the legendary of 18 weapons, Chinese martial arts are important for using the 18 weapons. Chinese martial arts training comprises training in three major areas: [24]

1. Basics (基本功). As its name tells, basics are the fundamental things. Basics training generally includes simple moves which are performed repeatedly over an interval of time. Without them, beginner cannot proceed to more advanced training stages.

2. Form (套路). It refers to a series of basic moves, which are chained together and practiced as one linear set of movements. It is normally designed to preserve the lineage of a particular style. It may be performed by a single person called “solo forms”; while some of them have to be performed by two or more people, they are named as “sparring forms”.

3. Application. Application training is needed when martial arts techniques are put in use. Before the actual fighting training, learner will have to learn how the motions are applied in different situation. In the fighting training it usually involves two learners, acting as attacker and defender. When the attacker performed one motion, the defender will have to react to attacker's motion by performing motions that are suitable for defending himself.

   Application training also involves training that applies different motions from different forms into fighting. Unlike motions in a form which is sequential, after a certain motion has been performed, several motions from different forms can be followed. Learner will have to utilize the technique learnt from different forms.

1.7 OUR MOTIVATION & OBJECTIVE

Although Wii is a revolutionary console in the game industry, still people can only enjoy such new methods by buying a console. However, it is relatively costly to buy a console separately (around HKD$2000), while buying a remote just costs you one-tenth of the price.

Fig 1.6 The Wii Console

In the meantime, every family is equipped with a computer in major modern societies such as Hong Kong. The computer can serve for many purposes including computer games and many people love playing computer games, too. Still at this moment there are no games for users to perform some motions as control in the game.

Fig 1.7 A typical Bluetooth® USB adapter

With the popularity of Bluetooth, one can purchase a Bluetooth USB adapter with around HKD$60 in different computer shops in Hong Kong. The adapter can pair with the Wii Remote, and connects it with the computer. That means user can use the Wii Remote to serve as an input to the computer. This is rather important as user can perform Wii-like motions in front of the computer, and the movements can be detected.

We believe that, if the data captured from the Wii Remote are analyzed with accurate motion classification algorithms, together with specially-designed games, a player can perform exactly the same thing to personal computers as what the player may do in front of the Wii console. The performance can be even better due to the high computational power of personal computers when compared to Wii console, as well as the presence of accurate motion classification algorithms.

Moreover, we also believe that this is a much bigger market since the number of personal computers over all the families must be more than the number of Wii consoles. Undoubtedly this will be a big market to all game developers and a new area of game playing, and we believe many interesting findings or creations can be done from this new area.

This is our motivation.

With this motivation, we decide to work on this project. Our objective is to first construct a motion classifier, using data from motion sensors in Wii Remote as the input. With this it is possible for players to perform complicated motions and be recognized accurately in a short time.

When the motion classifier is ready, the next step is to build an interface for players to perform motions directly in front of the computer, and be recognized correctly. The interface can be in the form of a game, or educational software.

With this, players can enjoy playing with such revolutionary approaches without the presence of Wii console. Players can use any modern computers to play rather than a designated machine; this increases flexibility and players do not need to pay extra costs.

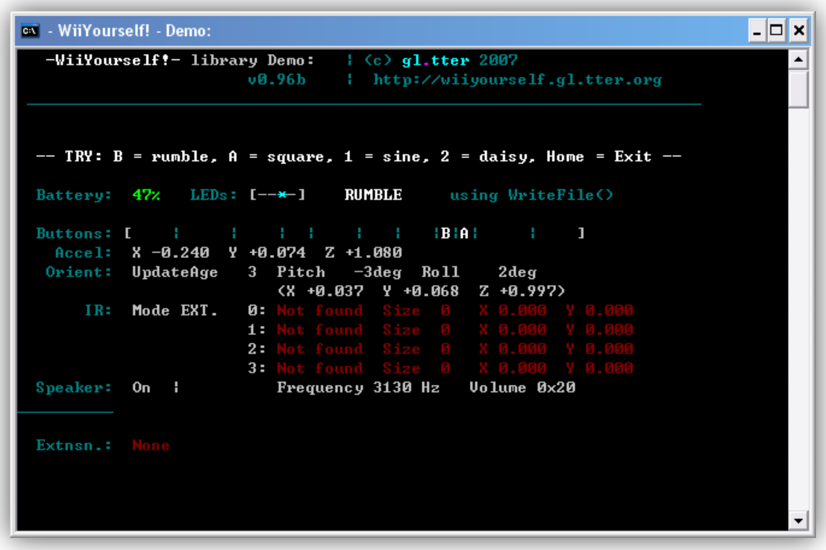
1.8 DEVELOPMENT ENVIRONMENT

Our system is developed under Microsoft Windows XP Professional Edition. Windows XP is chosen as the development system since it is currently the most popular operating system in personal computers.

To retrieve sensor inputs from Wii Remote, we have to use a Wii Remote API. We have selected *WiiYourself!* [5] as the API for our project. We select *WiiYourself!* because the source is freely available over the internet, such that we can integrate it with our system easily. At the same time it can provide some other useful data to help us in computation.

WII YOURSELF!
NATIVE C++ WIIMOTE LIBRARY BY gl.tter

In our project, we adopt C++ as our programming language, and Microsoft Visual Studio as our programming environment. We choose C++ because most Wii Remote API available, including *WiiYourself!*, are also written in C++. To facilitate system development, we decide to adopt C++ as our programming language. Microsoft Visual Studio allows a hierarchical structure among files and so we can organize our coding well.



```
- WiiYourself! - Demo:

 -WiiYourself!- library Demo:   ¦ (c) gl.tter 2007
                    v0.96b      ¦  http://wiiyourself.gl.tter.org


-- TRY: B = rumble, A = square, 1 = sine, 2 = daisy, Home = Exit --

Battery:  47%   LEDs: [--*-]    RUMBLE     using WriteFile()

Buttons: [    ¦     ¦    ¦  ¦    ¦   ¦   ¦B¦A¦     ¦    ]
  Accel:  X -0.240  Y +0.074  Z +1.080
 Orient:  UpdateAge   3  Pitch   -3deg  Roll    2deg
                         (X +0.037  Y +0.068  Z +0.997)
     IR:  Mode EXT.   0: Not found  Size  0   X 0.000  Y 0.000
                      1: Not found  Size  0   X 0.000  Y 0.000
                      2: Not found  Size  0   X 0.000  Y 0.000
                      3: Not found  Size  0   X 0.000  Y 0.000
Speaker:  On  ¦          Frequency 3130 Hz   Volume 0x20


Extnsn.:  None
```

Fig 1.8 A screenshot of *WiiYourself!* API Demo

CHAPTER *2* – WII REMOTE OVERVIEW

In this chapter we are going to introduce the main apparatus for this project – the Wii Remote. As claimed by Nintendo, the wireless Wii Remote is a multi-functional device, and it is only limited by game designer's imagination. Apart from its main functionalities – pointing and motion-detection, Wii remote can also serve for other purposes include controller feedback and speaker.

In this chapter, we will briefly introduce Wii Remote's design as well as the functionality. We will see a general picture on the use of Wii Remote on Wii Games

This chapter comprises the following sections:

- 2.1 Design
- 2.2 Functionality
  - 2.2.1 Motion Detection
  - 2.2.2 Pointing
  - 2.2.3 Expansion Port – the Nunchuk
- 2.3 Wii Remote on Wii Games

2.1 DESIGN

The Wii Remote, as its name specified, is a remote-like controller instead of the traditional gamepad controllers from previous gaming consoles. Unlike the traditional controllers, which require both hands to hold the controller, Wii Remote has a one-handed design such that players can handle them with either one of two hands. Its dimension – 148 mm long, 36.2 mm wide, and 30.8 mm thick, allows most of the players to hold the Wii Remote comfortably, such that they can enjoy using it more.

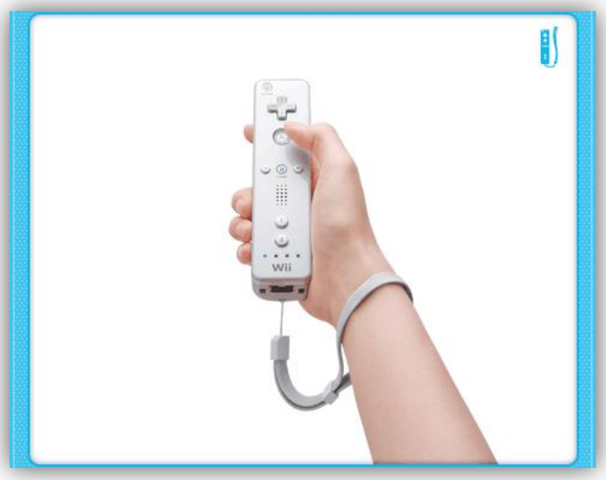


Fig 2.1 Nintendo's official picture of holding a Wii Remote

This design has 2 advantages [4] to suit for its purpose:

1. Motion sensing is more intuitive. In Wii, instead of pressing buttons to control everything inside the game, the majority of controls are done based on player's motion. To let players control everything easily, motion sensing has to be done in an easy, straight forward manner. The remote-like design of Wii Remote facilitates motion sensing a lot. It's not hard to image how difficult it is when players are asked to hold the traditional controllers to perform some motions.

2. Fit perfectly for pointing. Some types of games, for example, shooting games, require players to focus on a particular target to perform some actions, say shooting. As Wii Remote serves as the controller, undoubtedly player has to point to the correct position to specify the target. With Wii Remote, such pointing action is just as natural as using a TV remote and point to TV's infra-red sensor for operation.

2.2 FUNCTIONALITY

In this section we will focus on 2 main functionalities of Wii Remote – motion sensing and pointing.

2.2.1 MOTION DETECTION [12]

The motion sensing is done by accelerometers inside the Wii Remote. In fact accelerometers are some cantilevers hewn from silicon and teetering between two electrodes. A 1-volt voltage is constantly applied so that cantilever's beam will vibrate.

When the Wii Remote accelerates – either by rotation or movement of Wii Remote, the beam's tip traces an ellipse. The value of acceleration can then be measured based on the eccentricity of the ellipse.

Each accelerometer is responsible for tracking accelerations in 1 direction only. So to trace accelerations in the space, 3 accelerometers are placed at right angles with each other. The acceleration values are then reported to the console through Bluetooth.

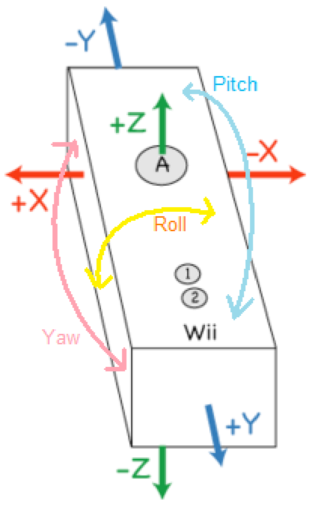
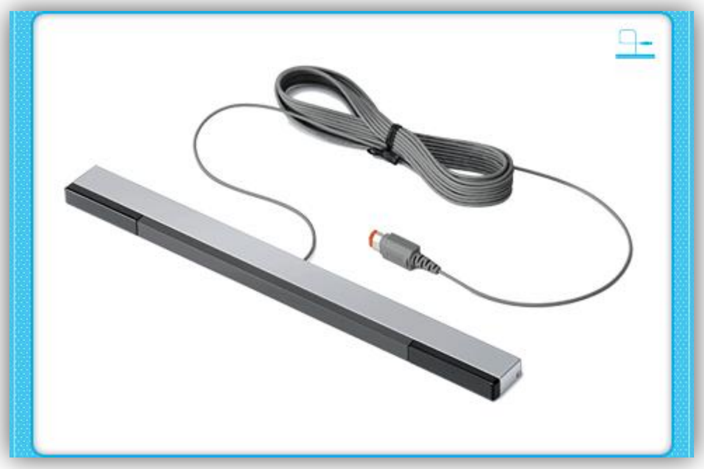


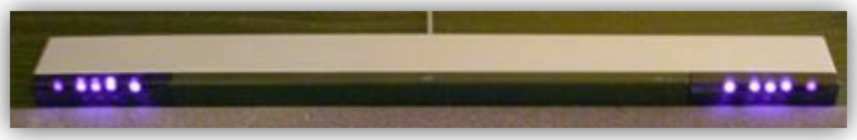
Fig 2.2 The 6 degrees of freedom of Wii Remote

2.2.2 POINTING [4]

Another functionality of Wii Remote is the pointing. Pointing in Wii Remote is achieved by the Wii Remote and Wii sensor bar.



(a)



(b)

Fig 2.3 The Wii infra-red sensor bar

< (a) – the overview; (b) – when the infra-red LEDs are highlighting>

The sensor bar consists of 10 infra-red LEDs, 5 on each end. In each side, the LED farthest away from the centre points slightly away from the centre; while the LED closest to the centre points slightly towards the centre; the remaining LEDs just group together and point straight forward.

Generally the sensor bar is placed at the centre, either above or below the television. The Wii Remote includes a 1024x768 monochrome camera, with an IR-pass filter in front of it. The camera includes a built-in processor capable of tracking up to 4 moving objects. [12] It is used to sense the infra-red emitted from each LED. It senses the light as 2 bright dots and the distance between the Wii Remote and the sensor bar is computed using triangulation. Thus the pointing position can then be known.

2.2.3 EXPANSION PORT – THE NUNCHUK

There is an expansion port for players to connect other components with the Wii Remote. One of the components that can be connected is the Nunchuk.

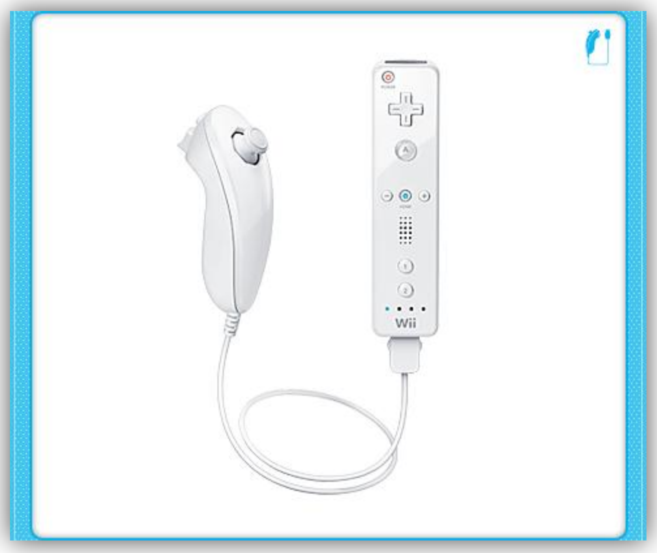


Fig 2.4 How nunchuk (Left) is connected to a Wii Remote

Generally Nunchuk contains the same motion-sensing technology used in the Wii Remote, i.e. it provides accelerometers for motion detection in 3 axes. On the other hand, it also includes an analog stick to assist characters movement, and two buttons for quick access.

Nunchuk works together with the main controller in many games, and this enables the ambidextrous nature of the Wii Remote – making use of both hands in playing games. Such nature is not found in other game controllers.

2.3 WII REMOTE ON WII GAMES

Although Wii Remote is a revolutionary design, still the game developers can design games making use of this remote. Some of the uses are as follows:

1. Adventure and First-person shooting Games. For these types of games, a player can treat the Wii Remote as a weapon, points and attacks an enemy in the game. Two outstanding examples are *Resident Evil 4 : Wii Edition* and *Medal of Honor Heroes 2*.

Fig 2.5 Screenshot of *Resident Evil 4* (Left) and *Medal of Honor Heroes 2* (Right)

2. Role-playing Games. One of the well known games in Hong Kong is *Cooking mama*.

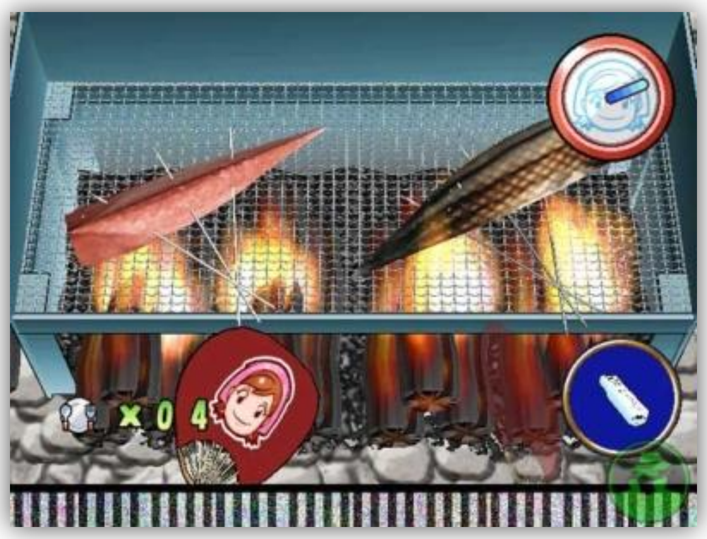


Fig 2.6 Screenshot of *Cooking mama*

The Wii Remote can serve for simulation purpose at this case. A player can imagine himself as holding the fishes on the hand; and rotate the Wii Remote if he wants to cook the other side of the fish.

3. Racing and driving games. Although it does not seem possible to drive cars using Wii Remote, still a number of manufacturers made it possible. One example is the *Mario Dart*.

   It is easy to turn left or right – just rotate the Wii Remote in clockwise or anti-clockwise direction. Some Wii lovers simply made a wheel themselves, with Wii Remote installed at the middle, so that they can enjoy more.

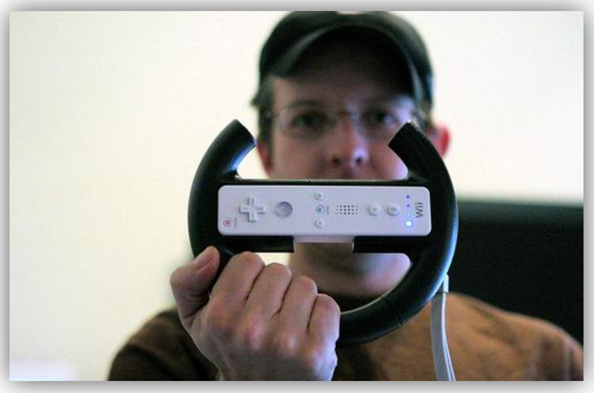
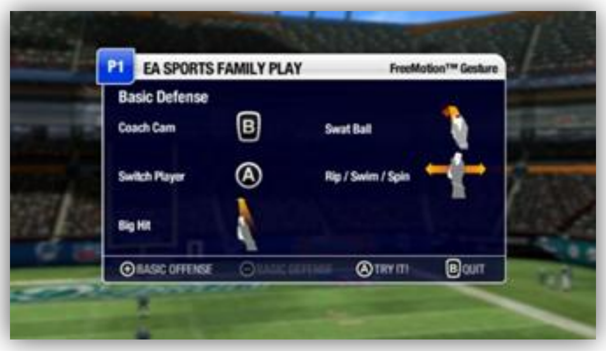


Fig. 2.7 (a) Fig. 2.7 (b)

(a) A screenshot of Mario Dart; (b) The man-made steering wheel

4. Sports game. In some sports, the Wii Remote can be served as the racket that players swing with their arm. Some examples include the tennis and golf. While in some other sports, the Wii Remote can be treated as a ball that the player is holding. Some examples are bowling and America football.



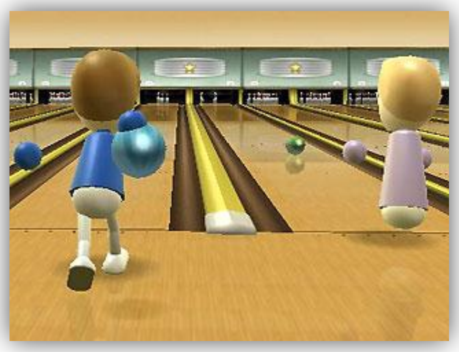


Fig 2.8 (a) Fig 2.8 (b)

(a) A screenshot for *Madden NFL 08* – the America football game; it teaches players on using Wii Remote to defense.
(b) A screenshot from *Wii Sports* – Bowling. Players can play just as if they are holding the ball

5. Fighting games. Traditionally players can fight with the computer by pressing buttons in particular sequences to have special skills be performed. However, with Wii Remote, players can simply move the Wii Remote to achieve the same goal. One outstanding example is *Mortal Kombat: Armageddon*.

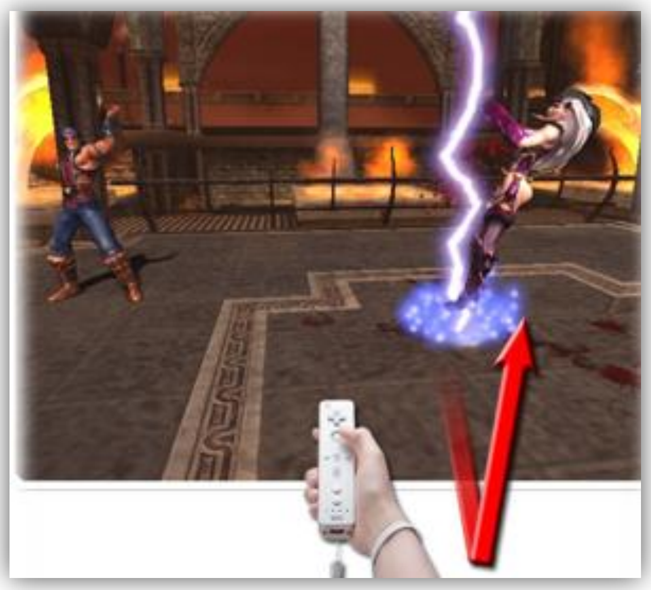


Fig 2.9 Screenshot of *Mortal Kombat: Armageddon*

Fig 2.9 shows a screenshot of the game. To have the game character performing the special skills shown at the screenshot, players should press the button on Wii Remote and move their hands as indicated by the arrow to have the skills be performed.

Apart from the Wii Remote, players can also use Nunchuk together with the Wii Remote in such fighting games. One of such example is *Wii Sports* – Boxing.

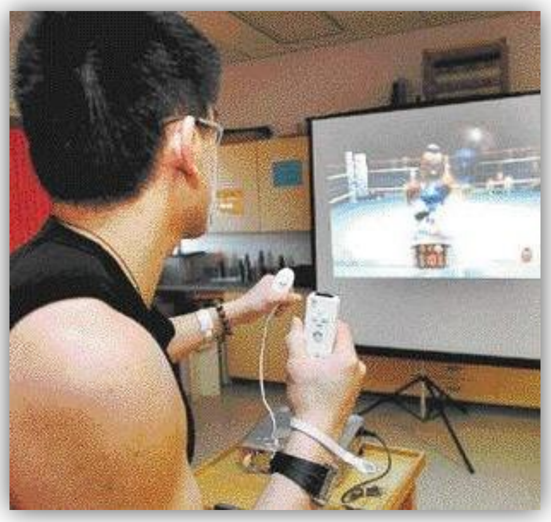


Fig 2.10 Photo showing a man is playing *Wii Sports* – Boxing

As shown in Fig 2.10, the boy treats the opponents standing as if it is in front of him. He may hold Wii Remote together with the Nunchuk, and performs fisting motions with both of the hands. Then his motion is reflected inside the game and the opponent inside the game will then be hurt.

CHAPTER 3 – MOTION CLASSIFICATION

In this chapter we are going to describe our studies on different classification methods. One of them is specially designed for Wii Remote on Wii, while others are well known methods used in different kinds of projects or area. We would also discuss how motion data are represented in the computer. In the meantime, we will have some investigation on the inter-relationships between motion classification, motion design and user dependency / independency.

This chapter comprises the following sections:

3.1 EXISTING CLASSIFICATION METHODS FOR WII - AILIVE LIVEMOVE

LiveMove, and recently has its new version renamed as LiveMove *Pro*, is invented by AiLive in 2006. LiveMove is generally powered by its context learning technology.

With LiveMove, developer can build motion classifiers by showing examples instead of coding – one can hold a Wii Remote, performs a number of motions repeatedly, and then a classifier is built. At runtime, the constructed motion classifiers can determine the correct motion that the player is performing.

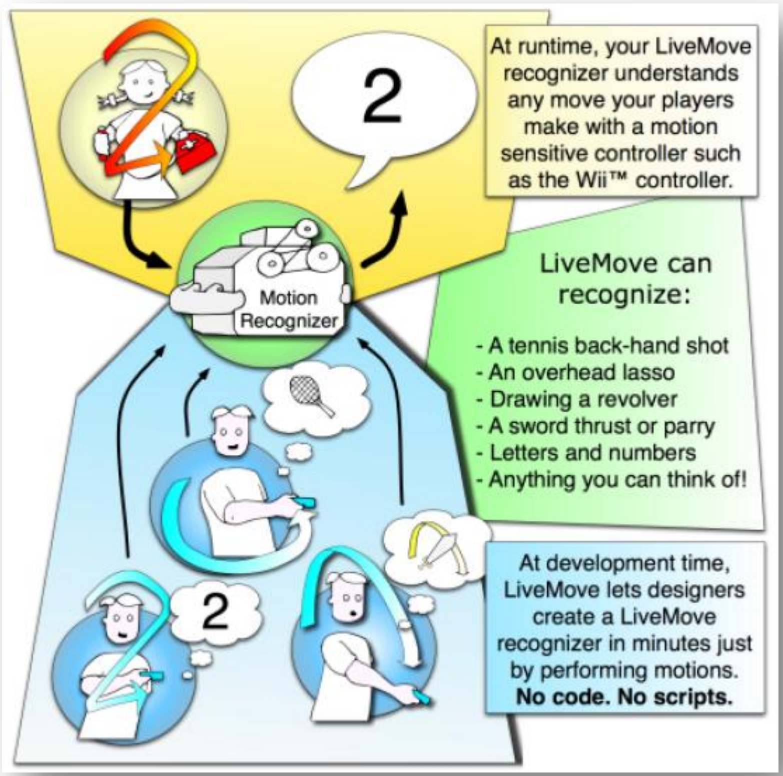

2
At runtime, your LiveMove recognizer understands any move your players make with a motion sensitive controller such as the Wii™ controller.
Motion Recognizer
LiveMove can recognize:
- A tennis back-hand shot
- An overhead lasso
- Drawing a revolver
- A sword thrust or parry
- Letters and numbers
- Anything you can think of!
2
At development time, LiveMove lets designers create a LiveMove recognizer in minutes just by performing motions. No code. No scripts.

Fig. 3.1 A diagram summarizes the work flow of LiveMove [6]

There are several features in LiveMove *Pro* [6] :

1. Zero-lag recognition. In its newest *Pro* version, motion classification can be done within a few frames after the motion ended. This minimizes the delay between the game responses and player's actual motion.

2. Player Synchronization. In the *Pro* version, several tools are provided to help n synchronizing player's physical motion with the motion on the screen, so that the motion on the screen actually reflects what the player does – in terms of speed, degree of rotation, etc.

Although LiveMove seems to be excellent, but still there are some constraints and limitations with LiveMove:

1. LiveMove is not open source. From its LiveMove Pro Director's Cut Manual[6], it is specified that license for one PC costs about USD$2500. It also has a runtime license to be used in the game, and it costs USD$10000 per game as well. That means a normal human, except those working in wealthy game developers, does not have any chance to use this software.

2. LiveMove is designated for Wii only. From AiLive's site on ordering LiveMove *Pro*, one ordering requirement is the possession of the official Wii NDEV development hardware. This implies that LiveMove can only be used for developing games on Wii platform but not the others.

3. Animations and moves must be pre-defined by the developer / author. Obviously this limits creativity in the player side.
4. LiveMove is not specifically designed for Chinese martial arts.

Since there are a number of inconveniences, that's why we would like to construct a new classifier ourselves.

3.2 REPRESENTATION OF DATA

Data representation is an important and yet fundamental step towards any programming. In the following, we will describe the format of data obtained from Wii Remote, and followed by some possible representations.

3.2.1 DATA FORMAT FROM WII REMOTE

With the use of *WiiYourself!*, the Wii Remote API, accelerations from 3 different and perpendicular axes can be obtained. The values are real number. If the acceleration is 0, this means that the Wii Remote is at rest along a particular axis. If the value is greater than 0, this means that the Wii Remote is moving towards the positive side of that axis, and vice versa. The direction of positive / negative axes can be found in Fig. 2.2. Note that the gravity is also measured by the Wii Remote. Therefore even the Wii Remote is resting on a table, the value of the axis parallel to the gravity is not zero.

By obtaining the three accelerations continuously in a regular interval of time, we will have a set of time series data, which can be represented as three curves in 2 dimensional spaces with the time axis.

Generally speaking, the accelerations obtained from Wii Remote can have variations and the curve representing them may not be smooth. This may increase the difficulty in comparing the similarities between the inputs and the sample motion in the classifier.

As the data can be represented as curves in 2 dimensional spaces, in order to have a better result, representation such as Piecewise Linear Approximation, Discrete Fourier Transform and Polynomial Functions can be used for representing the motion. Moreover, since human motions are continuous but not discrete, continuous representation will be best to fill up the gap between two consecutive intervals where no data present.

Apart from accelerations, Wii Remote can also obtain Infra-red information when the Infra-red sensor bar is available. The information obtained is useful for determining the position of the Wii Remote in which it is pointing.

3.2.2 PIECEWISE LINEAR APPROXIMATION (PLA) [13]

Given a curve with n data points, there exist algorithms to approximate the curve into segments of discontinuous straight lines. Such representation is called Piecewise Linear Approximation (PLA) of the curve.

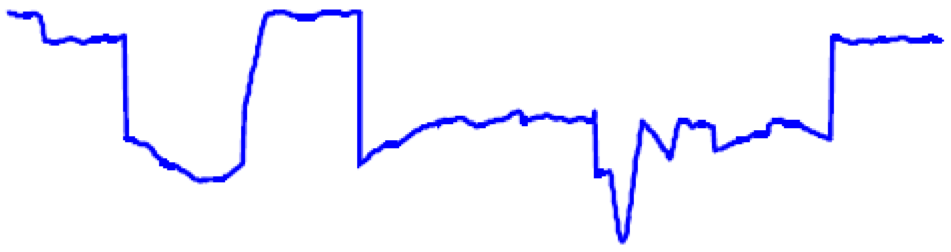


Fig 3.2 (a) A sample data curve

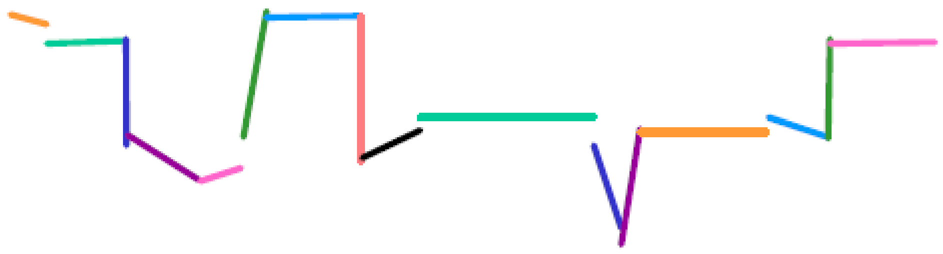


Fig 3.2 (b) The PLA of (a)

We investigated on PLA because there are several advantages:

1. According to [13], PLA supports fast and exact similarity search.
2. The change point is sharp so that similarity search can be more accurate in some situation.

Theoretically PLA can be obtained using Dynamic Programming, but it is too slow to be obtained, and so there are a number of alternatives available on the internet, which includes:

1. Top-down. Sequences of data-points are recursively partitioned until the partition meets some stopping conditions.
2. Bottom-up. The algorithm begins from the finest possible approximation; segments are formed by repeated merging until meeting stopping criteria.
3. Sliding Window. Segment keeps on growing until error bound is exceeded.

We have chosen SWAB – Sliding Windows And Bottom-up, to implement a mixture of sliding windows and bottom-up approaches. This is proven to be more accurate and efficient than some existing algorithms in [13].

The algorithm begins by first initializing the buffer size such that 5 to 6 segments can be created. Data points are taken into the buffer continuously, and bottom-up is applied in the buffer. In an iteration, the leftmost segment is “reported” and will not be included in the buffer anymore. The process is repeated until the whole curve is approximated.

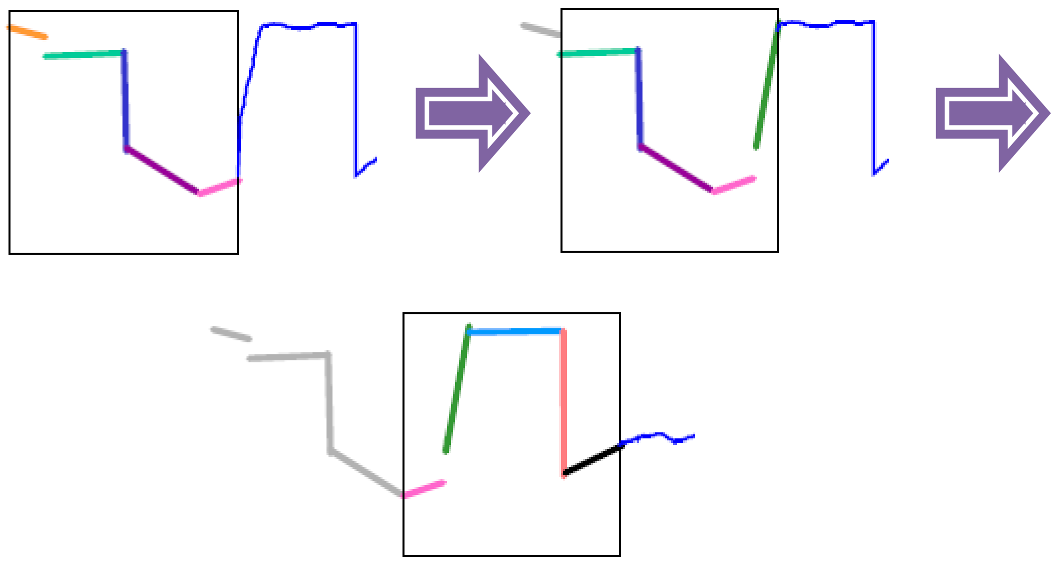


Fig 3.3 Flow chart showing how PLA is applied on the curve of Fig 3.2(a)

3.2.3 DISCRETE FOURIER TRANSFORM (DFT) [15]

Mathematically, *Discrete Fourier Transform* (DFT) is one of the specific forms of Fourier analysis. With DFT, it transforms a periodic sequence $x_0, x_1, \ldots, x_{N-1}$ (non-zero values have a finite duration *N*) into another discrete sequence $X_0, X_1, \ldots, X_{N-1}$, the frequency domain representation of the input sequence, according to the following formula:

$$x_n = \frac{1}{N}\sum_{k=0}^{N-1} X_k e^{\frac{2\pi i}{N}kn} \qquad 0 \le n \le N-1$$

$$X_k = \sum_{k=0}^{N-1} x_n e^{-\frac{2\pi i}{N}kn} \qquad 0 \le k \le N-1$$

where *e* is the base of natural logarithm, *i* is the imaginary unit of complex number.

The equation can be interpreted in the following manner. At point *k*, the sequence value x_k is a linear combination of values of N sinusoids:

$$e^0, e^1, \ldots, e^{\left(\frac{2\pi}{N}\right)k(N-1)}$$

The coefficients of the sinusoids are $X_0, X_1, \ldots, X_{N-1}$ respectively, and their frequencies are $\frac{k}{N}$ cycles per sample. By writing the equations in this form, sinusoids are expressed in the form of complex exponentials making use of Euler's formula.

If DFT is computed directly based on the formulas above, it would take $O(N^2)$ arithmetic operations, which is too slow for classification. So there are a number of algorithms used to compute DFT in only *O(N log N)* operations, and they are generally named as *Fast Fourier Transform* (FFT).

In our investigation, we have chosen FFTW, which stands for the somewhat whimsical title of "Fastest Fourier Transform in the West", to compute DFT. FFTW is a subroutine library for computing DFT in one or more directions. In FFTW, it mainly uses *Cooley-Tukey FFT algorithm* for computation.

Cooley-Tukey FFT algorithm [16] speeds up the computation by re-expressing the DFT of an arbitrary composite size *N* into smaller DFTs of sizes N_1 and N_2, where $N = N_1N_2$, recursively and making use of radix-2 decimation-in-time (Radix-2 DIT).

In Radix-2 DIT, it begins by dividing a DFT of size N into two interleaved DFTs of size N/2 with each recursive stage. The next step is to first compute the Fourier Transforms of the even-indexed numbers $e_m = x_{2m}(x_0, x_2, \dots, x_{N-2})$, and followed by the computation of odd-indexed numbers $o_m = x_{2m+1}(x_1, x_3, \dots, x_{N-1})$. The 2 results are then combined to produce Fourier Transforms of the whole sequence. The decomposition can be summarized as follows:

$$\begin{aligned}
X_k &= \sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-\frac{2\pi i}{N}(2m)k} + \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-\frac{2\pi i}{N}(2m+1)k} \\
&= \sum_{m=0}^{M-1} e_m e^{-\frac{2\pi i}{M}mk} + e^{-\frac{2\pi i}{N}k} \sum_{m=0}^{M-1} o_m e^{-\frac{2\pi i}{M}mk} \\
&= \begin{cases} E_k + e^{-\frac{2\pi i}{N}k} O_k & \text{if } k < M \\ E_{k-M} - e^{-\frac{2\pi i}{N}(k-M)} O_{k-M} & \text{if } k \geq M \end{cases}
\end{aligned}$$

where E_j and O_j denotes the DFT of the even-indexed numbers e_m and the odd-indexed numbers o_m, respectively.

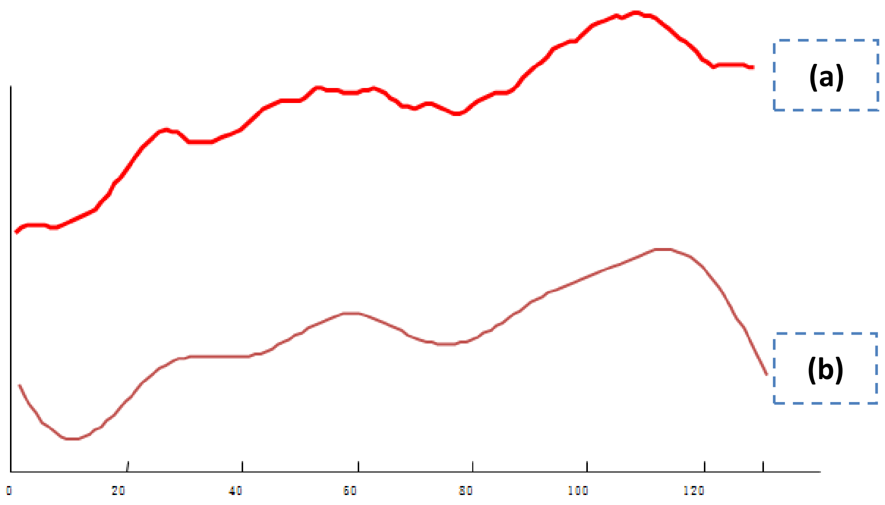


Fig 3.4

Fig 3.4 above illustrates the effect of DFT. In (a) it is the input curve. After performing DFT, the curve becomes smoother as shown in (b).

3.2.4 POLYNOMIAL FUNCTION

Apart from the complicated algorithms above, in fact a curve can also be approximated by using simple regression.

Regression tries to find the "best" curve to fit all the data points. The approximated curve may not pass through all the data points, but the error is guaranteed to be the minimum. Regression is useful when the data has certain amount of noises.

To find the best-fit curve using regression, "Linear Least Square" method can be used. Depend on the complexity of the motions; the degree of the curve that is used to fit the data points can be adjusted. The following shows how a curve with degree 2 can be calculated:

Suppose there are N data points. As the curve is in degree 2, so it must be represented in the following forms:

$$\boldsymbol{y_i = a_0 + a_1x_i + a_2{x_i}^2 + e_i}$$
$$\boldsymbol{for\ i = 1, 2, \ldots, N, where\ e_i\ represents\ the\ error}$$
$$\boldsymbol{between\ the\ best - fit\ curve\ at\ i\ and\ the\ i^{th}\ data\ point.}$$

With N data points, the above equation can be written in matrix form as follows:

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & x_1 & {x_1}^2 \\ 1 & x_2 & {x_2}^2 \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \\ 1 & x_n & {x_n}^2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ \vdots \\ e_n \end{bmatrix}$$

To find the best-fit line, a possible strategy is to minimize the sum of the squares of the residuals between the measured *y* value and the actual data point, i.e.

$$\boldsymbol{S_r = \sum_{i=1}^{n} (y_i - a_0 - a_1 x_i - a_2 {x_i}^2)^2}$$

To minimize S_r, a possible approach is to set the partial derivatives of S_r to be zero and solve the resulting systems of equations such that a_0, a_1 and a_2 can be found.

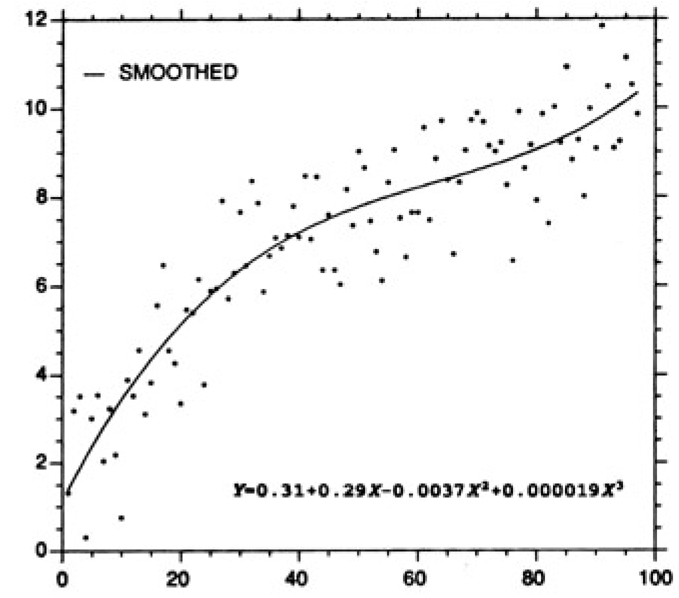


Fig 3.5 A graph showing the best-fit curve after applying linear regression.

3.2.5 RAW FORMAT

Apart from using different approximation algorithms, we have also used the most basic raw data format for computations. Raw data here refers to the actual acceleration values collected from the Wii Remote. They are used directly without any treatments.

3.3 "USER DEPENDENT" AND "USER INDEPENDENT"

Motion classification is largely depending on how user performs the motions. Different users perform the same action in different styles. For example, in writing the number "2", some people may include a small circle at the bottom left part of the number while some may just write it in the same way as the one appear in this report.

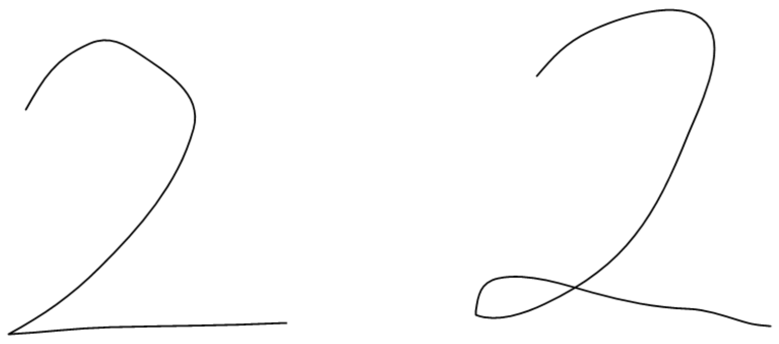


Fig 3.6 Two different styles of writing the number "2"

So here comes the question: How to recognize the motion if different players try to perform the same motion in their own ways? In another words, how to achieve user independent motion classification?

Intuitively, we can include all different styles of the same motions in the database, and compare each motion one by one. This method, however, is time and space inefficient since the number of comparison is large, and storing all the motions is space consuming. Therefore typically, we will observe the pattern of the data, and extract some useful features from it. We then build our rule for classification according to these features extracted from different styles of the same motion.

Achieving user dependent, on the other hand, is a lot easier to be done. The main problem to achieve user dependent is that it is impossible for a user to do the same motion in an identically every time. Besides extracting features from the data, since the motions performed by the same user are more or less the same, therefore we can perform a similarity search on the previously obtained data from that user with a predefined error threshold for classification.

3.4 MOTION DESIGN

Another factor that affects motion classification is the motion itself. For example, having to classify two similar motions, big circle and small circle, will be difficult since the motions are draw in the same way except the size are different.

Designing motions thus becomes a challenge, especially when the number of motions increases. Sometimes, motions have to be redesigned in order to achieve a higher classification rate, or relationships of the motions have to be set up. For instance, a big circle drawing is always followed by small circle drawing.

The main factors that need to be considered in designing motions:

1. What kinds of motions are performing? Are they sword motions? Or they are just 2D numbers drawing in 3D space? What are their features?
2. How the motions are used? Is early recognition required?
3. Are the motions too similar? Are they distinctive enough for classification?
4. Are the motions too complicated for the user to memorize and perform?

3.5 RESEARCHED CLASSIFICATION METHODS

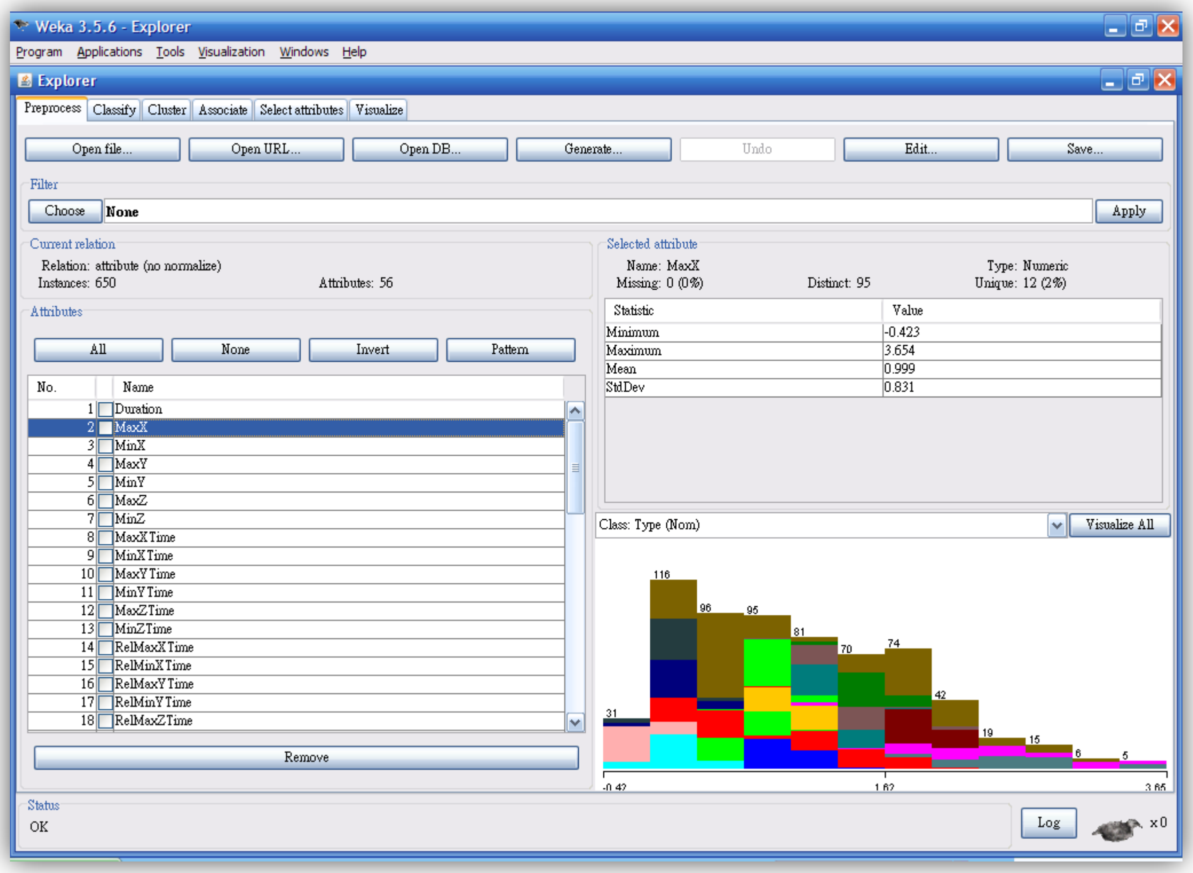
Before we decided to use a particular algorithm for motion classification and recognition, we have studied a number of methods before our final decision. In the 3 sections below we will introduce each of them, the principles behind, and the strength and weaknesses of each of them.

3.5.1 WEKA – A DATA MINING SOFTWARE

3.5.1.1 INTRODUCTION

Weka [7] is an open source software implemented by a research group in The University of Waikato. It is a collection of machine learning algorithms for data mining classes. The software contains different tools for data-preprocessing, classification, regression, clustering, association rules and visualization.

In the book "*Data Mining: Practical machine learning tools and techniques*" [7], it is said there are 3 different ways of using Weka – "Apply a learning method to dataset and analyze the output", "use learned models to generate predictions on new instances", and "apply several different learners and compare their performance in order to choose one for prediction". In our project, we used Weka mainly in the 2nd and 3rd way – we tried several learners to try on a given data set, and we would like to see which learner performs better based on the predictions on new instances, and eventually we can implement classifiers based on the classification results.


Weka 3.5.6 - Explorer
Program Applications Tools Visualization Windows Help
Explorer
Preprocess Classify Cluster Associate Select attributes Visualize
Open file... Open URL... Open DB... Generate... Undo Edit... Save...
Filter
Choose
Apply
Current relation
Attributes
All None Invert Pattern
Remove
Selected attribute
Visualize All
116
96
95
81
70
74
42
31
19
15
6
5
-0.42
1.62
3.65
Status
Log
x 0

None

Relation: attribute (no normalize)
Instances: 650 Attributes: 56

No.	Name
1	Duration
2	MaxX
3	MinX
4	MaxY
5	MinY
6	MaxZ
7	MinZ
8	MaxXTime
9	MinXTime
10	MaxYTime
11	MinYTime
12	MaxZTime
13	MinZTime
14	RelMaxXTime
15	RelMinXTime
16	RelMaxYTime
17	RelMinYTime
18	RelMaxZTime

Name: MaxX Type: Numeric
Missing: 0 (0%) Distinct: 95 Unique: 12 (2%)

Statistic	Value
Minimum	-0.423
Maximum	3.654
Mean	0.999
StdDev	0.831

Class: Type (Nom)

OK

Fig. 3.7 A screenshot of Weka data-preprocessing section

3.5.1.2 TESTED CLASSIFIERS

Weka provides different types of classifier algorithms, which includes – Bayes, Trees, Rules, Functions and Lazy. We have selected 5 classifiers, 3 from Trees and 2 from Rules, to investigate and compare their performances.

	Name	Function [7]
Trees	J48	C4.5 decision tree learner (implements C4.5 revision 8)
	RandomTree	Construct a tree that considers a given number of random features at each node
	REPTree	Fast tree learner that uses reduced-error pruning
Rules	PART	Obtain rules from partial decision trees built using J4.8
	JRip	RIPPER algorithm for fast, effective rule induction
Functions	Multilayer Perception	A Classifier that uses back propagation to classify instances.

Tabel 3.1 Lists of classifiers selected for investigation and their function

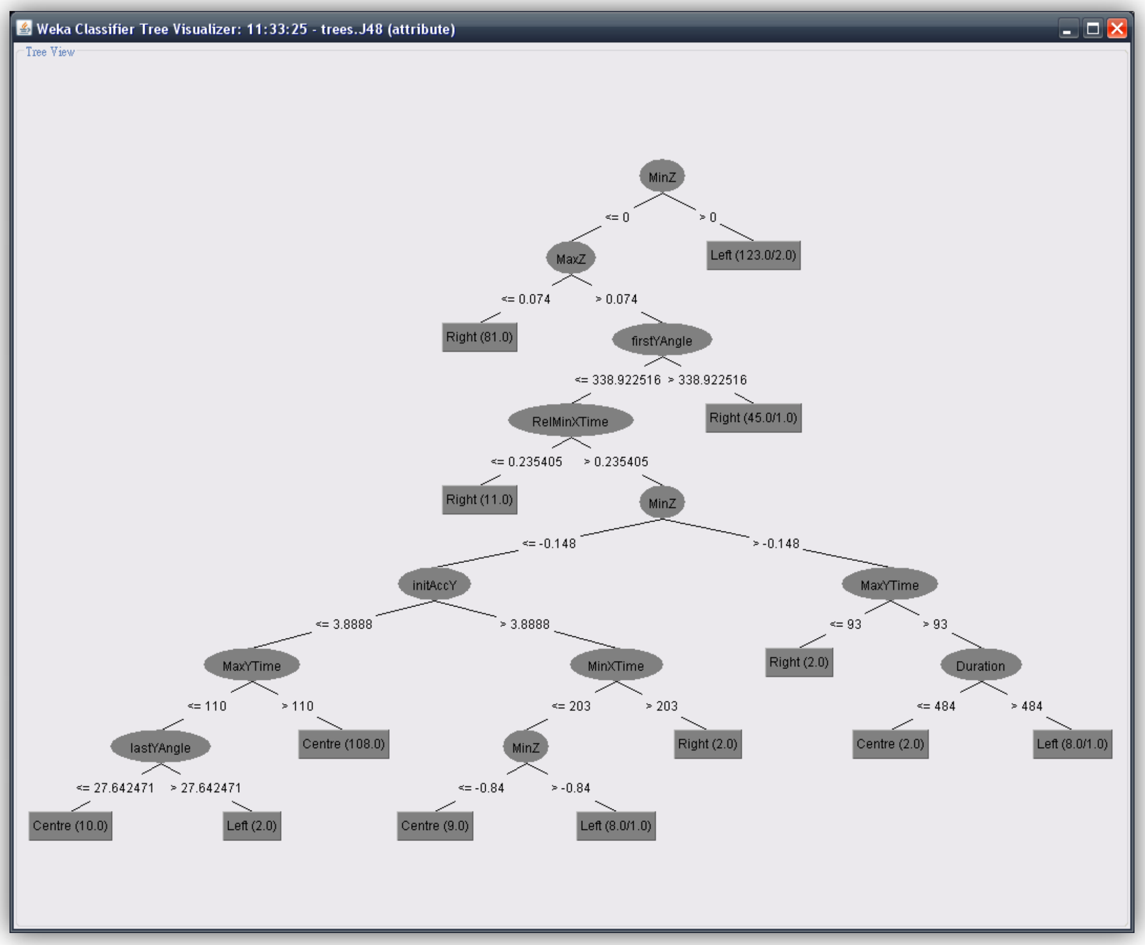


Fig. 3.8 A decision tree generated by Weka using J48 implementation

3.5.1.3 ATTRIBUTES USED FOR CLASSIFIERS

Classifiers will not work just based on the motion data itself, which is just a sequence of numbers without any meaningful information; so we have computed a number of attributes based on the capture motion data, i.e. the time frame number, and the accelerations of the 3 axis, to serve as the inputs for classifiers.

Name of Attribute	Meaning of the attribute
RelMaxXTime / RelMaxYTime / RelMaxZTime	The time relative to the whole motion (**relative time**) when the acceleration is maximum in **positive** X / Y / Z axis direction.
RelMinXTime / RelMinYTime / RelMinZTime	The relative time when the acceleration is maximum in **negative** X / Y / Z axis direction.
relIntersectXY / relIntersectXZ / relIntersectYZ	The relative time when the acceleration of X and Y axis / X and Z axis / Y and Z axis is the same **for the first time**
relZeroAccX / relZeroAccY / relZeroAccZ	The relative time when the acceleration of X / Y / Z axis is zero **for the first time.**
initAccX / initAccY / initAccZ	The average acceleration along X / Y / Z axis in the first 75 milliseconds
endAccX / endAccY / endAccZ	The average acceleration along X / Y /Z axis in the last 75 milliseconds before the motion ends
firstCombinedAcc	The average combined acceleration of X, Y and Z axis in the first 75 milliseconds
lastCombinedAcc	The average combined acceleration of X, Y and Z axis in the first 75 milliseconds
firstXAngle / firstYAngle / firstZAngle	The initial angle of deflection on YX / ZY / XZ plane

Tabel 3.2 List of attributes we have computed

(Cont'd Table 3.2)

Name of Attribute	Meaning of the attribute
endXAngle / endYAngle / endZAngle	The angle of deflection on YX / ZY / XZ plane before the motion ends.
relMaxAccTime	The relative time when the combined acceleration of X, Y and Z axis is maximum.
relMinAccTime	The relative time when the combined acceleration of X, Y and Z axis is minimum.
averageXAcc / averageYAcc / averageZAcc	The average acceleration of X / Y / Z axis throughout the motion
averageCombAcc	The average combined acceleration of X, Y, Z axis throughout the motion

3.5.1.4 Data Pre-processing

To further improve classification accuracy, we have done some pre-processing on the computed attributes before it is used for classification.

3.5.1.4.1 Normalization of amplitude [14]

Normalization is any process that makes something more normal, which typically means conforming to some regularity or rule, or returning from some state of abnormality.

In our case, since our input is a set of data points. Normalization is hardly to be done except on the amplitude of the points. In our normalization, we normalize the amplitude between -10 and 10.

3.5.1.4.2 Discretization of attributes

Discretization refers to the separation of numeric attributes into a smaller number of distinct ranges. This idea exists because some classification algorithms can only deal with nominal attributes only and cannot handle attributes on a numeric scale [7]. So as to increase the reliability of classification, attributes are discretized before actual classification starts.

3.5.1.5 PROS AND CONS AMONG CLASSIFIERS

Among the classifiers we have tested, we got the following conclusion about each of them:

1. When compared to neural networks, tree and rule-based classifiers are easy to be coded. Weka visualizes the classification tree or rules to the user after learning is completed. Obviously this makes coding of the decision trees or rules much easier by just using a number of if statements. However, for neural networks, it is not easy to have prototype within a second and thus we cannot use this to construct classifiers easily.

2. From our trials, we found that neural network can always guarantees accuracy greater than 90%. However, the training time is longer when compare to other algorithms.

3. For tree or rule-based algorithm, as well as neural networks, in order to have an accurate result, a large number of data samples must be ready in the learning step.

3.5.2 EUCLIDEAN DISTANCE ALGORITHM

3.5.2.1 INTRODUCTION

Apart from computing attributes based on the acceleration values, another possible approach for motion classification is to perform similarity search among the input signals and the sample data stored in the system, by computing the Euclidean Distance between the 2 different set of points.

3.5.2.2 WORKING PRINCIPLE

Euclidean Distance is defined mathematically as the "ordinary" distance between two points that one would measure with a ruler [8]. It is computed as follows:

$$For\ 2\ set\ of\ points\ P = (p_1, p_2, \dots, p_n)\ and\ Q = (q_1, q_2, \dots, q_n),$$

$$the\ Euclidean\ Distance\ between\ points\ P\ and\ Q, is\ defined\ as:$$

$$\sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2} = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}$$

To perform similarity search, sequences of acceleration values from both the input signals and the sample data on the same axis, are aligned sequentially. Euclidean Distance is then computed.

The total error of the input motion is defined as the sum of Euclidean Distances for the 3 axis.

To correctly classify the motions, similarity search is repeated for all the samples in the classifier. The classifier will base on the error values, and classify the input motion as a particular motion if the error value is the minimum.

3.5.2.3 STRENGTHS AND WEAKNESS

Euclidean Distance algorithm is straight forward to implement, and classifiers making use of this algorithm can allow user-defined motions to be added to the classifiers. From this aspect, this is better than the tree-based and rule-based classification algorithms mentioned in section 3.5.1.2.

However, the error rate for classification using Euclidean Distance algorithm is high due to its linear one to one mapping. In particular, if the duration for the input motion is longer than the sample motion, then data points near the end is truncated and thus increasing the error. This algorithm assumes there are no variations between 2 sequences in terms of time or speed, which is undoubtedly not suitable for motion classification since different players may do the same motion differently in different trials.

3.5.3 DYNAMIC TIME WARPING (DTW)

3.5.3.1 INTRODUCTION

Dynamic Time Warping (DTW) is yet another algorithm for measuring similarity between two sequences, even if there are variations in time or speed. It can be used to analyze any data which can be turned into a linear representation finally.

One example of the restrictions imposed on the matching of the sequences is on the monotonic property of the mapping in the time dimension. Continuity is less important in DTW than in other pattern matching algorithms; DTW is an algorithm particularly suited to matching sequences with missing information, provided there are long enough segments for matching to occur.[9]

Similar to Euclidean Distance algorithm mentioned in chapter 3.5.2, the motion is classified based on the errors with the sample motions in classifiers.

3.5.3.2 WORKING PRINCIPLE

In general, Dynamic Time Warping tries to find the best match between two given sequences under some constraints. The sequences are mapped non-linearly within the time dimension so that similarity can still be detected regardless of some non-linear variations in time and speed.

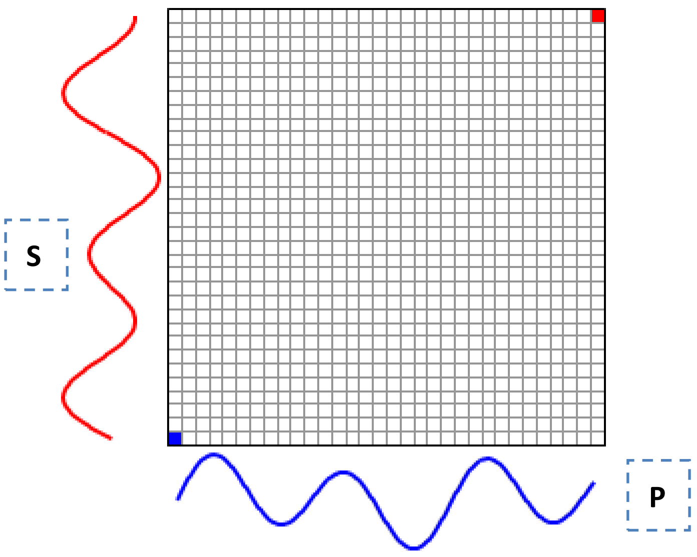


Fig. 3.9 An example grid for DTW [11]

Fig. 3.9 shows a grid for DTW. It forms a two dimensional cost matrix when the value of each cell is filled. Generally speaking, red curve represents the stored curve, and blue curve represents the input curve. In our project, the red curve resembles the sequence of acceleration values from a model motion in the classifier; while the blue curve resembles the sequence of acceleration values captured from the Wii Remote by user's motion.

In order to find the best match between the two sequences, we can find a path through the grid which minimizes the total distance to travel from the blue square at the bottom left corner, i.e. the beginning of two curves, to the red square at the top right corner, i.e. the end of two curves.

The equation for computing values at each cell in the cost matrix is defined as follows [11]:

$$\gamma(i,j) = d(s_i, p_j) + \min[\gamma(i-1,j-1), \gamma(i-1,j), \gamma(i,j-1)]$$

Formula 3.1 Function for computing values in each cell in cost matrix

where $d(s_i, p_j)$ refers to the difference in values between ith point of stored curve and jth point of the input curve.

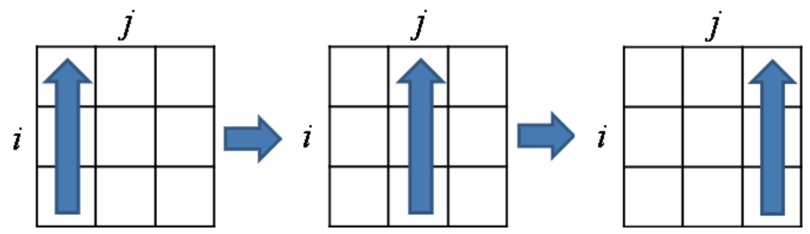


Fig. 3.10 The order of the cost matrix is filled

∞	14.6	13.3	10.5	8.2	7.3	6	5.4	4.8	4.6
∞	11.2	10.2	7.9	6.2	5.5	4.7	4.4	4.3	4.8
∞	8	7.2	5.6	4.3	3.9	3.6	3.8	4.6	5.6
∞	5.5	4.9	3.8	3.1	2.9	3.3	4.2	5.5	7.2
∞	3.5	3.1	2.6	2.3	2.7	3.5	4.5	6.2	8.1
∞	1.8	1.6	1.7	2.2	2.9	4.2	5.7	7.9	10.5
∞	0.7	1.3	1.6	2.3	3.3	4.8	6.6	9.1	11.9
0	∞	∞	∞	∞	∞	∞	∞	∞	∞

Fig. 3.11 An example of the cost matrix, a grid with all values filled

After computing the all the values, the warping path, which is the shortest path from the blue square to the red square can then be found by using backward induction, is shown in the figure below:

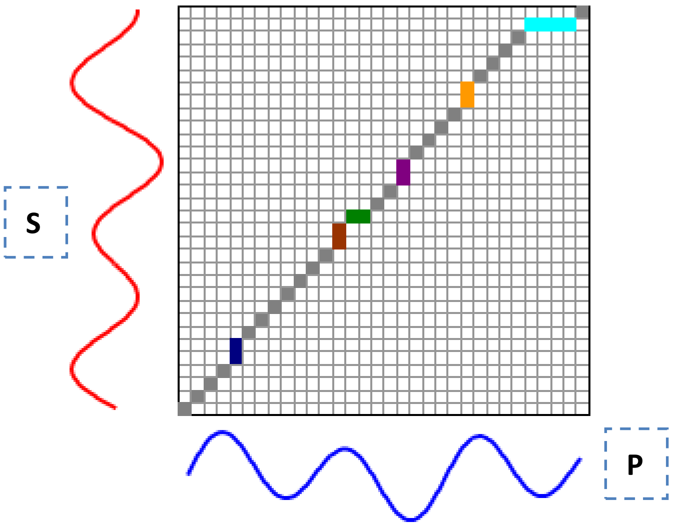


Fig. 3.12 DTW grid with warping path found [11]

With this warping path, a non-linear mapping between two curves is formed, and the error is then computed as well.

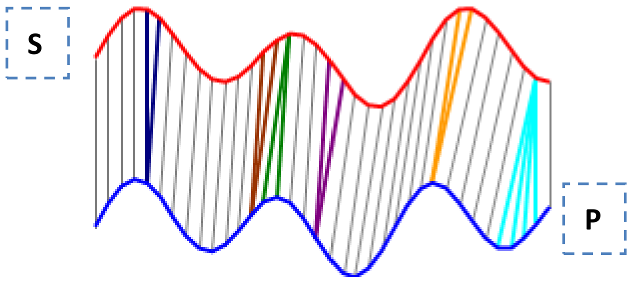


Fig. 3.13 A non-linear mapping between 2 different curves [11]

Fig. 3.13 shows a non-linear mapping between curves S and P mentioned previously. Attention can be drawn to the colored (and bolded) lines, in which some points map to multiple points, and some other points are mapped with multiple points. The mapping is based on the result from the grid with warping path found.

Although DTW can allow non-linear mapping, however its native version spends too long time to compute values that will not be used. Generally the mapping has some properties [10]:

1. Monotonic condition – the indexes either stay or increase, and will never go back.
2. Continuity condition – only one step advancement at a time.
3. Adjustment window condition – the curve generally does not vary much from the diagonal.
4. Slope constraint condition – the path is not too sheep or shallow so that short sequences is not matched with long sequences.

By considering these conditions, optimizations can be made. One example is to impose global constraints of the warping path such that boxes outside a particular area will not be computed [11].

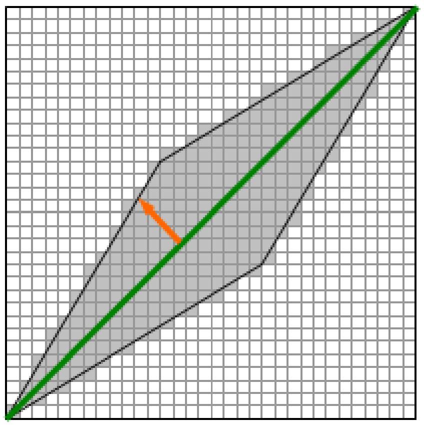


Fig. 3.14 A DWT grid with global constraints imposed.

Fig. 3.14 shows the DWT grid with global constraints imposed. Since the warping path must be obeying the 4 properties stated above, so the values outside the grey area will not be considered and computed, and thus saving computation time.

3.5.3.3 Why Dynamic Time Warping?

When compared to Euclidean Distance algorithm, Dynamic Time Warping allows non-linear mapping, thus non-linear variations on time or speed can be overcome. We illustrate the advantage of Dynamic Time Warping with an example below.

Fig 3.15 in the next page shows how a stored curve (the curve on top) is compared with the input curve. Obviously 2 curves are doing the same thing. However, in the stored curve, the 4 crests occur consecutively; while in the input curve, 2 crests occur first, and then followed by a short period of rest, and another 2 crests occur.

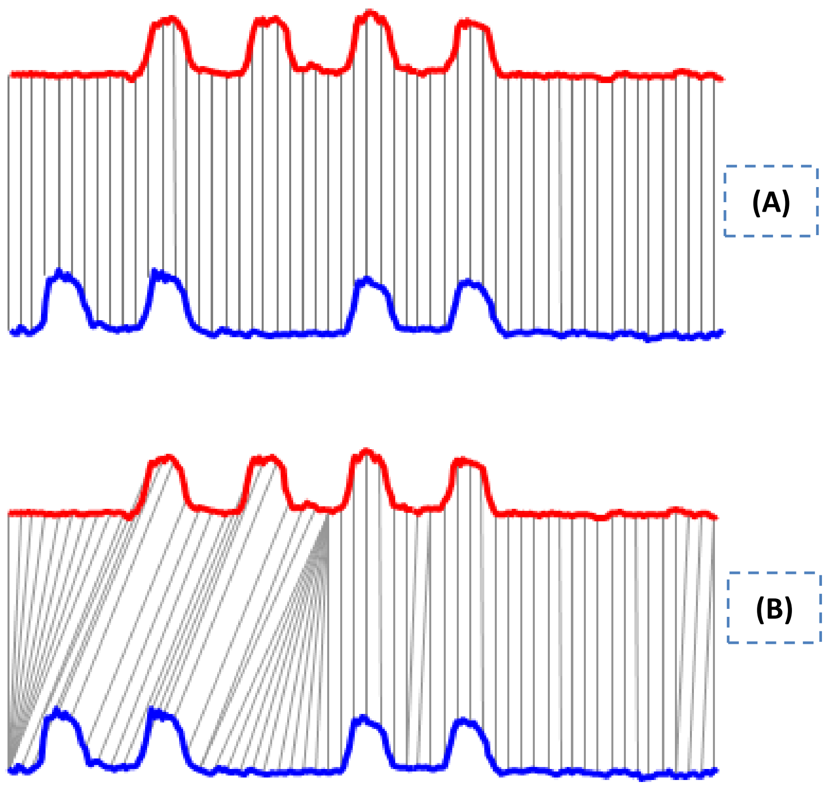


Fig. 3.15 Comparisons between Euclidean Distance algorithm (graph A) and Dynamic Time Warping (graph B) [11]

As shown in graph A, if Euclidean Distance algorithm is used, the errors will be large due to appearance of crests at different time. On the other hand, if Dynamic Time Warping is used, the time difference in the occurrence of crests can be overcome since one point can map to multiple points. That means crests at the one curve are mapped to corresponding crests at the other curve. The computed error will then be much smaller, and thus the input can then be classified at a higher accuracy.

From the example, we can see why Dynamic Time Warping is more preferable to Euclidean Distance algorithm.

3.5.3.4 STRENGTH AND WEAKNESS

One obvious advantage of Dynamic Time Warping is the high accuracy. No matter user performs the motion with different speed, still it can be classified correctly. On the other hand, it still allows players to define their own motion.

However, as the motion is recognized based on similarity search, so if there exists two or more similar motions, the classifier is unable to classify them accurately. On the other hand, it also assumes players to always hold the Wii Remote in the same manner; otherwise it cannot be classified accurately as well.

CHAPTER 4 – THE CLASSIFICATION ALGORITHM – DYNAMIC TIME WARPING

We have finally chosen Dynamic Time Warping as the classification algorithm. In this chapter we are going to describe how we use Dynamic Time Warping for motion classification.

This chapter comprises the following sections:

4.1 WHY DYNAMIC TIME WARPING IS SELECTED?

After studied different algorithms mentioned before, we decided to choose Dynamic Time Warping as our classification algorithm. There are a number of reasons for that:

1. Only one motion data is needed - Since DTW itself is a similarity search algorithm, one perfect motion data will be enough to be used for classification. This is especially useful to achieve user-defined motion function so those users are not required to train the classifier a lot.

2. Similarity value output - Unlike typical data mining method, the result of DTW can be used as an indicator on how similar the motion is performed, and decided to accept or reject the motion according to the similarity.

3. Fast Learning - Unlike Neural Network and other data mining methods, to add new motions, we just record the motion into the database.

4. Extensible - DTW is based on the distance between two points, without any limitation on the number of dimensions. Therefore we can apply DTW in 3 dimensional spaces by using Euclidean Distance.

5. Optimization is possible - There are several existing methods to optimize the time and space complexity of this algorithm.

4.2 Initial coding – Used Dynamic Time Warping wrongly

When we first applied the DTW algorithm, we did not consider the fact that the accelerations of three axes should not be considered separately. Therefore, in the beginning we applied DTW on each axis and executed DTW three times on the same motion. This approach is wrong because DTW is about the mapping of the points. If DTW is applied separately, most likely the mappings are not consistent among other axes, even though the classification rate are quite high using this wrong approach.

4.3 Extension of Dynamic Time Warping

We discovered that we applied DTW wrongly when we tried to implement the algorithm for finding the warping path. As DTW finds the mapping based on the distance, instead of using the distance between two points in one axis, we extended DTW to calculate the Euclidean Distance between two points in 3 dimensional spaces, by using the acceleration values in one time frame as a 3 dimensional coordinate. On average, the extended version of DTW performs much better than the initial coding one.

Below is an example showing the details of the extension:

Time	X Acceleration	Y Acceleration	Z Acceleration
0	1	3	7
15	4	5	2

Table 4.1 Example illustrating the details of extension

Distance between Two Points in X Axis $= |1 - 4| = 3$

Distance between Two Points in Y Axis $= |3 - 5| = 2$

Distance between Two Points in Z Axis $= |7 - 2| = 5$

Distance between Two Points in 3 Dimensional Space

$= \sqrt{(1-4)^2 + (3-5)^2 + (7-2)^2} = 6.16$

The advantage of this algorithm is that it can be easily extensible when more data are available. For example, if Wii Remote expansion port is connected with Nunchuk, three more axes of accelerations will be available. If the two set of accelerations on the two remotes are correlated, based on the principle, since Euclidean Distance is generalized to n-Dimensional Space, we can calculate the Euclidean Distance in 6 Dimensional Space.

4.4 TUNING

In order to achieve a higher classification rate, different methods are used to tune the input data for the Dynamic Time Warping algorithm.

4.4.1 FILTERING INPUT

Before we discovered that we applied DTW wrongly, two methods are used for filtering the input motion data in order to remove noise, namely polynomial regression and low pass filtering. It seems that there are no significant improvements. As the DTW algorithms we are using now are in three dimensional spaces, these two filtering methods can no longer be used.

4.4.2 DATA TRANSFORMATION

Besides treating the three axes of acceleration values as a Cartesian coordinate, we can transform it into other coordinate system. The below shows the coordinate systems we tried before we discovered the bug on applying DTW. The coordinate is transformed into three angles, and the length of the vector. However, the result isn't very good. Possibly it is because there is "jumping" of values in the angle when some of the coordinate values change their signs. As the DTW algorithms we are using now are in three dimensional spaces, data transformation can no longer be applied.

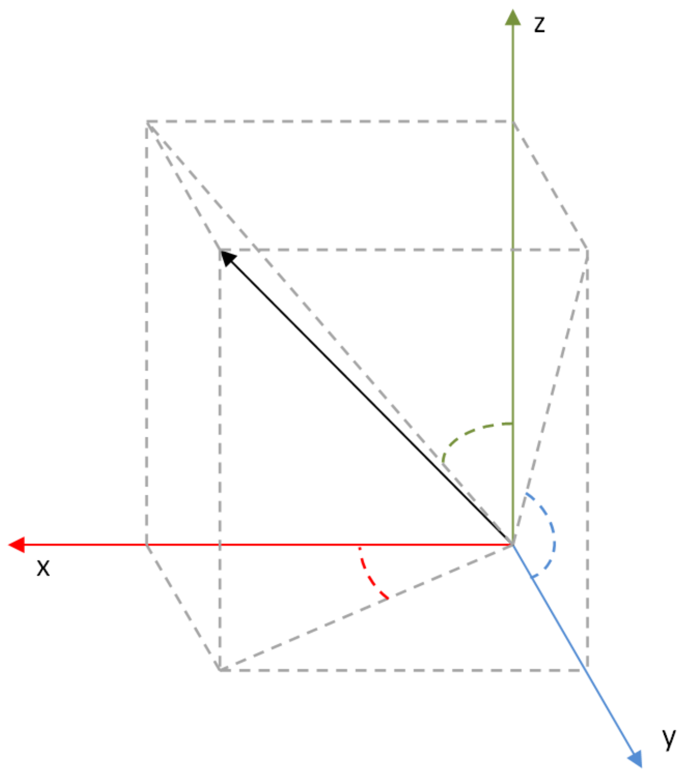


Fig 4.1 The coordinate system we tried before

4.4.3 MODEL MOTION TUNING

DTW is performed between the to-be-classify motion data and the model motion data. As a result, the quality of the model motion data largely affects the classification rate. Therefore, we tried the following in order to minimize this problem. The result of the analysis is shown after this section.

4.4.3.1 NUMBER OF SAMPLES MATCHING

Typically the to-be-classify motion data is matched with each model motion data once only. However, we can allow the user to input some more motion data for the same motion as model motion data. Therefore the to-be-classify motion data can be used to compare with other slightly varied model motion data.

4.4.3.2 ERROR COMPENSATION

Besides having more samples for comparisons, we can, on the other hand, take some more motion data from the user, and calculate the average DTW error among those motion data. When doing the classification, the newly computed DTW error can be subtracted by the average DTW errors obtained previously.

4.4.3.3 AVERAGE OF MOTION DATA

To increase the quality of the model motion data, one of the methods is to take the average of the motion data from the user as model motion data. Several numbers of motion data are obtained from the user. By finding the warping path, we obtain the mapping of each point between different pairs of motion data. For each set of mapped points, we calculate the average coordinate value for that point, and make it as an average motion data.

4.5 ANALYSIS RESULT OF MODEL MOTION TUNING

4.5.1 MOTIONS USED

Several categories of motions are used. Below is the list of motions:

- Five directions
  - Horizontal Left
  - Diagonal Left
  - Vertical Down
  - Diagonal Right
  - Horizontal Right
- Cursive Writing Letters
  - A, B, C, D, E, G

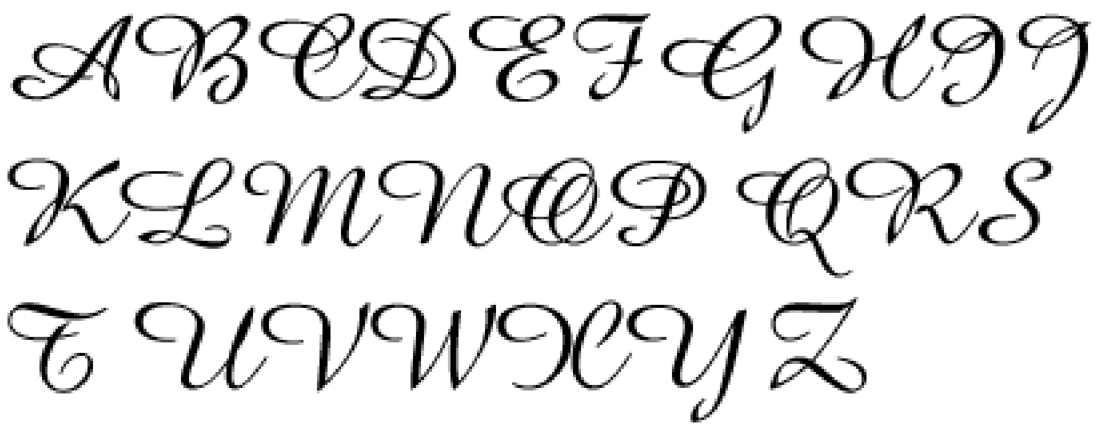


Fig 4.2 Cursive Writing Letters

- Discrete Motions
  - Forward Left Up
  - Forward Right Down
  - Forward Down Left
  - Back Right
  - Back Down Left
  - Forward Right Up
  - Forward Up Left
  - Forward Left Down

- Numbers
  - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

- Dao Motions
  - Dao motions are too complicated to be described in text. Below are some photos of how the five Dao motion look like.
    - Motion 1 – Horizontal Left
    - Motion 2 – Top Right Circle
    - Motion 3 – Left Circle
    - Motion 4 – Forward
    - Motion 5 – Vertical Circle

- ***Motion 1 – Horizontal Left***

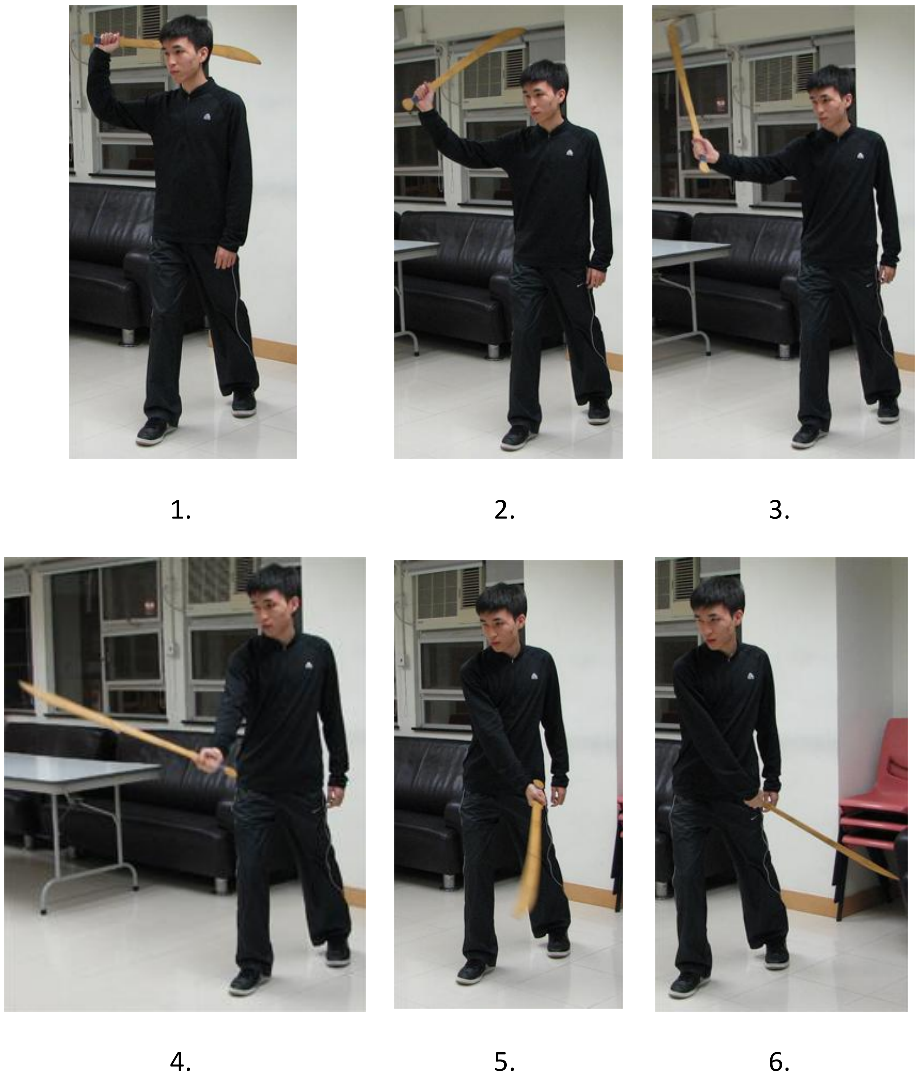
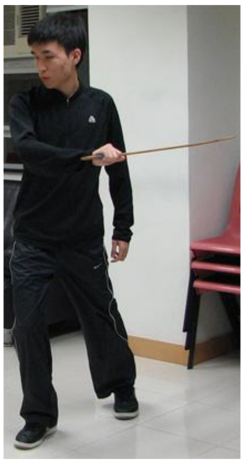
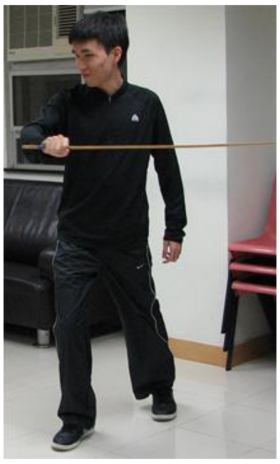
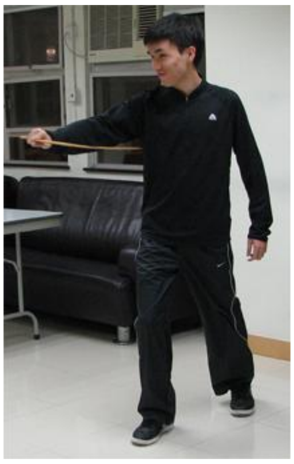


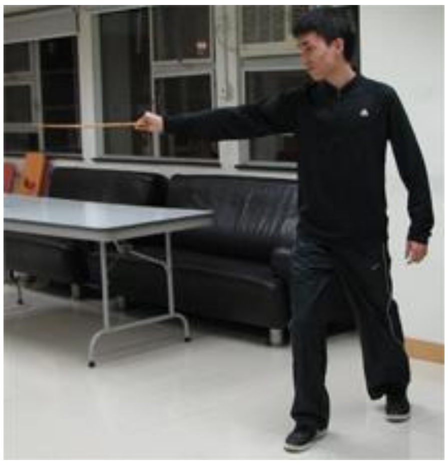
Fig 4.3 Photos illustrating Dao Motion – "Horizontal Left"

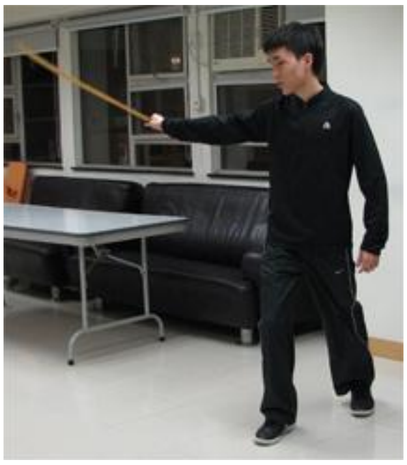
- ***Motion 2 – Top Right Circle***

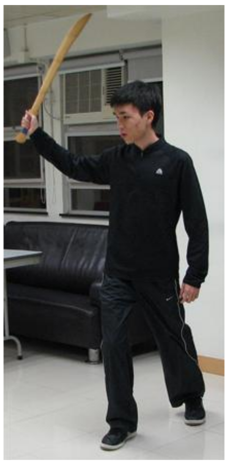

1.

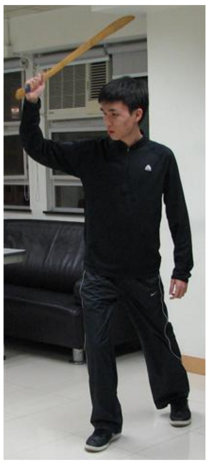

2.

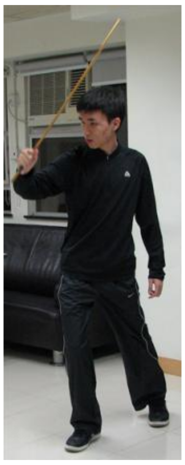

3.


4.


5.


6.


7.


8.

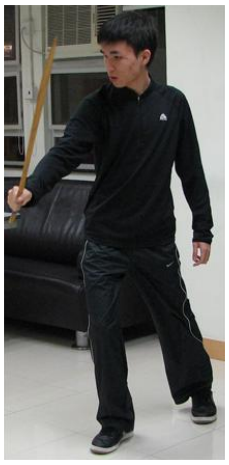
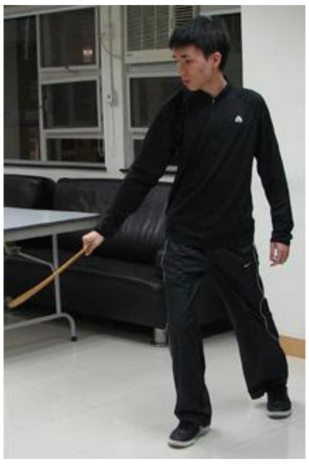

9.

Fig 4.4 Photo illustrating Dao Motion – "Top Right Circle"


10

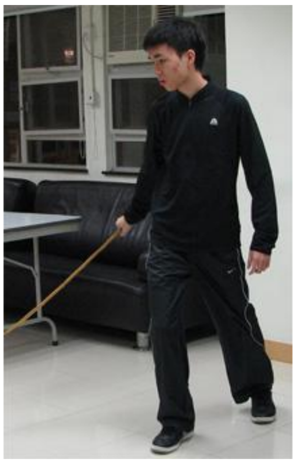
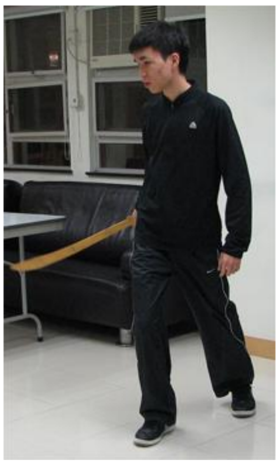
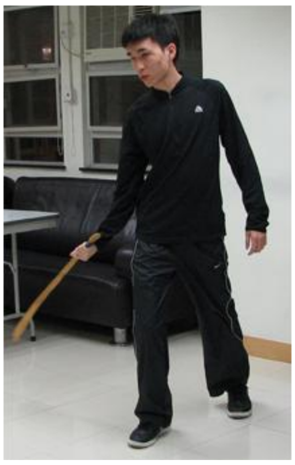
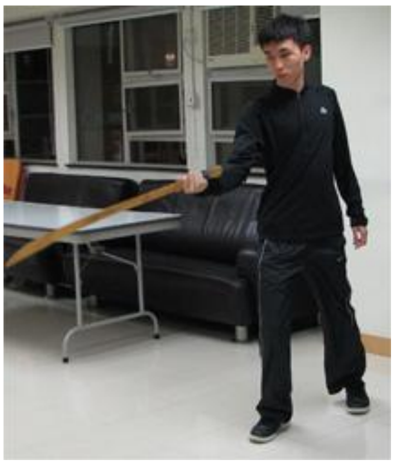

11.

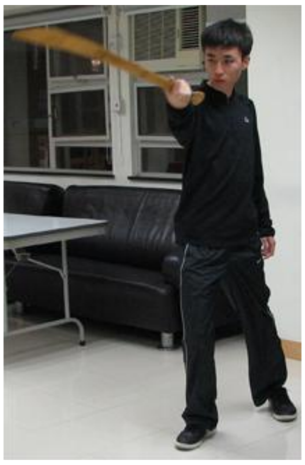
Fig 4.4 Photo illustrating Dao Motion – "Top Right Circle" (cont'd from last page)

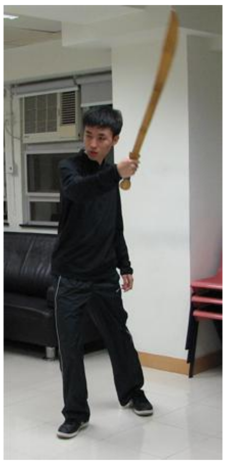
- ***Motion 3 – Left Circle***


1.


2.


3.


4.


5.

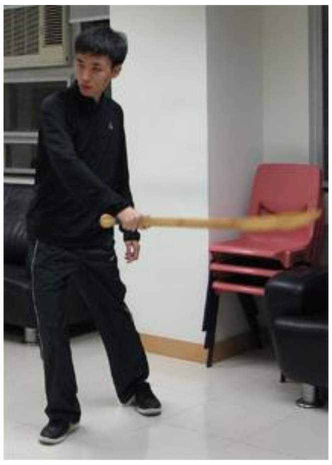
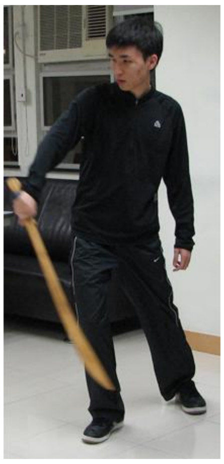
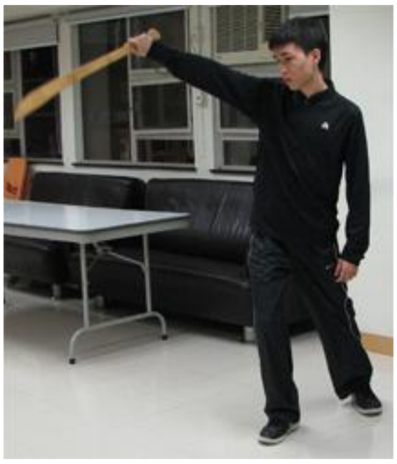

6.

Fig 4.5 Photo illustrating Dao Motion – "Left Circle"



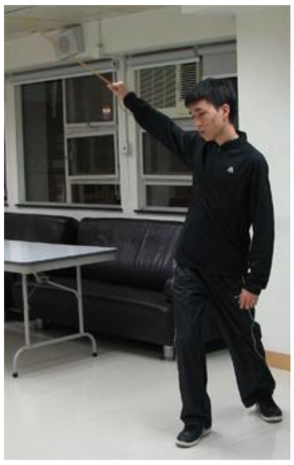
7.



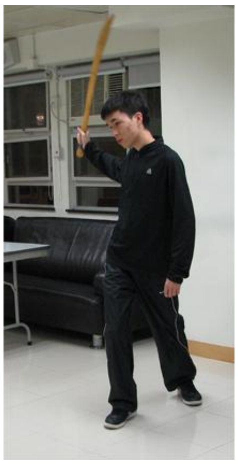
8.

Fig 4.5 Photo illustrating Dao Motion – "Left Circle" (cont'd from last page)

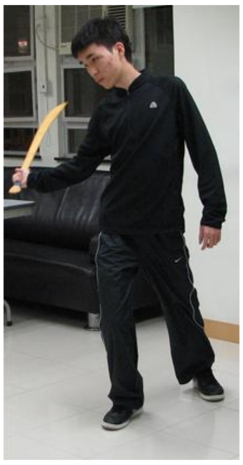
- ***Motion 4 – Forward***



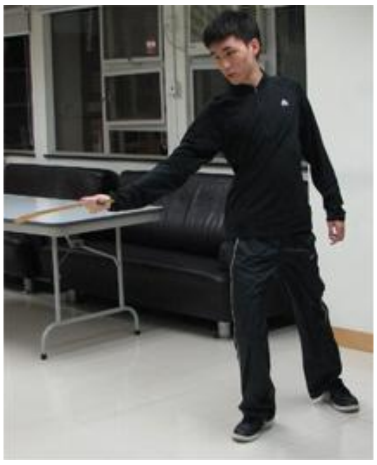
1.



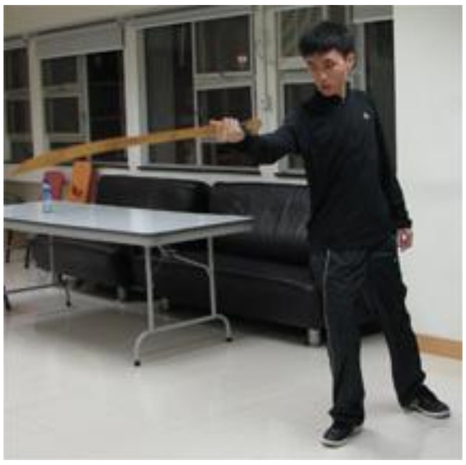
2.



3.



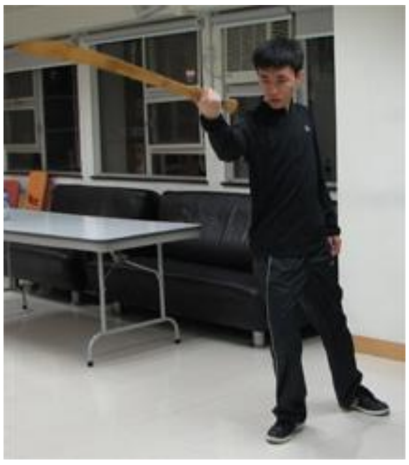
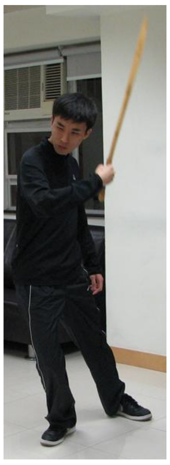
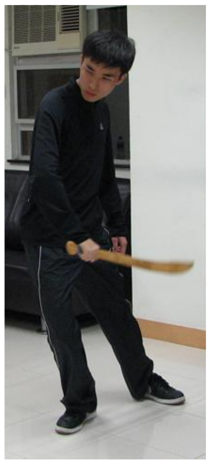
4.



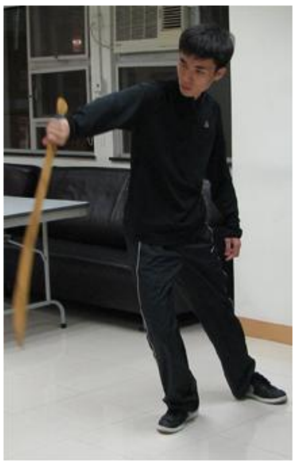
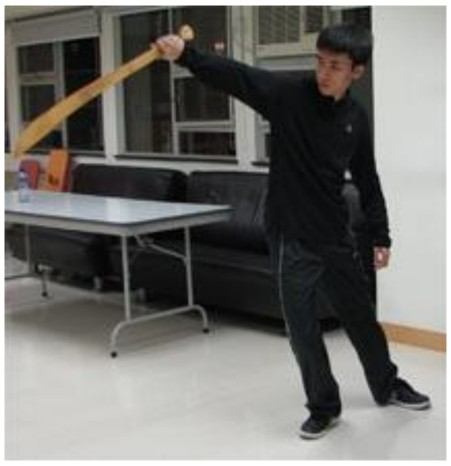
5.

Fig 4.6 Photo illustrating Dao Motion – "Forward"

- ***Motion 5 – Vertical Circle***

1. 2. 3.

4. 5.

Fig 4.7 Photo illustrating Dao Motion – "Vertical Circle"

4.5.2 CONFIGURATION USED

Eight different configurations are used in the analysis. Below is the list of configuration:

- 1 Sample Matching
- 1 Sample Matching, with Error Compensation from 3 Samples
- 3 Samples Matching
- 3 Samples Matching, with Error Compensation from 3 Samples
- 5 Samples Matching
- 5 Samples Matching, with Error Compensation from 3 Samples
- Average of 3 Motion Data
- Average of 3 Motion Data, With Error Compensation from 3 Samples

4.5.3 RESULT

Table 4.2 in the next page shows the corrected motion classification rate (grouped by categories of motion) using different configurations specified above. Each motion in a category contains at least 30 samples from the involved players.

Motion / Configurations	Five Directions			Letters	Discrete	Dao Motion	Numbers			Average
	Player A	Player B	Player C	Player A	Player A	Player C	Player A	Player B	Player C	
1 Sample	95.33 %	91.67 %	90.71 %	100 %	99.51 %	100 %	97.2 %	78.8 %	81.07 %	92.70 %
1 Sample with Error Compensation from 3 Samples	98.67 %	96.67 %	91.43 %	100 %	98.52 %	100 %	96 %	78.8 %	96.43 %	95.17 %
3 Samples	88.67 %	91.67 %	92.86 %	100 %	99.51 %	100 %	97.6 %	88.8 %	100 %	95.46 %
3 Samples with Error Compensation from 3 Samples	92.67 %	91.67 %	91.43 %	100 %	99.51 %	100 %	97.2 %	97.2 %	100 %	96.63 %
5 Samples	92.67 %	94.17 %	95.71 %	100 %	99.51 %	100 %	96 %	95.2 %	100 %	97.03 %
5 Samples with Error Compensation from 3 Samples	94 %	95 %	95.71 %	100 %	99.51 %	100 %	97.2 %	96.8 %	100 %	97.58 %
Average of 3	91.33 %	92.5 %	90.71 %	100 %	99.51 %	100 %	96.8 %	95.6 %	100 %	96.27 %
Average of 3 with Error Compensation from 3 Samples	100 %	90.83 %	87.14 %	100 %	99.51 %	99.29 %	96 %	75.6 %	99.64 %	94.22 %

Table 4.2

4.5.4 CONCLUSION

From table 4.2, we can conclude that, on average, the more the samples, the higher the accuracy is, at the expense of time complexity. But there are cases that more samples can lead to a drop in accuracy. One possible reason is that the model motion data of the newly included samples are of poor quality.

On the other hand, error compensation on average increases the accuracy.

When comparing the results of the two different configurations on an average of 3 motion data, we find that applying error compensation will have a lower accuracy. Possibly it is because the error has already been included when doing the average of data. Therefore applying error compensation is actually considering the error twice and therefore the accuracy is lowered.

When we look at different categories of motions, we discovered that if motions are too similar, the classification will have a lower accuracy.

All in all, the quality of model motion data and the design of motion largely affect the accuracy. Therefore more model motion data are needed to improve the quality of model motion data. On the other hand, depending on the requirements, if accuracy is more important than speed, we can run DTW a few more times on different motion data to further improve the accuracy. Hybrid of the three methods may also be considered.

CHAPTER 5 – THE CLASSIFIER

In the first semester, we studied different representations of the data, different classification methods, as well as different methods to increase the classification rate. We chose the algorithm for our classifier as well as the data representation.

In the second semester, we applied the methods studied in the last semester in increasing the classification rate. In addition, we implemented some more features to enhance our classifier.

The following is the configuration we used:

- Weapon to study: Dao (Sabre)
- Representation of Data: Raw Format
- Classification Algorithm: Dynamic Time Warping on 3D Space

This chapter comprises the following sections:

- 5.1 Terminology
- 5.2 Classification Working Flow
- 5.3 Chosen Weapon – Dao
- 5.4 Chosen Data Representation – Raw Format
- 5.5 Chosen Algorithm – Dynamic Time Warping

5.1 TERMINOLOGY

For the sake of easy understandings, we defined some terms here which will be used in the following sections.

Terminologies	Meaning
Frame Data	Accelerations data of three axes obtained from Wii Remote in a particular time
Motion Data	A collections of time frame data, representing the whole motion
Motion Database	A collection of different motion data of different motions.
DTW	Dynamic Time Warping Algorithm
DTW Error	The result generated by using Dynamic Time Warping
Model Motion	The sample motion used in Dynamic Time Warping for comparison with other motion
Average Motion Data	By finding the mapping of the points using DTW, we can find the average of each acceleration values for each point and generate a new average motion data.
Motion FSM	Motion Finite State Machine

Table 5.1 Terminologies which will be used in the later chapters

5.2 CLASSIFICATION WORKING FLOW

The classification working flow is divided into three main steps –obtaining data, data preparation, and classification.

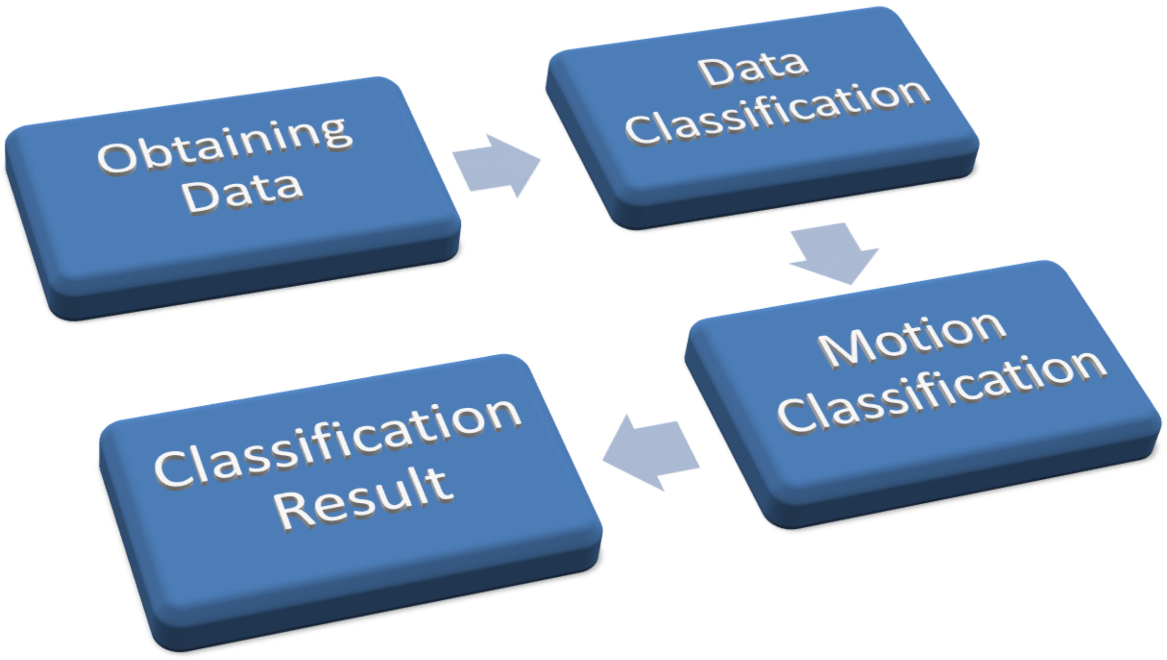


Fig 5.1 The classification workflow

1. Obtaining Data

   Before anything can be done, we have to connect to the Wii Remote to obtain the motion data as model motion data. By using the API *WiiYourself!*, we can obtain different kinds of information such as accelerations of the x, y, z axes at a particular time. We call this the frame data. Currently, frame data are reported at an interval of 15 milliseconds. With these frame data from one motion, we form the "Motion Data".

2. Data Preparation

   Before a motion data is treated to be a model motion data, we can perform some kind of preprocessing to make the model motion data more perfect. For example we can calculate the average motion data. We can also capture more motion data and use them as model motion data of the same motion.

3. Classification

   Finally it is the time to do the classification. We first obtain a motion data that has to be classified, and then put it together with the motion database into the classifier. Depend on the needs of programmers, the classifier can return either the name of the best matched motion, with the associated DTW error, or a list of names with associated DTW errors of each motion exist in the database.

5.3 CHOSEN WEAPON – DAO

In the Legendary of Eighteen Weapons, Dao (Sabre), Qiang (Spear), Jian (Sword) and Gun (Staff) are referred as "The General of All Weapons".[17] As a beginning, we chose to use Dao motions since it involves one controller - Wii Remote only. On the other hand, we can have the assumption on how the user hold the Wii Remote, since dao itself is single-edged-bladed weapon. Due to the time limitation, we are not able to study other weapons.

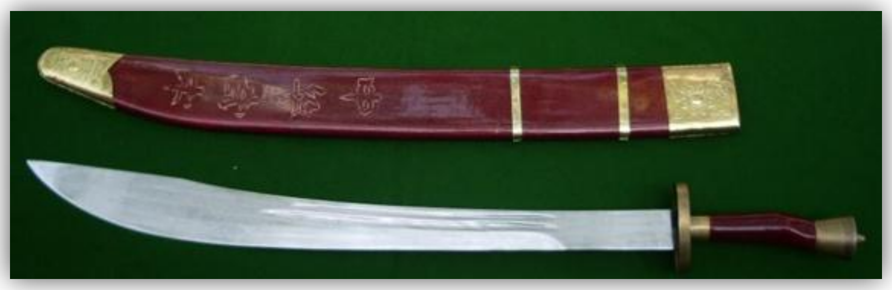


Fig 5.2 Dao

5.4 CHOSEN DATA REPRESENTATION – RAW FORMAT

In the beginning, the first thing we did was to study the representation of the data. Different kinds of representations were studied and were mentioned in section 3.2. However, since there are no immediate available codes for Piecewise Linear Representation, and also we have difficulties in understanding and applying Discrete Fourier Transform, therefore we choose raw format for the data representation. The original idea for introducing polynomial representation is to carry out noise reduction. However, polynomial representation shows insignificant improvement on classification rate, making us to decide to stick to the raw format.

5.5 CHOSEN ALGORITHM - DYNAMIC TIME WARPING

Please refer to the dedicated chapter, Chapter 4 - The Classification Algorithm – Dynamic Time Warping, for the algorithm.

5.6 ENHANCEMENT IN APPLYING ALGORITHM - EARLY CALCULATION

5.6.1 MOTIVATION

In our previous classifier in first semester, all the calculation is done after the player release the button. The calculation mainly involves applying the Formula 3.1 to calculate the cost matrix like Figure 3.11. The size of this matrix is proportional to the length of the model motion data and the motion data obtained by the user.

Knowing this fact, for a long motion, the time required for calculation will be longer. As a result, user could experience lagging when a long motion is performed. Therefore the motivation of this enhancement is to reduce the delay in classifying the motion.

5.6.2 IMPLEMENTATION

According to Fig. 3.10, the cost matrix is filled cell by cell in a column, and then column by column to form the matrix. Recall Formula 3.1, to calculate $\boldsymbol{\gamma(i,j)}$, we need $\boldsymbol{d(s_i, p_j)}$, $\boldsymbol{\gamma(i-1,j-1)}$, $\boldsymbol{\gamma(i-1,j)}$ and $\boldsymbol{\gamma(i,j-1)}$. As soon as we obtain a frame data from the user, we will be able to calculate the value of $\boldsymbol{d(s_i, p_j)}$. Since the values of $\boldsymbol{\gamma(i-1,j-1)}$, $\boldsymbol{\gamma(i-1,j)}$ and $\boldsymbol{\gamma(i,j-1)}$ have been calculated in previous iterations, we can immediately obtain $\boldsymbol{\gamma(i,j)}$.

Therefore, we can actually start calculating the cost matrix as soon as the first frame data is available when the user starts performing the motion. When a frame data is available, the next column in the cost matrix is calculated. In this way the calculation is performed, while the user is performing the motion at the same time. With the computational power of PC, although the calculation is done online, it does not consume much CPU power. This new method actually reduces the delay in classifying motion. For a database with five motions, using the previous method will take a delay of around 78 milliseconds, while the new method reduced the delay to less than 15 milliseconds.

On the other hand, if we only require the similarity of the two motions, what we actually need is not the whole cost matrix, but the values of in the cell with the largest indices according to Formula 3.1, that is:

$$\boldsymbol{\gamma(i_{max}, j_{max})}$$

where $\boldsymbol{i_{max}}$ is the number of frame data of the model motion data and $\boldsymbol{j_{max}}$ is the number of frame data in the user motion data.

Therefore, it is possible for us to keep only values in the previous column, since other columns are not used again in calculating the cost matrix.

5.6.3 ADVANTAGES & LIMITATIONS

The main advantage of the new method is that the calculation of the cost matrix is done online, and therefore the delay of the classification is not observable.

The limitation is that, depend on the number of motions to classify and the requirement of the programmer, this method may result in higher memory usage when compare to the previous method. For example if we required need to know all cost matrices, memory usage will be high since we now need to store all the calculated columns of all motions while the motion is performing.

5.7 DATA PREPROCESSING

In the last semester, we studied different methods in tuning the model motion data. In this semester, we choose the tuning method that we have studied, implemented and applied these methods into our classifier. Currently our classifier uses the tuning method "Average of Motion Data" mentioned in chapter 4.4.3.3. "Number of Samples Matching" mentioned in chapter 4.4.3.1 is also implemented, although this tuning should not regarded as data preprocessing.

5.8 Motion Finite State Machine

5.8.1 Motivation

Recall the training of Chinese martial arts mentioned chapter 1.6, the training require the learner to learn in three different aspects, namely the basic, the form and the application. Motion finite state machine is actually designed to cope with the training of the form and the application. A sequence of states in a finite state machine represents a series of pre-determined motions in a form. Different transitions of a state represent different possible motions that can be used for a user to react given a situation, or represent how different motions can be chained so that the user can utilize the technique learned from different forms.

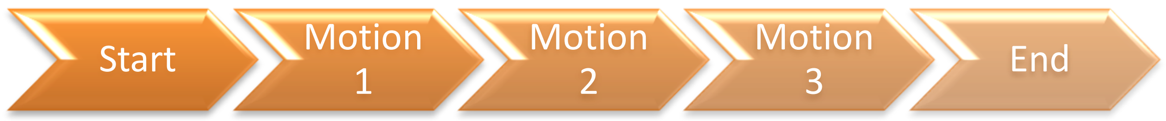


Fig. 5.3 Motion FSM representing a Form

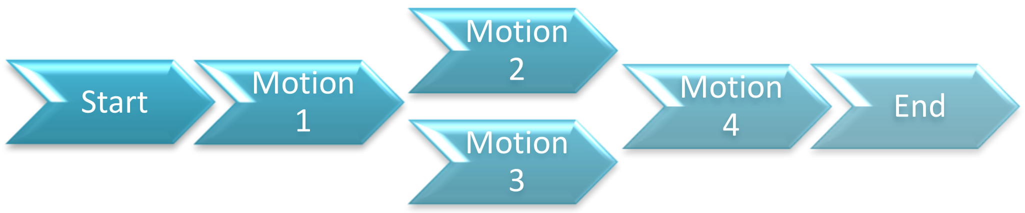


Fig. 5.4 Motion FSM representing how different motions can be chained

5.8.2 Advantages

There are several advantages in implementing motion FSM:

- Simulate the training of Chinese martial arts

  Chinese martial arts are different from just a free fighting game. Learners are required to memorize all the motions in a form in the beginning. Then, they need to apply the techniques they learnt in different forms according to a situation. These forms can actually be represented by a motion FSM, and thus simulate the training of Chinese martial arts with a motion FSM.

- Number of motions can be as much as you want!

  In a classifier without motion FSM, although it is possible to include many motions in the classifier, the long computation time to classify the motions make it impossible to apply in any games. However, with motion FSM, as long as the classification of following motions completes in a reasonable amount of time,, the number of motions in the classifier can be as much as you want!

  For example, in Figure 5.5, in each classification, three motions are needed to classify, which can be classified within a reasonable time. However, the classifier itself actually contains six motions, which allows the user to perform more motions without the needs of having many classifiers.

  This also benefits the training of Chinese martial arts as well. In Chinese martial arts, a simple form actually contains thirty to forty motions. For several forms within a classifier, it will have over hundreds of motions. Furthermore, there may have chain relationships between each motion.

Motion FSM is particular suitable in representing such a large amount of motions and allow the user to chain their motions.

- No more misclassifying of similar motions

  Misclassification can be quite often if two motions are similar. For example, drawing of number "6" is actually quite similar to drawing of number "0". If we want to include both motions in the same classifier without motion FSM, misclassification is likely to happen all the day. With motion FSM, we can actually separate the number "6" and number "0" to different state like Figure 4.5 such that the two numbers will never beginning classified in the same classification. In this way, we can include the two numbers and yet, avoid misclassifying the two numbers.

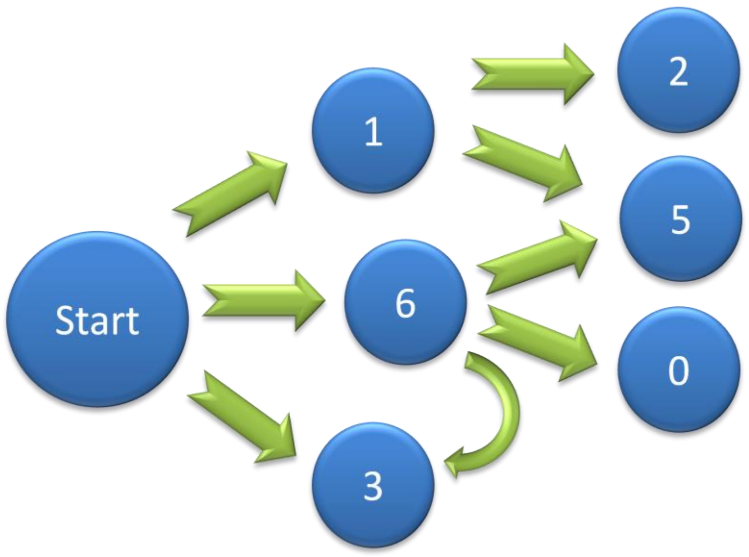


Fig. 5.5 Motion FSM with motions of drawing numbers

CHAPTER *6* – WII STYLE & DEVELOPMENT TOOL

In this chapter we are going to describe one of our deliverables in our project – the Wii Style, a C++ library built by us for other applications to use our motion classifier; and its associated development tool for users to record and build the motion classifier in a graphical user interface.

This chapter comprises the following sections:

6.1 INTRODUCTION

A motion classifier is unable to demonstrate its functionality if it cannot be manipulated easily, and cannot be used in other applications. So after building up our motion classifier in the first semester, we create a C++ library for calling our classifier so that it can be used in other applications. We also spent a lot of time in constructed a GUI for manipulation the motion classifier, for example, adding model motion data to it.

6.2 PREVIOUS WORK: DEVELOPED CONSOLE

In the last semester, we built our motion classifier together with a text-based console to test and demonstrate the power of our motion classifier. The menu of the developed console is shown below:

- Motion Database Operation
    - Register New Motion
    - Display Registered Motions
    - Edit Existing Motion
        - Display First Recorded Motion
        - Re-recorded First Motion
        - Display Second Recorded Motion
        - Re-recorded Second Motion
        - Display Third Recorded Motion
        - Re-recorded Third Motion
        - Display Other Recorded Motion
        - Re-recorded Other Motion
        - Re-recorded Other Motion
    - Unregister Motion

- Save Database
- Load Database
- Back to Main Menu

- Record Motions In Batch
  - Record Motion Sequentially
  - Back to Main Menu
- Classification
  - Benchmark Using Dynamic Time Warping
  - Real Time Using Dynamic Time Warping
  - Benchmark Using Dynamic Time Warping with Specific File
  - Back to Main Menu
- Exit Program

6.3 WII STYLE C++ LIBRARY

Although we have built a development console in the first semester, the development console along cannot do anything. Therefore *Wii Style* is developed. *Wii Style* is a C++ library for motion recording and classification. With *Wii Style*, our motion classifier can then be integrated to other applications by linking the library. Currently our *Wii Style* library relies on *WiiYourSelf!!*, another C++ Wii Remote library, to get connected with Wii Remote. Please refer to Appendix 1 for the API for Wii Style.

6.3.1 UML DIAGRAM

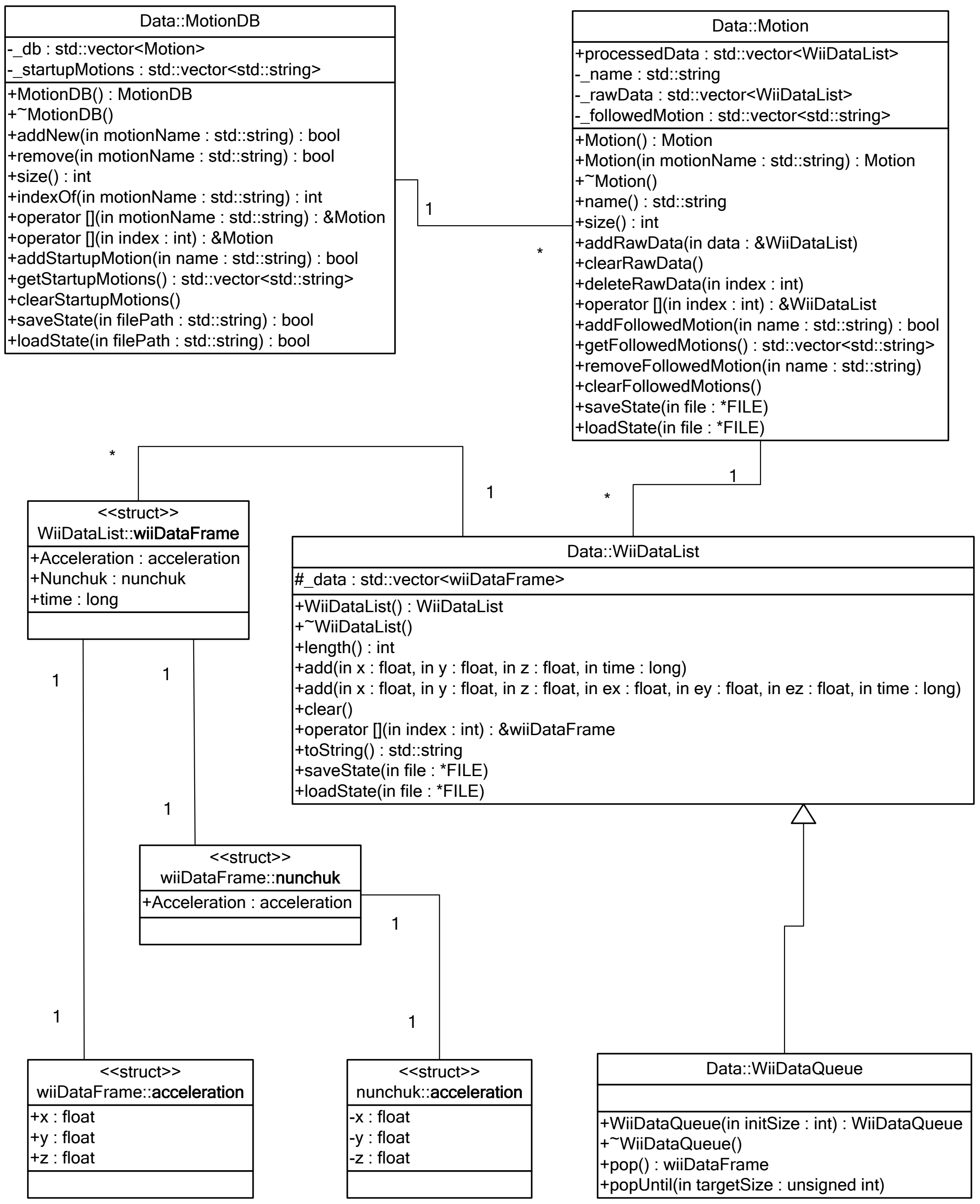

Data::MotionDB
-_db : std::vector<Motion>
-_startupMotions : std::vector<std::string>
+MotionDB() : MotionDB
+~MotionDB()
+addNew(in motionName : std::string) : bool
+remove(in motionName : std::string) : bool
+size() : int
+indexOf(in motionName : std::string) : int
+operator [](in motionName : std::string) : &Motion
+operator [](in index : int) : &Motion
+addStartupMotion(in name : std::string) : bool
+getStartupMotions() : std::vector<std::string>
+clearStartupMotions()
+saveState(in filePath : std::string) : bool
+loadState(in filePath : std::string) : bool
1
*
Data::Motion
+processedData : std::vector<WiiDataList>
-_name : std::string
-_rawData : std::vector<WiiDataList>
-_followedMotion : std::vector<std::string>
+Motion() : Motion
+Motion(in motionName : std::string) : Motion
+~Motion()
+name() : std::string
+size() : int
+addRawData(in data : &WiiDataList)
+clearRawData()
+deleteRawData(in index : int)
+operator [](in index : int) : &WiiDataList
+addFollowedMotion(in name : std::string) : bool
+getFollowedMotions() : std::vector<std::string>
+removeFollowedMotion(in name : std::string)
+clearFollowedMotions()
+saveState(in file : *FILE)
+loadState(in file : *FILE)
*
1
1
*
<<struct>>
WiiDataList::wiiDataFrame
+Acceleration : acceleration
+Nunchuk : nunchuk
+time : long
Data::WiiDataList
#_data : std::vector<wiiDataFrame>
+WiiDataList() : WiiDataList
+~WiiDataList()
+length() : int
+add(in x : float, in y : float, in z : float, in time : long)
+add(in x : float, in y : float, in z : float, in ex : float, in ey : float, in ez : float, in time : long)
+clear()
+operator [](in index : int) : &wiiDataFrame
+toString() : std::string
+saveState(in file : *FILE)
+loadState(in file : *FILE)
1
1
1
<<struct>>
wiiDataFrame::nunchuk
+Acceleration : acceleration
1
1
1
<<struct>>
wiiDataFrame::acceleration
+x : float
+y : float
+z : float
<<struct>>
nunchuk::acceleration
-x : float
-y : float
-z : float
Data::WiiDataQueue
+WiiDataQueue(in initSize : int) : WiiDataQueue
+~WiiDataQueue()
+pop() : wiiDataFrame
+popUntil(in targetSize : unsigned int)

Fig 6.1 UML diagram for namespace WiiStyle::Data of Wii Style

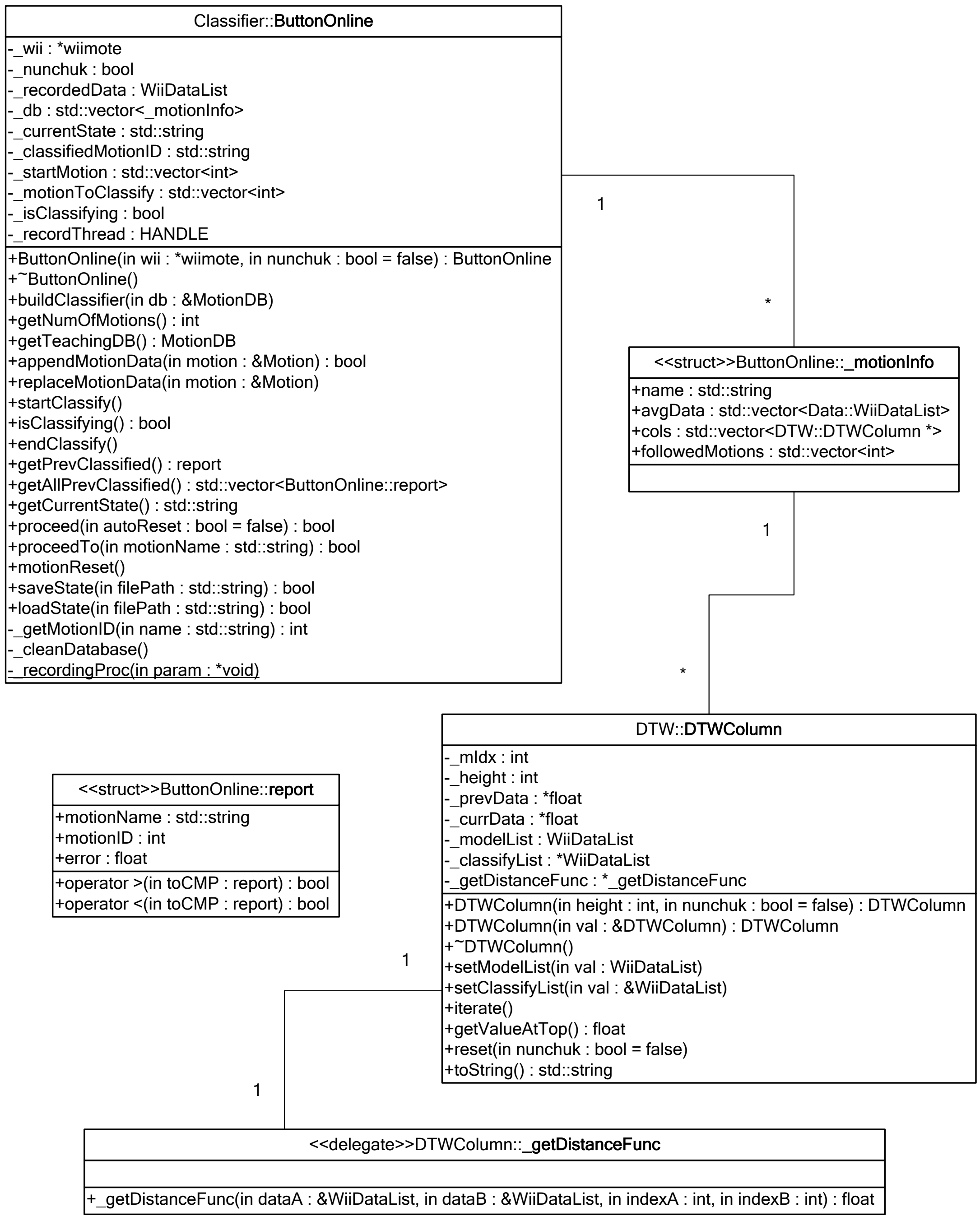

Classifier::ButtonOnline
-_wii : *wiimote
-_nunchuk : bool
-_recordedData : WiiDataList
-_db : std::vector<_motionInfo>
-_currentState : std::string
-_classifiedMotionID : std::string
-_startMotion : std::vector<int>
-_motionToClassify : std::vector<int>
-_isClassifying : bool
-_recordThread : HANDLE
+ButtonOnline(in wii : *wiimote, in nunchuk : bool = false) : ButtonOnline
+~ButtonOnline()
+buildClassifier(in db : &MotionDB)
+getNumOfMotions() : int
+getTeachingDB() : MotionDB
+appendMotionData(in motion : &Motion) : bool
+replaceMotionData(in motion : &Motion)
+startClassify()
+isClassifying() : bool
+endClassify()
+getPrevClassified() : report
+getAllPrevClassified() : std::vector<ButtonOnline::report>
+getCurrentState() : std::string
+proceed(in autoReset : bool = false) : bool
+proceedTo(in motionName : std::string) : bool
+motionReset()
+saveState(in filePath : std::string) : bool
+loadState(in filePath : std::string) : bool
-_getMotionID(in name : std::string) : int
-_cleanDatabase()
-_recordingProc(in param : *void)
1
*
<<struct>>ButtonOnline::_motionInfo
+name : std::string
+avgData : std::vector<Data::WiiDataList>
+cols : std::vector<DTW::DTWColumn *>
+followedMotions : std::vector<int>
1
*
DTW::DTWColumn
-_mIdx : int
-_height : int
-_prevData : *float
-_currData : *float
-_modelList : WiiDataList
-_classifyList : *WiiDataList
-_getDistanceFunc : *_getDistanceFunc
+DTWColumn(in height : int, in nunchuk : bool = false) : DTWColumn
+DTWColumn(in val : &DTWColumn) : DTWColumn
+~DTWColumn()
+setModelList(in val : WiiDataList)
+setClassifyList(in val : &WiiDataList)
+iterate()
+getValueAtTop() : float
+reset(in nunchuk : bool = false)
+toString() : std::string
<<struct>>ButtonOnline::report
+motionName : std::string
+motionID : int
+error : float
+operator >(in toCMP : report) : bool
+operator <(in toCMP : report) : bool
1
1
<<delegate>>DTWColumn::_getDistanceFunc
+_getDistanceFunc(in dataA : &WiiDataList, in dataB : &WiiDataList, in indexA : int, in indexB : int) : float

Fig 6.2 UML diagram for namespace WiiStyle::Classifier and WiiStyle::Classifier::DTW of Wii Style

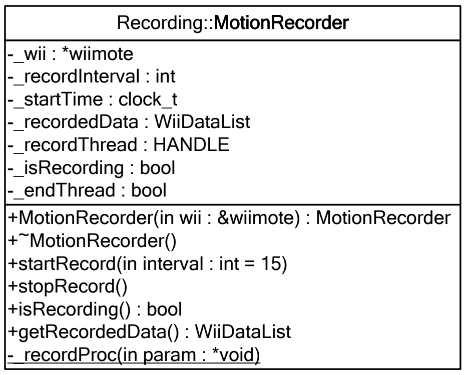
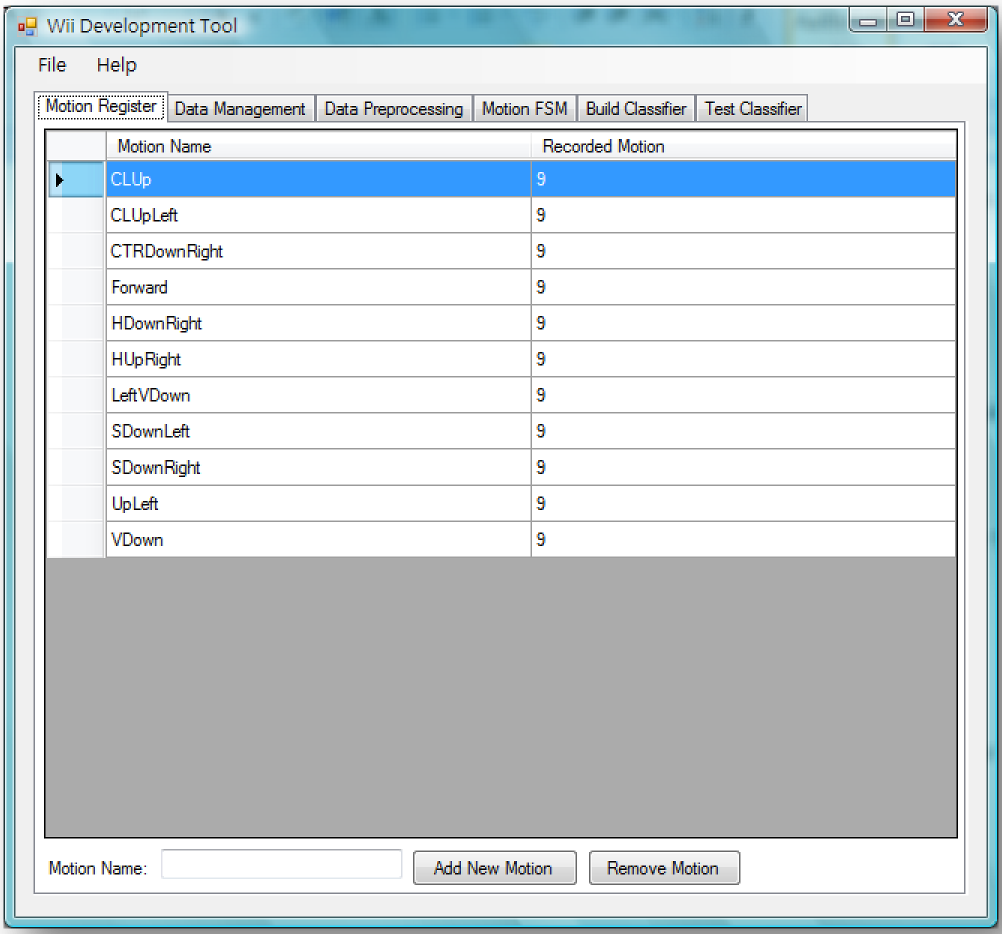

Recording::MotionRecorder
-_wii : *wiimote
-_recordInterval : int
-_startTime : clock_t
-_recordedData : WiiDataList
-_recordThread : HANDLE
-_isRecording : bool
-_endThread : bool
+MotionRecorder(in wii : &wiimote) : MotionRecorder
+~MotionRecorder()
+startRecord(in interval : int = 15)
+stopRecord()
+isRecording() : bool
+getRecordedData() : WiiDataList
-_recordProc(in param : *void)

Fig 6.3 UML diagram for namespace WiiStyle::Recording of Wii Style

6.4 WII STYLE DEVELOPMENT TOOL

In order to let developers collect and manipulate the model motion data for the classifier, we developed a GUI development tool for Wii Style.



Motion Name	Recorded Motion
CLUp	9
CLUpLeft	9
CTRDownRight	9
Forward	9
HDownRight	9
HUpRight	9
LeftVDown	9
SDownLeft	9
SDownRight	9
UpLeft	9
VDown	9

Fig. 6.4 Screenshot of Wii Style Development Tool

Figure 6.4 shows the screenshot of the Wii Style Development Tool. As shown in the screenshot, the tool comprises the following components:

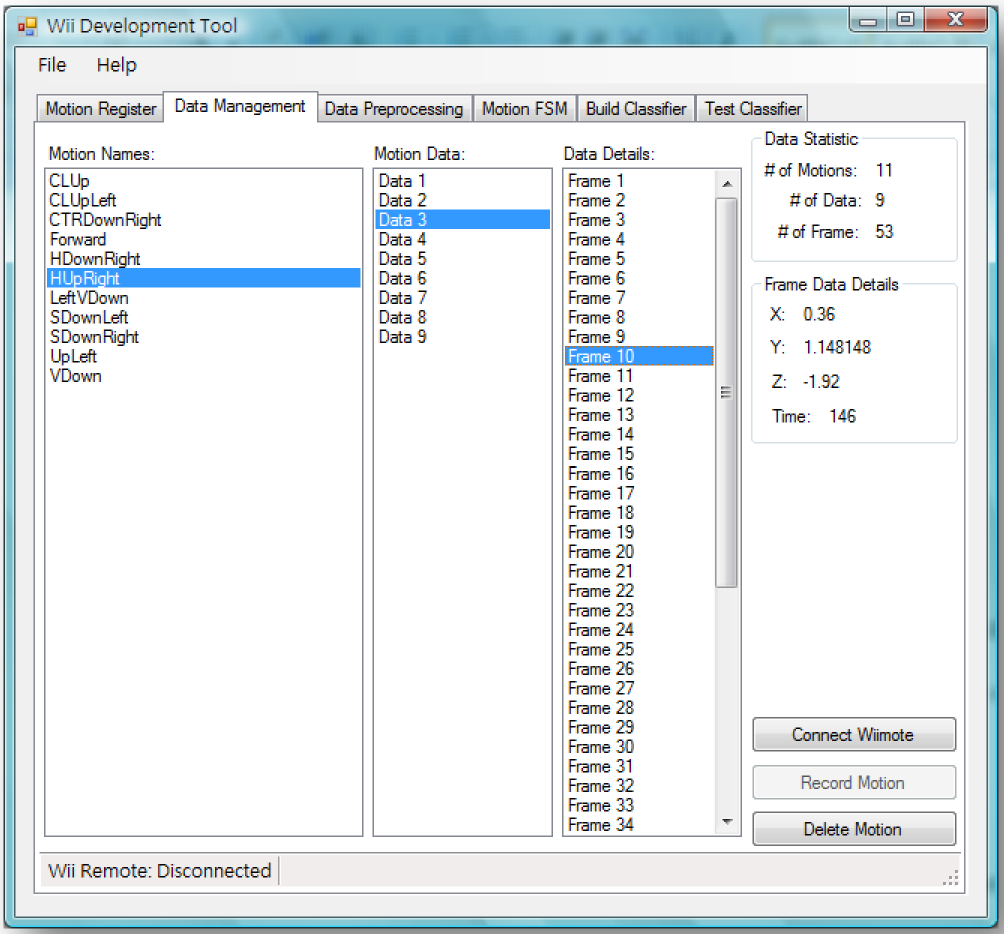
- Motion Register
- Data Management
- Data Preprocessing
- Motion FSM
- Build Classifier
- Test Classifier

In the following sections we will introduce the six components one by one.

6.4.1 MOTION REGISTER

In motion register, developers can add new motions, or remove certain motions from the motion database. As shown in Figure 6.4 in the previous page, there are 11 different motions in this motion database; and for each type of motion in the motion database, there are nine sets of recorded motion data.

6.4.2 DATA MANAGEMENT



Motion Names: CLUp, CLUpLeft, CTRDownRight, Forward, HDownRight, HUpRight, LeftVDown, SDownLeft, SDownRight, UpLeft, VDown

Motion Data: Data 1, Data 2, Data 3, Data 4, Data 5, Data 6, Data 7, Data 8, Data 9

Data Details: Frame 1, Frame 2, Frame 3, Frame 4, Frame 5, Frame 6, Frame 7, Frame 8, Frame 9, Frame 10, Frame 11, Frame 12, Frame 13, Frame 14, Frame 15, Frame 16, Frame 17, Frame 18, Frame 19, Frame 20, Frame 21, Frame 22, Frame 23, Frame 24, Frame 25, Frame 26, Frame 27, Frame 28, Frame 29, Frame 30, Frame 31, Frame 32, Frame 33, Frame 34

Data Statistic
# of Motions: 11
# of Data: 9
# of Frame: 53

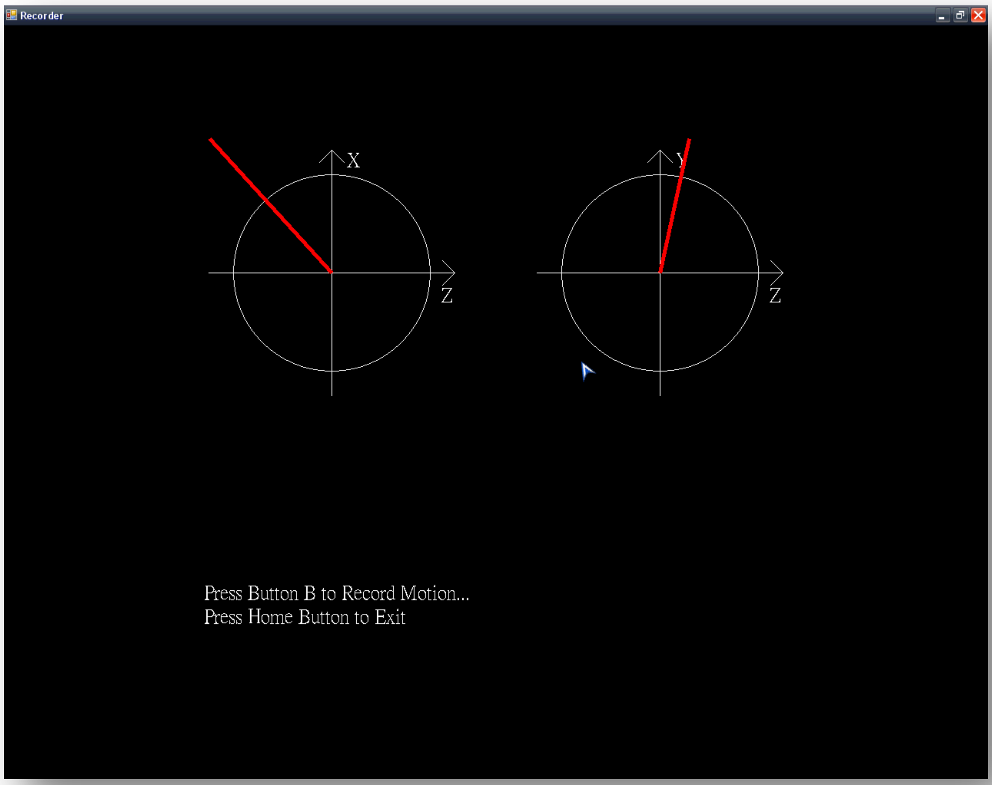
Frame Data Details
X: 0.36
Y: 1.148148
Z: -1.92
Time: 146

Wii Remote: Disconnected

Fig 6.5 The data management module

Figure 6.5 above shows the data management module. This module allows developers to manipulate motion data recorded for all the motions.

To record new motion data for a motion, developers can press “Connect Wiimote” to connect Wii Remote to the computer first. After it is connected, motion recording can be started by first selecting the name of motion and follow by pressing “Record Motion” button. Another screen in Figure 6.6 will be shown. Once the button on Wii Remote is pressed, then motion data will be captured; the capturing will end once the button on Wii Remote is released.



Press Button B to Record Motion...
Press Home Button to Exit

Fig 6.6 Screenshot for recording motions

The recorded motion data is then available and is shown at "Motion Data" column. Developers may double click a particular motion data to see a graphical representation of the data, which is shown in Figure 6.7.

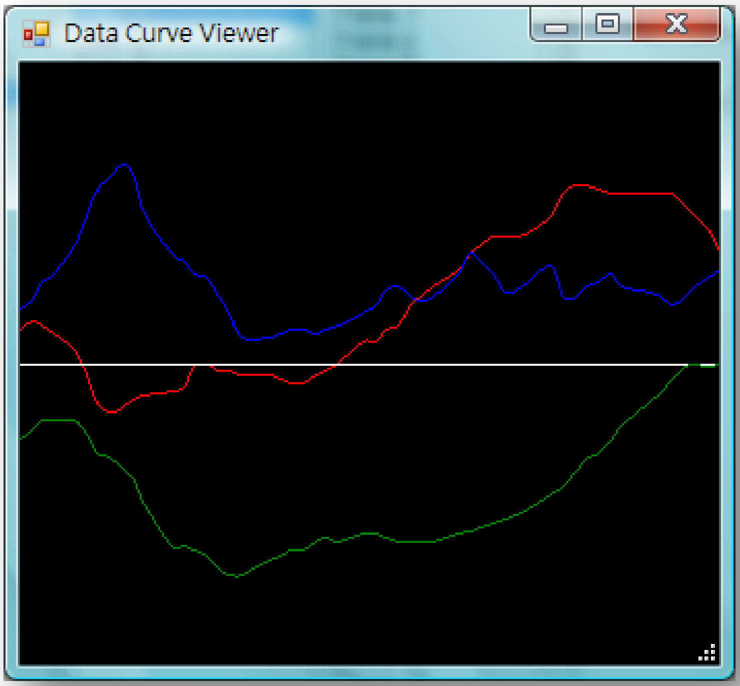


Fig 6.7 Data Curve Viewer – A graphical representation of motion data. The red curve, blue curve and green curve represent acceleration values of X-axis, Y-axis and Z-axis, respectively. The vertical axis is the acceleration while the horizontal axis is the time.

The column "Data Details" keeps the acceleration values for all the frames throughout a motion. As shown in Figure 6.5, there are 53 frames for this motion data, in which one frame is about 14 – 15 milliseconds, as mentioned in Chapter 4.2. At that instant, the accelerations values of X-axis, Y-axis and Z-axis are of 0.36, 1.148148 and -1.92 respectively.

6.4.3 DATA PREPROCESSING

To increase classification accuracy, Wii Style development tool also provides a "Data Preprocessing Module".

In the module, developers are supposed to preprocess and tune their recorded motion data in the database. This is to increase the quality of the motion data by using the functions provided in this module such as noise filtering. The preprocessed or tuned motion data will then be stored as processed data in the database for building the classifier later.

However, due to the time limitation, we are not able to implement the module as described above. Currently, only auto data preprocessing is available. The auto data preprocessing will take the every three motion data in the database and calculate the average of the three motion data and store it as processed data in the database.

6.4.4 MOTION FSM

In Section 5.8, we mentioned that our motion classifier supports Motion Finite State Machine (Motion FSM). An interface is constructed in the development tool to set up the motion FSM. Figure 6.8 below shows the interface of setting up the motion FSM.

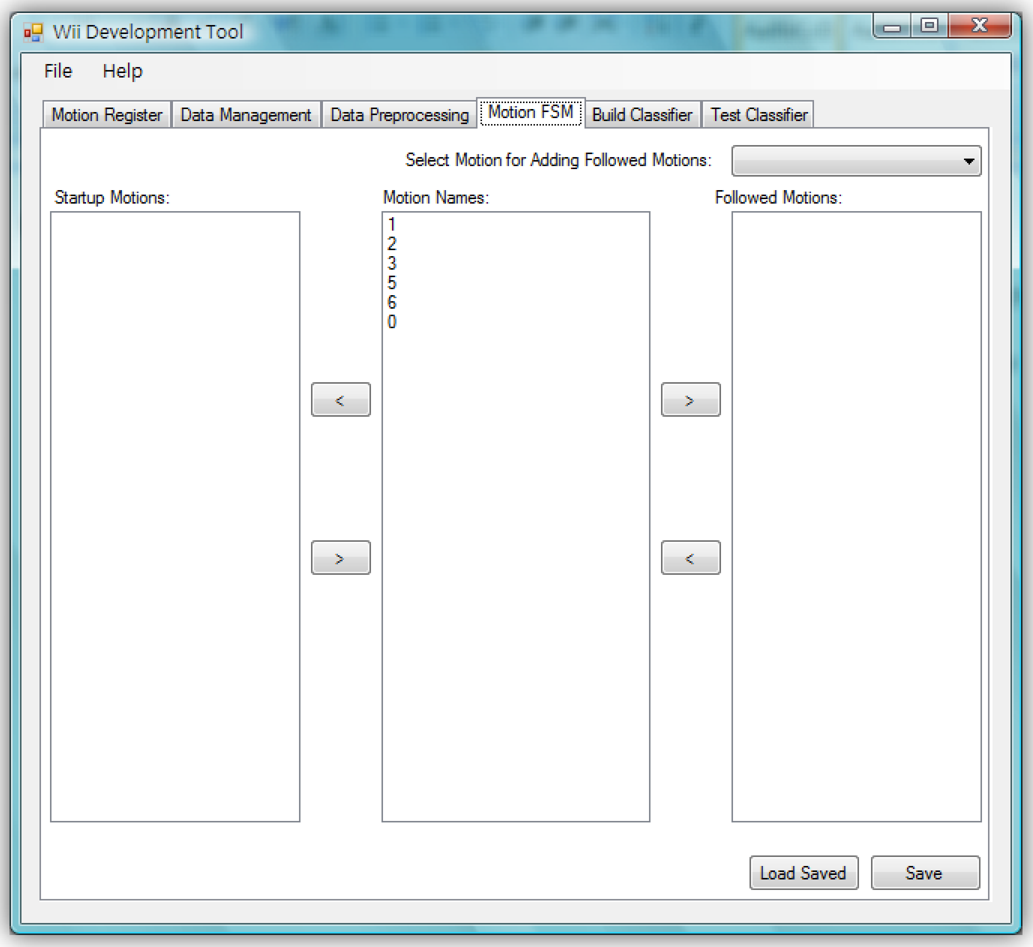


Fig 6.8 The Motion FSM Module

The interface consists of three list box in columns – Startup Motions, Motion Names, and Followed Motions.

- "Startup Motions", as its names tells, states the type of motion that can be performed at the beginning (start state)
- "Motion Names" lists out all the available motions in the motion database

- “Followed Motions” lists out the motions that can be followed after a particular motion is done. The content in this column depends on the type of motion that is selected in the combo box above the list box in the column.

Recall Figure 5.5 in which a simple FSM on drawing numbers is drawn. Here we try to build up that FSM in our motion classifier as an example to illustrate how the interface is used.

At the start state, players are allowed to draw either 1, 3 or 6, and so these three motions should be put into “Startup Motions”, by first selecting the motion, and press the "<" button next to the column of “Startup Motions”. The result is shown in Figure 6.9.

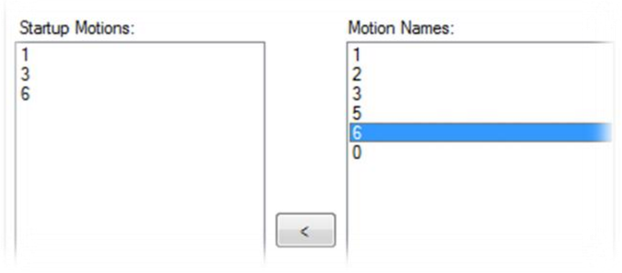
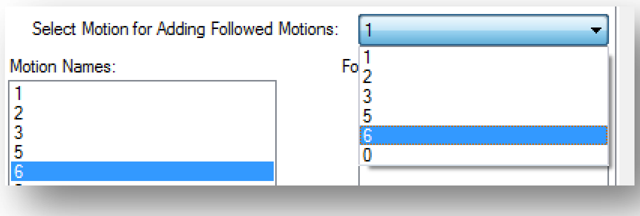


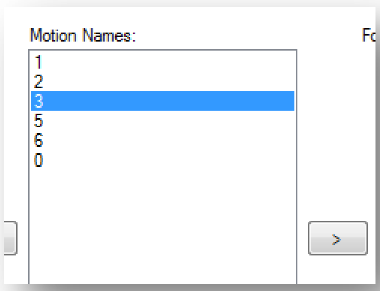
Fig 6.9 A screenshot of Motion FSM interface after adding startup motions

Next is to add the followed motions. For example, after drawing a 6, players are allowed to draw 3, 5 or 0. To add these motions, developers should first choose 6 from the pull down menu in the combo box above, which is shown in Figure 6.10(a); the next step is to select the motions within “Motion Names”, which is shown in Figure 6.10(b), and press ">" to add it to “Followed Motions”, which is shown in Figure 6.10(c).

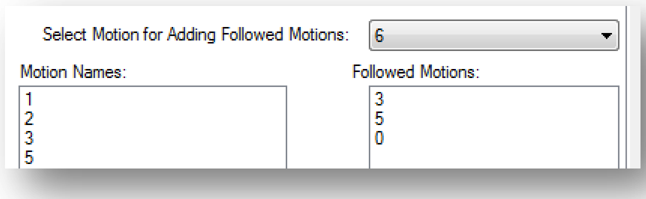
Just repeat the above step to add followed motions for other motions. Save the FSM once completed everything.



(a) Select the motion for which followed motion is needed



(b) Select the followed motion and press ">"



(c) The followed motion is then added

Fig 6.10 How developers can add followed motions for a motion in FSM in three steps

6.4.5 BUILD CLASSIFIER

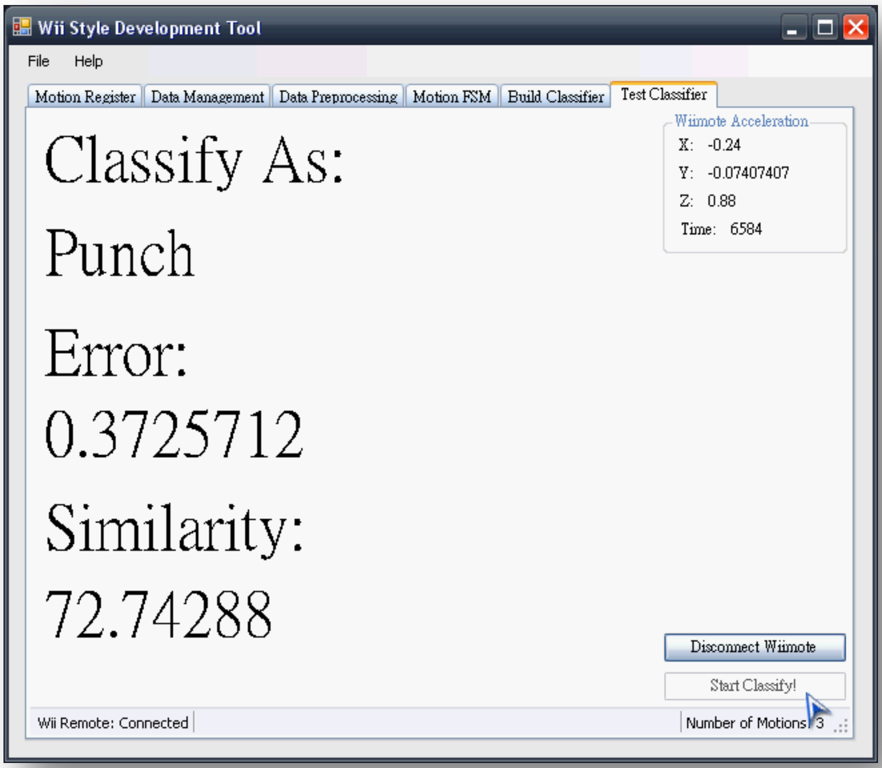
After obtained model motion data as well as prepared the motion FSM, the classifier is ready to be built in “Build Classifier Module”.

In this module, developers are supposed to decide which motion data are used as model motion data in the classifier. Model motion data can be chose from unprocessed motion data or processed motion data. The developers can first use the motion data suggested by this module as the model motion data, and then modify it accordingly to increase the classification accuracy.

However, due to the time limitation, we are not able to implement the module as described above. Currently, only auto build is available. The auto build will take all processed motion data in the database as the model motion data.

6.4.6 TEST CLASSIFIER

This module is used for developers to test the classifier that is built previously. To test the classifier, developers just have to connect Wii Remote to the computer and perform motion themselves to test it. The classification result is then shown with the DTW error between the input motion data and the model motion data and the similarity which is derived from the DTW error. This helps developers to evaluate the performance and accuracy of the classifier so that they may consider modifying some model motion data to guarantee a better performance. Fig 6.11 shows a screenshot of the classification result inside “Test Classifier” Module.



Classify As:
Punch
Error:
0.3725712
Similarity:
72.74288

X: -0.24
Y: -0.07407407
Z: 0.88
Time: 6584

Wii Remote: Connected
Number of Motions: 3

Fig 6.11 Screenshot of "Test Classifier" Module

6.4.7 SAVING MOTION DATABASE AND CLASSIFIER

To let the developer continue working on recording motion data in the motion database in the future, and to let other applications use the motion classifier, developers should save the motion database once finished recording, and save the classifier once they ready to be used. By clicking, "File" in the menu, and developers can find options of saving motion database and classifier, respectively. The saved classifier can then be used by Wii Style classifier in C++ library.

CHAPTER 7 – THE DEMO

To demonstrate the power of our motion classifier interactively to the players, undoubtedly computer game is always the best choice among various types of applications.

In this chapter we are going to describe some other deliverables in our project, which is an existing game integrated with Wii Remote and our motion classifier, as well as a mini-game that showing the features of our motion classifier.

This chapter comprises the following sections:

7.1 ONIMUSHA 3

Onimusha 3: Demon Siege is an action adventure game developed by CAPCOM and Ubisoft. The game consists of different scenes; player characters have to use the accompanied weapons to attack enemies in order to go through the game.

Fig. 7.1 Screenshots of *Onimusha 3: Demon Siege*

7.1.1 MOTIVATION

Dao, one of the major 18 major weapons in traditional Chinese martial arts, is available as a weapon for the character in most of the action game. Comparatively speaking, dao moves is easier to be mastered by beginner to Chinese martial arts. On the other hand, combat dao motions are performed by the character, which is common in traditional Chinese martial arts, too.

However, for all the computer games featuring dao so far, people can just control the game character by keyboards or gamepads. Even if they want the game character to perform certain combat motions, they have no other choices but to press several keys in a particular order. Obviously this is not a realistic approach, and also the number of motions that can be performed is limited by the keys that are available.

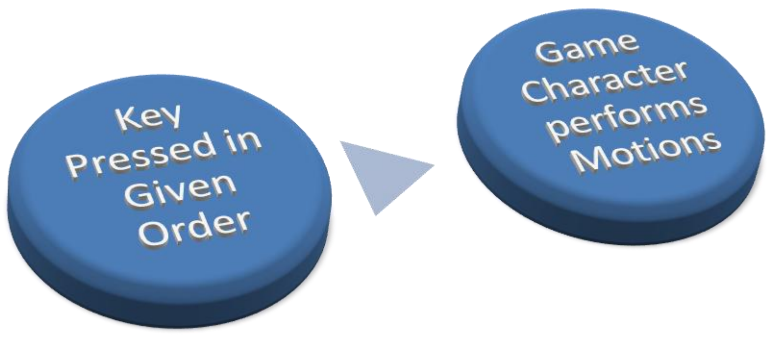


Fig 7.2 The current way that we can control the game character to perform motions

However, with Wii Remote, together with our motion classifier, if people want the game character to perform certain motions, they can simply perform the motion themselves so that the game character performs the same motion inside the game. Undoubtedly this is an advantage over the traditional keyboard and gamepad approach.

With this advantage, together with the fact that Onimusha 3 features dao, which is relatively easy to master for beginners to Chinese martial arts; that's the reason why we have chosen to integrate our motion classifier, which features different moves of Chinese martial arts, together with Wii Remote, into Onimusha 3. This is done on an existing game to guarantee a good game play, and at the same time, to bring a new approach in playing the game for a more realistic feeling.

7.1.2 TEACHING THE CLASSIFIER

One may ask the following questions – Different users perform motion differently, how can it be recognized accurately? Yes, in order to have the motions be classified correctly, the classifier has to be taught to recognize our motions in advance.

7.1.2.1 WHY THE CLASSIFIER SHOULD BE TAUGHT?

Teaching classifier is necessary because our motion classifier classifies motions based on the similarity between the performed motion and the model motion stored in the classifier. Thus, different users may have different classification result when performing the same motion.

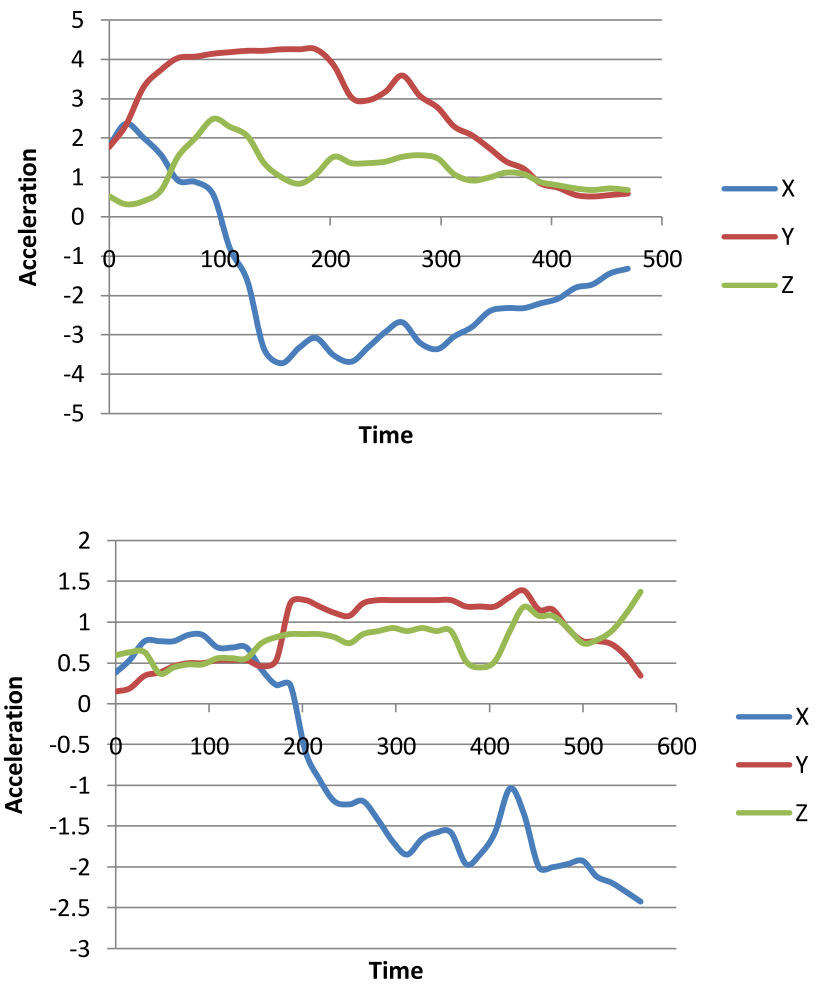


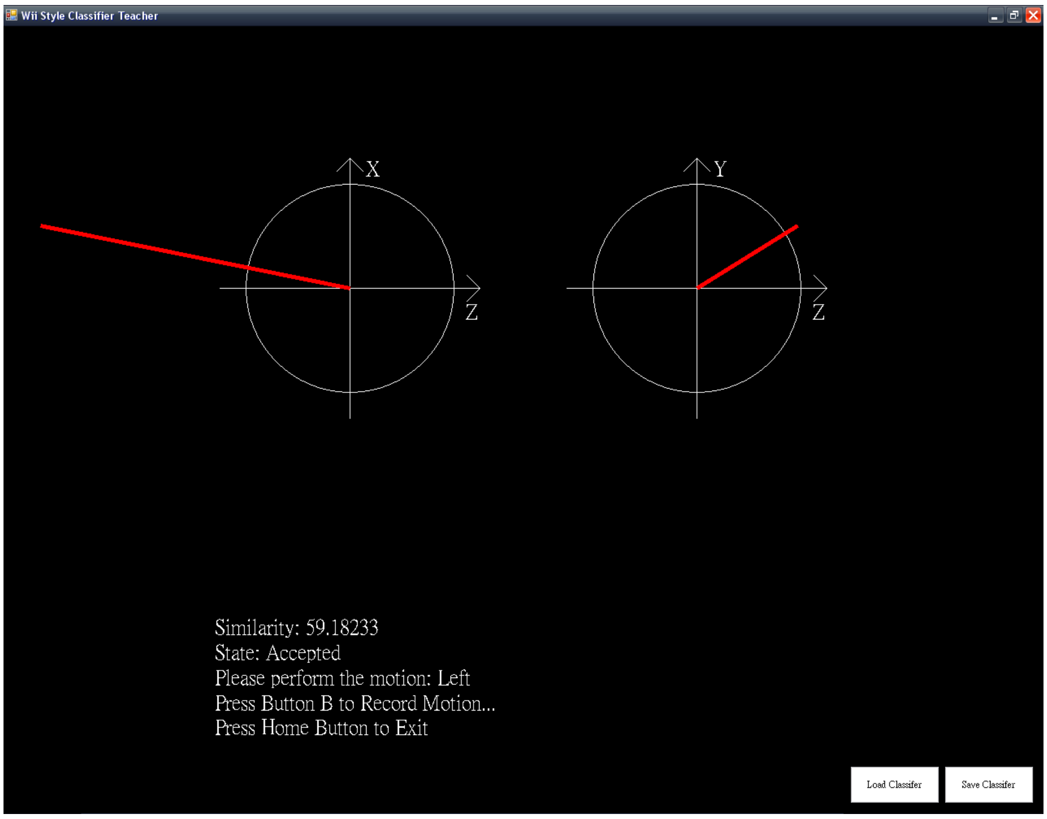
Fig. 7.3 A plot of values against time on a dao motion performed by two different users

As illustrated by Figure 7.3, although the dao motion is the same, but different data are obtained for different users; there are chances that these two motions are quite different from the model motion, and hence they are classified wrongly.

In our demo, *Onimusha 3*, the main purpose is to allow player to play and enjoy the realistic feeling of attacking enemies by performing dao motions using the Wii Remote. It is not essentially teaching the player the Chinese dao motions. Definitely we can't let players feel frustrated when they tried their very best to perform the motions but none is recognized by the classifier. They are playing, but not learning! Therefore players' motion data is used instead in this demo.

7.1.2.2 HOW TO TEACH THE CLASSIFIER?

Players can create their profile which contains their own model motion data by teaching the classifier. To teach the classifier, developer has to implement and integrate their own classifier teacher. As a demo, we have implemented "*Wii Style Classifier Teacher*" a generic classifier teacher that to show how the process of teaching classifier works.



Similarity: 59.18233
State: Accepted
Please perform the motion: Left
Press Button B to Record Motion...
Press Home Button to Exit

Fig. 7.4 Screenshot for Wii Style Classifier Teacher – the interface for teaching the classifier

In Wii Style Classifier Teacher, it comprises the followings:

- Two 2-dimensionals coordinate planes. The left and right planes show the real-time acceleration values on X-Z plane and Y-Z plane, respectively.
- A status box which has the following fields:
  - Similarity – Shows the similarity of performed motion when compared to the model motion data. The maximum similarity is 100.
  - State – Shows the status of the last performed motion, whether it is:
    - Accepted as one of the model motion data.
    - Rejected since it is not similar.
  - "Please perform the motion" – Indicates the next motion that should be performed by the player.
- Two buttons for different operations:
  - Load classifier – Load the classifier that is going to teach by the player.
  - Save classifier – Save the taught classifier.

Before launching classifier teacher, developer should first prepare a tutorial for the player to let them know what kind of motions they can perform. This step is omitted in our classifier teacher.

In the classifier teacher, players have to follow the instruction in classifier teacher to perform the required motions so to prepare their own model motion data.

For example, in our "*Wii Style Classifier Teacher*", after the program starts, players are promoted to perform a motion. Once the players performed the motion, the motion data is kept at the same time.

The motion data is then passed into motion classifier for comparison with the corresponding model motion data. Similarity is computed based on the comparison result, which is then displayed on the screen.

If the similarity reaches a predefined threshold value, the motion just performed is accepted; otherwise the motion is rejected. For accepted motions, the corresponding motion data is stored as one of the model motion data for players. The state is also displayed on the screen to notify players.

After the result is displayed on the screen, players are prompted to perform the same motion again until three accepted motion data is performed. The above procedures are then repeated until all motions in the classifier contain three accepted motion data respectively. Several motion data are collected to increase the classification accuracy, which is proved to be successful in section 5.4.3.1.

Once all the different motions are recorded, then players' model motion data is completed. After the new classifier is saved, it can be used to play the game.

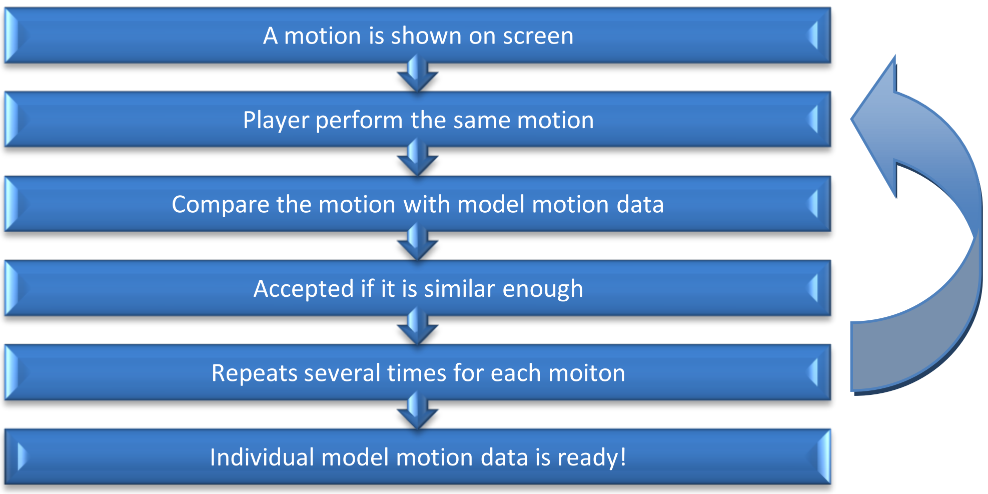

A motion is shown on screen
Player perform the same motion
Compare the motion with model motion data
Accepted if it is similar enough
Repeats several times for each moiton
Individual model motion data is ready!

Fig. 7.5 Steps showing how users can create individual model motion data

7.1.3 OUR IMPLEMENTATION

Before *Onimusha 3* is integrated with our classifier, the character inside the game can only be controlled by keyboards or gamepads; for example, if we want the game character to perform certain dao motions, users have to press several keys so that the game character performs the desired moves.

To integrate the motion classifier to *Onimusha 3* without changing the game source code, this can be done by simulating key presses on keyboard. Before the actual game starts, we can launch the program which has a built-in key simulator and motion classifier, and also maintains connection with the Wii Remote. In the game, we can control the character by using the Wii Remote; for example, we can press the forward button on Wii Remote to have the character moving forward.

Since we are not able to change the source code of *Onimusha 3*, therefore the set of motions we used in our classifier is constrained by motions used in the game. Figure 7.6 shows the motion FSM for *Onimusha 3.*

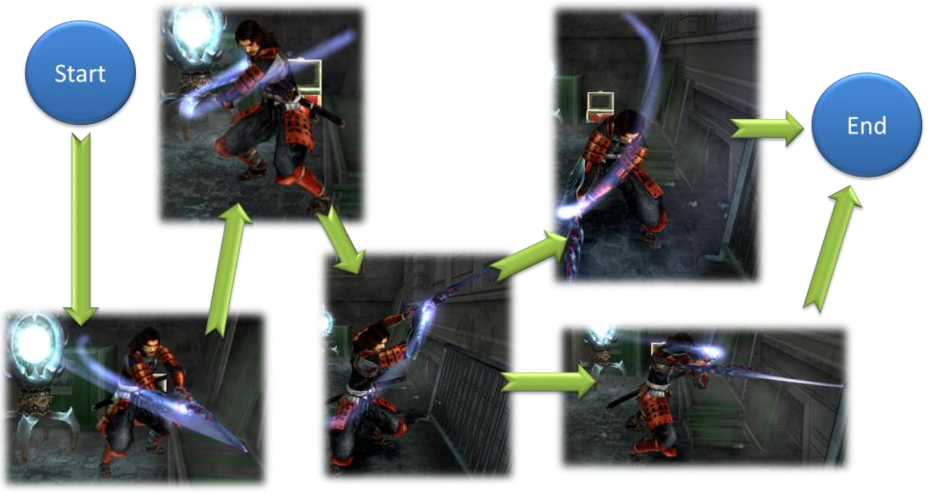


Fig. 7.6 Motion FSM for *Onimusha 3*

In *Onimusha 3*, state transitions involve a time out limit. That is, after the player performed one motion, if the player is not able to perform the next motion before time out, the player has to start performing the motion all over again from the start state. Therefore, besides just integrating our classifier into the game, we also need to implement the time out system so that motion performed too late will result in no key pressing.

During the game, if the player wants the game character to perform a dao moves, the player can simply hold the Wii Remote, press the button, perform the dao moves, and release it when finished. The motion is recorded and then passed to the motion classifier. Classification is done on-the-fly while the player is performing the motion. Motion classifier classifies the motion as certain move after the motion is finished; based on the classification result and the similarity, we can make use of key simulator to simulate certain key pressed on the keyboard, and so the game character performs the motion inside the same.

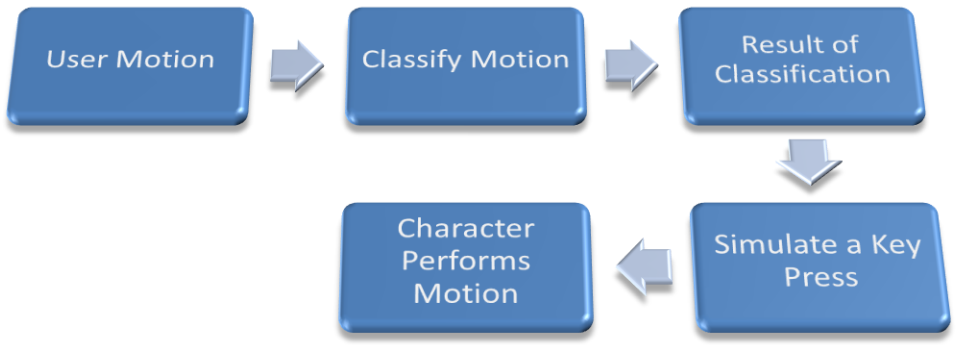


Fig. 7.7 The work flow for the game character to perform dao motion after integrating our classifier into the game

7.2 DEMO OPTIMIZED WITH OUR CLASSIFIER

Apart from integrating our classifier into *Onimusha 3*, we have also constructed another demo, which optimized with our classifier.

7.2.1 MOTIVATION

It is well-known that Chinese martial arts are not easy to master. To get familiar with it, the best method is to perform the same moves correctly and repeatedly under guidance of some experts. However, it is not easy to have an expert always stay with you to guide the training.

Our motion classifier does not only classify the type of motion, but can also tell users how similar the motion is when compared to model motion. This certainly provides a learning opportunity for everyone to learn Chinese martial arts on their own by knowing how similar their motions are.

Onimusha 3 failed to serve as a learning tool kit of Chinese martial arts due to two reasons:

1. *Onimusha 3* failed to show the similarity of motions as an effect in the game. It can only demonstrate motion classification – in terms of correct or incorrect classification result. Some users may have the motion be classified correctly just by chance, and they don't know what should be the correct way of performing the moves.

2. *Onimusha 3* is originally designed for users to play and enjoy the game using keyboard or gamepad. The design does not fit for learning Chinese martial arts using Wii Remote.

That's the reason why we have to build this demo. We wish to use this demo to demonstrate how our classifier tells the similarity between performed motions and model motion, and serves as a beginner learning kit for users to learn Chinese martial arts themselves.

7.2.2 TOOLS USED

When building up the demo, we have used the following applications and libraries:

- Irrlicht Graphic Engine
- Freetype Library
- WiiYourSelf!, Wii Style and Wii Style Development Tool
- 3D Studio Max and MilkShape

We have also investigated on the use of motion capturing system to prepare animations for the character.

7.2.2.1 IRRLICHT ENGINE

The Irrlicht engine[18] is a cross-platform 3D graphics rendering engine written in C++. It is a powerful API and contains the following features to enable us to build the 3D game demo easily:

- Direct import of image files for applying on different models;
- Direct import of common mesh file formats so that external models can be used;
- Easy, fast and simple collision detection and response;
- Supports character animation with morph target animation;

7.2.2.2 FREETYPE LIBRARY

The FreeType library[19] is a software font engine to produce high quality images from TrueType fonts. In Irrlicht, bitmap fonts are used by default, which is not large and clear enough to be seen. With add-ons made by zgock[20] and Cristian[21] from Irrlicht Engine Forum, TrueType fonts can be displayed through Irrlicht to produce a better looking.

7.2.2.3 WIIYOURSELF!, WII STYLE AND WII STYLE DEVELOPMENT TOOL

WiiYourSelf![5], Wii Style and Wii Style Development Tool. WiiYourSelf is needed to connect Wii Remote with the computer, and obtain data from accelerometers of Wii Remote; Wii Style is used to classify users' motion; while Wii Style Development Tool is also used to record model motion data to build the classifier.

7.2.2.4 3D STUDIO MAX AND MILKSHAPE

3D Studio Max and MilkShape are mainly dealing with the character mesh and animations in the demo. First of all, the character is obtained from a website Turbo Squid[22]. It is then edited with 3D Studio Max, and exported into Steam Source SDK model source file (file extension: smd). MilkShape is used to convert the Steam Source SDK model source file into Blitz3D mesh file (file extension: b3d). Finally, the Blitz3D mesh file can be loaded directly by the Irrlicht engine. To animate the character in the demo, we have used 3D Studio Max create different animations for the character.

7.2.2.5 MOTIONSTAR WIRELESS® 2

During the time when demo was constructed, we have considered the use of motion capturing system to prepare animations for models. We thank our department for lending us the MotionStar Wireless® 2[23].

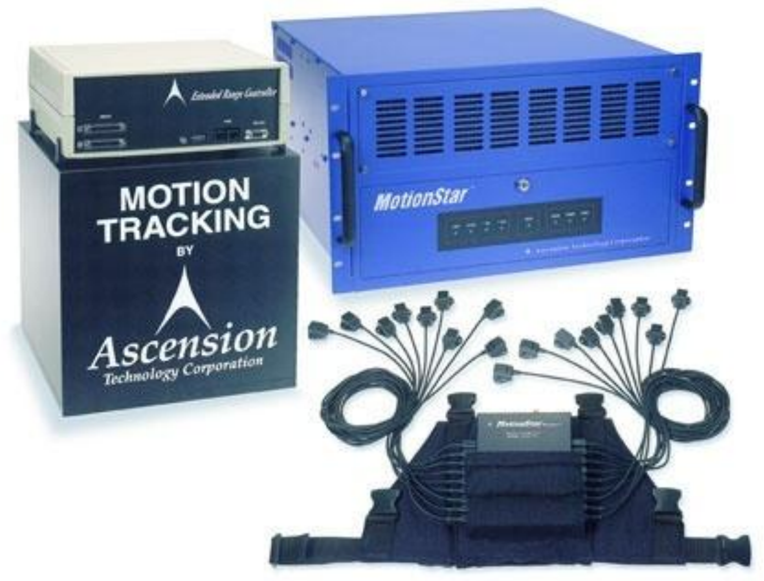


Fig 7.8 The MotionStar Wireless® 2

MotionStar Wireless® 2 is a real-time magnetic motion capturing system with 20 wireless sensors. Each sensor is attached to different body joints and delivers a full six degrees-of-freedom (including X, Y, Z, Yaw, Pitch, Roll) solution every second under the presence of a magnetic field. The solution obtained can be used to generate 3D animated models, in which the animations are more realistic.

During our trials, we noted that the system can collect data from sensors; but unfortunately, the obtained data cannot be passed to other computers for further processing, therefore we have given up using this system and used 3D Studio Max to prepare animations instead.

7.2.3 THE GAME PLAY

Unlike the previous mentioned *Onimusha 3*, which is designed for users to play; this demo is designed for users to learn different moves of Chinese martial arts, and so, the classifier should not be taught with the users' motion data.

The game begins in scene where there is a moving enemy with certain health points, and a controllable character. The goal is to attack the enemy until its health point drops to zero. To attack the enemy, we can perform motion ourselves with Wii Remote, which is then classified as certain type of motion, and eventually performed by the game character.

Once the attack to the enemy is successful, a score is the displayed above the head of the enemy. The score in red color is actually the drop in the health point to the enemy, with 100 as the maximum. It is computed based on the similarity of the performed motion – more similar the motion, higher the score. There is another blue score which indicate the remaining health points of the enemy. If the health point is below zero, the enemy is disappeared and regenerated at another position.

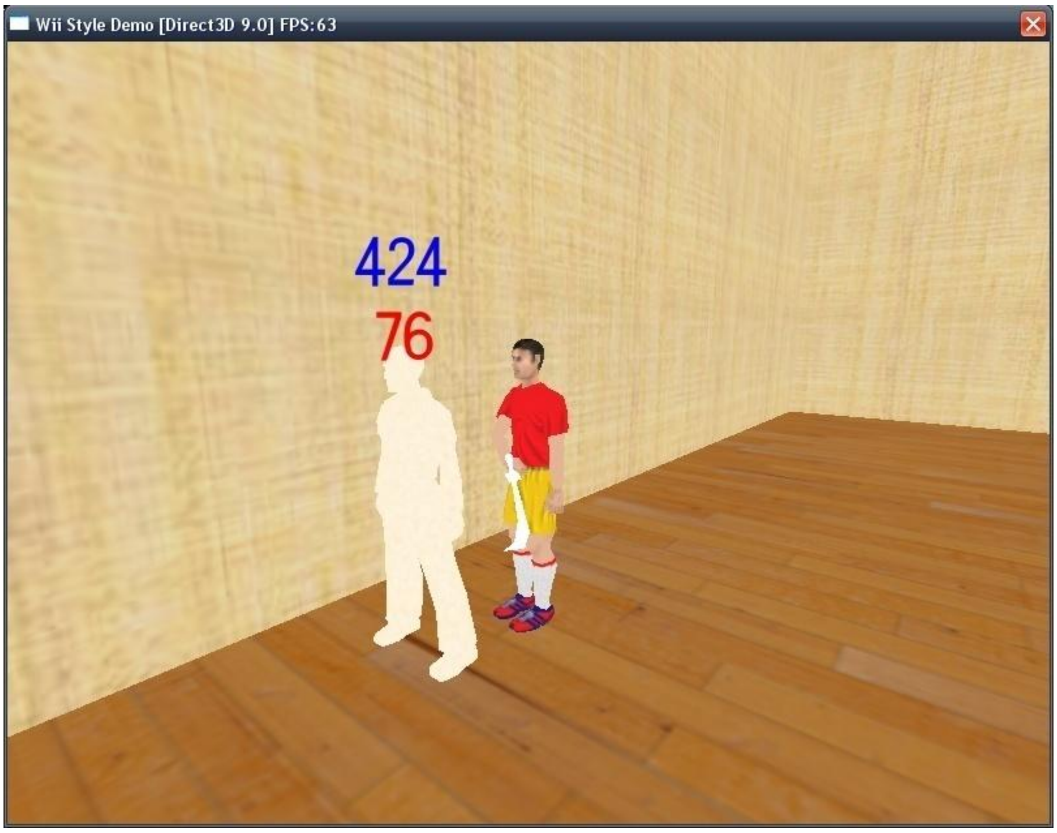


Fig 7.9 Screenshot of Wii Style Demo – demo that utilizes our motion classifier. The blue score corresponds to the current health point of the enemy; while red score is the damage to the enemy for the player’s last performed motion.

There are two modes to play the demo – the free-fight mode, and the series mode.

7.2.3.1 FREE-FIGHT MODE

Under free-fight mode, users may perform any motion in any orders, provided that the motion is known by the classifier. This mode is designed for beginners to master basic moves of Chinese martial arts before learning advanced things.

7.2.3.2 SERIES MODE

It is well-known that Chinese martial arts feature “forms”, which is series of predefined moves. After mastering the basic moves, beginners may go further to learn “form” by performing sequences of combat moves. Finally, they may go even further in learning “application” by performing motions in other orders so as to utilize the technique they learnt.

This mode is implemented making use of motion finite state machine provided by Wii Style, so that users can perform motions in a predefined sequence.

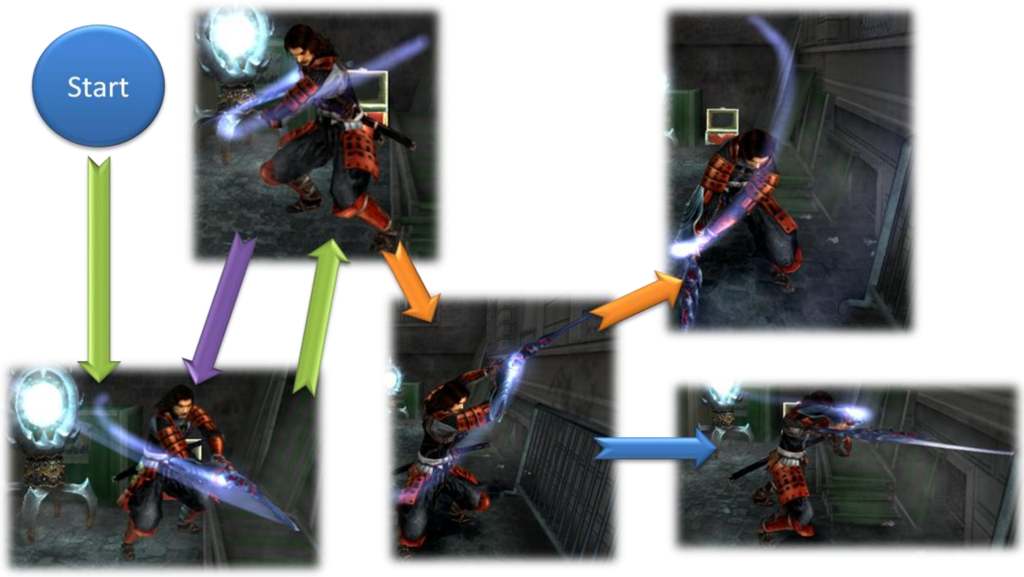


Fig 7.10 An illustration of learning Chinese martial arts under series mode

Figure 7.10 shows the motion FSM that is used for learning Chinese martial arts under series mode.

In the first stage, users are only allowed to perform the first two moves, which are indicated by the green arrows. The game would calculate the similarity of the performed motion so that users know whether their motion is doing correctly or not. The similarity is also recorded in the game, so that when the users perform the moves more and more accurate, it will be the right time to let to user to learn more by going to second stage.

In the second stage, two more motions are unlocked for the user to learn, as indicated by the orange arrows. At this stage, a complete “form” is available for the user. Users should be able to learn the “form” by that time. The same evaluation method can be use to unlock the next stage.

In the third stage, instead of adding a new motion, one state transition is unlocked as indicated by purple arrow. This allows the user to know more on the motions that already learnt previously since the motions are belong to the same “form”. The same evaluation method can be use to unlock the next stage.

In the last stage, one more motion from other “form” is unlocked as indicated by the blue arrow. Unlike the previous stages, it is the first stage that there consists of two state transitions in one state. That is, user is able to perform two different motions after a particular motion. In this way, user will be able to learn how other motions in other “forms” can be chained. Users therefore can get experience on different “forms” through trying out this game.

CHAPTER 8 – PROJECT PROGRESS

In this chapter we are going to describe the progress of our project.

Period	Progress
Summer 2007	● Developed a little program to show the variation of accelerations as a hand-on practice on handling data captured from the Wii Remote. ● Got the Wii Remote connected to PC via Bluetooth and Wii Remote API. ● Read some papers about motion classification using accelerometers ● Searched for Wii Remote API for Windows.
September 2007	● Refined our project goal and schedule ● Searched for more information about Wii games and Wii Remote ● Studied different kinds of data representations ● Successfully make the data report in regular interval using threading ● Switched to a better Wii Remote API
October 2007	● First implemented a classifier based on Euclidean Distance Algorithm, ● Implemented a console to generate more than 40 types of attributes in total; served as the input for Weka ● Got familiar with the use of Weka and made use of it to generate source code for a tree-based classifier ● Tried to improve the accuracy of tree-based classifier by - including sample data with different orientations - normalizing amplitudes - discretizing attributes
November 2007	● Implemented classifiers based on Dynamic Time Warping Algorithm, and spent some effort in improving the classification rate following what we have mentioned in chapter 4.4. ● Console rewrote to suit Dynamic Time Warping ● Included benchmark mode in console, which was used to analyze the classification rate on different types of motions ● Worked on the first term report

Period	Progress
Christmas 2007	● Started implementing early calculation on motion classification. ● Studied the use of alternate inputs on different game or graphics rendering engine
January 2008	● Implemented online classification - Speedup motion classification ● Tried out Buttonless classification ● Started to implement Wii Style Library for better integration on other application ● Decided to use Irrlicht as the rendering engine; built a program to control characters to move around
February 2008	● Finished implementing Wii Style Library - Embedded it into Irrlicht to demonstrate motion classification ● Designed 11 dao motions and animated them in 3D Studio Max ● Developed the Wii Style Development Tool ● Met with department staff regarding the magnetic motion capturing system
March 2008	● Worked on the demo game - Integrated Wii Remote and Wii Style library for motion classification - Completed the player model with a dao attached - Imported FreeType library - Finished a playable demo ● Implemented Motion Finite State Machine - To support series mode in demo games ● Re-wrote the program to convert Wii Remote input into keyboard input for better performance - Used to play Onimusha 3 ● Prepared for the presentation
April 2008	● Some work on improving the demo game - Added a free-fight mode to suit our learning purpose ● Implemented classifier teacher program - For users to create their own profile ● Worked on the final report

CHAPTER *9* – DIFFICULTIES ENCOUNTERED

In this chapter we are going to describe the difficulties we encountered in this project so far. There are two main difficulties encountered up to this moment – User independent, and how players hold Wii Remote: the orientation.

This chapter comprises the following sections:

- 9.1 Problem on User independent
- 9.2 How player holds Wii Remote: Orientation

9.1 PROBLEM ON USER INDEPENDENT

As mentioned previously in section 3.3, "user independent" means different players perform the same motion in their own style, for example, performing at different speed, or with different amplitude. However, this leads to the problem on classification.

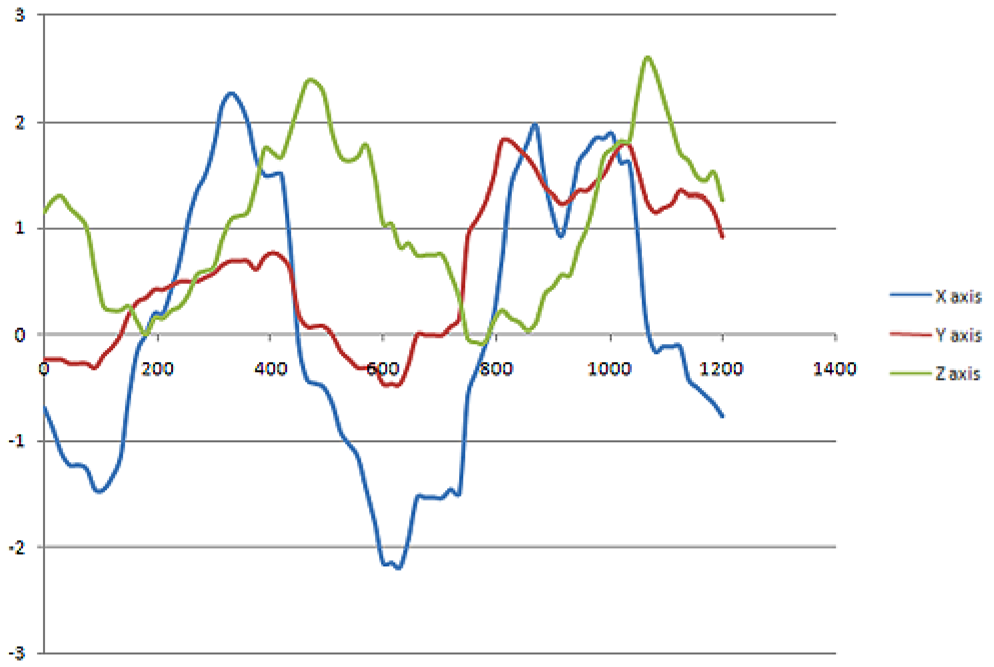


Fig 9.1 (a)

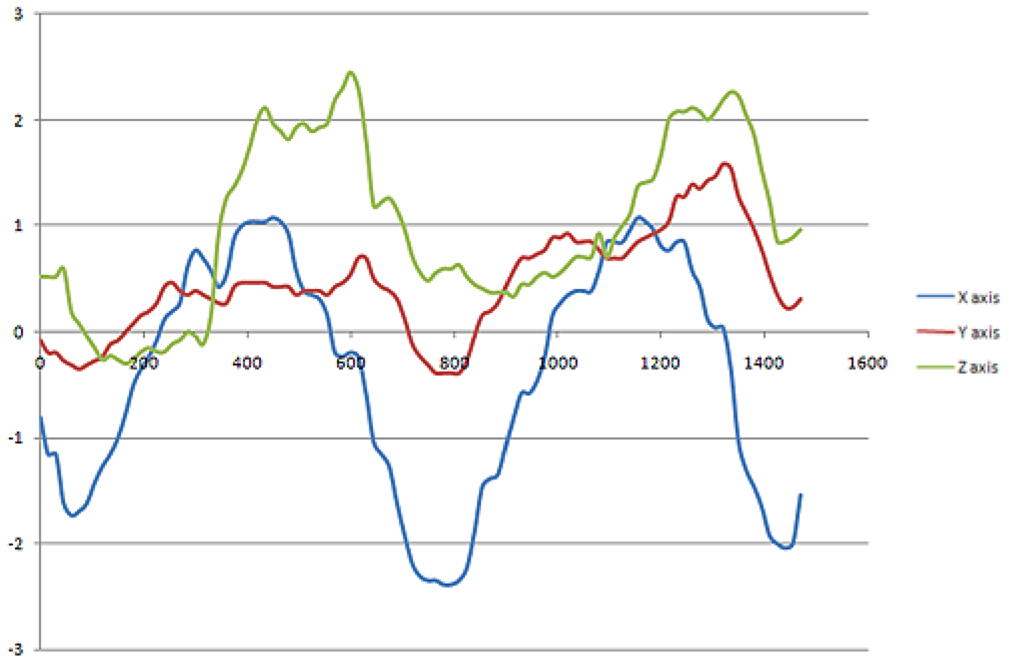


Fig 9.1 (b)

Fig 9.1 shows the acceleration-time graphs in which 2 players perform the same motion – drawing the number "3". Although they perform the same motion but still there are some differences in the changes of accelerations.

In fig 9.1 (a) and (b), it is observed that the changes of accelerations for some axes are quite different, including general shape of the graph, as well as the amplitudes of accelerations. For example:

- For Z axis, the curve shape is generally sharper in (a) than in (b).
- It is also observed that the curve representing the accelerations of x axis in (a) has greater amplitude than that in (b).

Dynamic Time Warping basically performs similarity search on 2 different curves. Although it is observed that the durations of motions, as well as the position of local maximums and minimums of X and Y axes are quite different; still this is solved by Dynamic Time Warping due to its non-linear mapping properties.

However, Dynamic Time Warping itself cannot overcome the 2 differences said above. These 2 differences are just treated as errors in classification. Therefore if there is another motion that is similar, the motion can be classified wrongly.

It seems to be possible of using tree-based or rule-based classifiers to classify the motions; however, using such classifiers requires a huge amount of sample data from different players. If the amount of sample data is not enough, then a player's style of performing will not be taken into considerations. Another problem is, if there are new sets of sample data added to the classifier, it has to be re-built again.

9.2 How player holds the Wii Remote: Orientation

Another difficulties facing is the orientation. Orientation refers to how Wii Remote is held, whether it is held upright, slightly rotated, or rotated greatly.

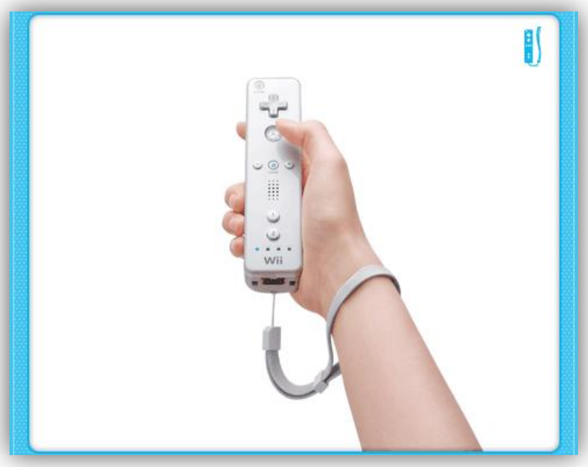


Fig 9.2 A player is holding Wii Remote upright.

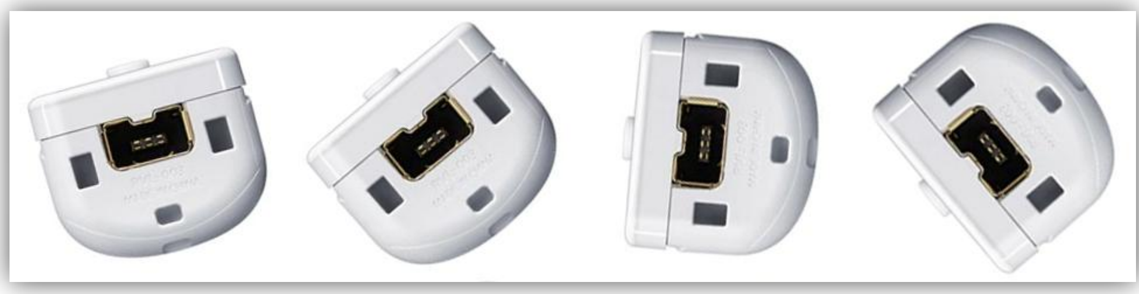


Fig 9.3 Some other possible orientations of holding a Wii Remote (back view)

We noted that if the Wii Remote is holding with different orientations, no matter if the remote is at rest or moving, the recorded accelerations are different as well.

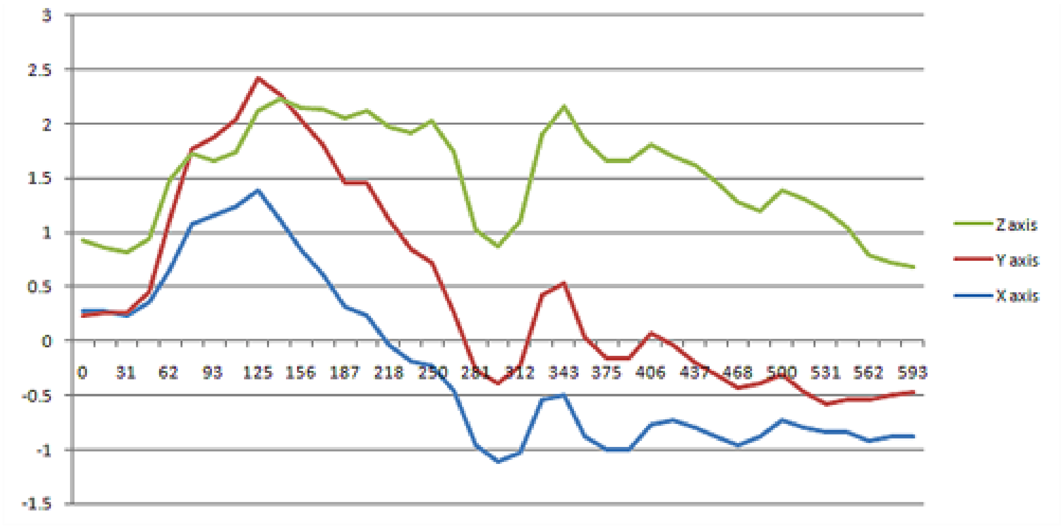


Fig. 9.4 (a)

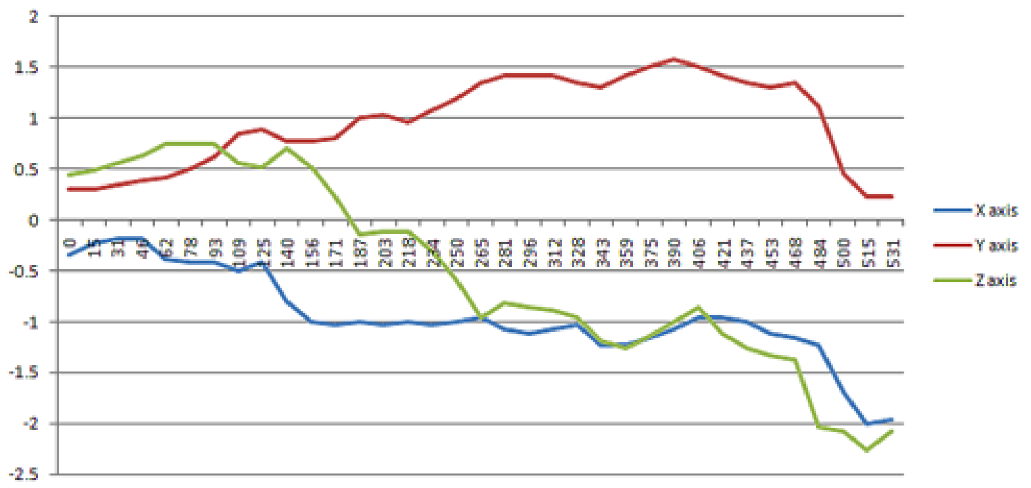


Fig 9.4 (b)

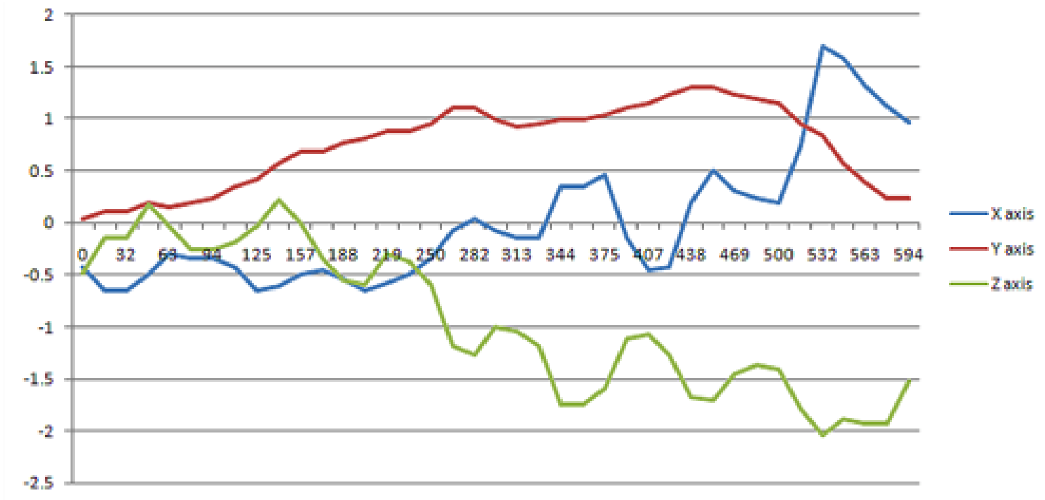


Fig 9.4 (c)

Fig 9.4 (a), (b) and (c) show the acceleration-time graph of the same motion done by the same player, with 3 different orientations of holding Wii Remote. It is obvious that there are some differences between the curves of accelerations. The differences are summarized as follows:

- Consider X axis. It is observed in graph (c) the acceleration values generally increase; whereas in graph (b), the acceleration values generally decrease; still in graph (a), the acceleration values first reach a global maximum, and then fall down. With 3 different orientations, the trend of the change in acceleration values is completely different from each other.
- Consider Y axis. Although graphs (b) and (c) show similar shape, still in graph (a), the acceleration values attain the maximum at the beginning, and then keep decreasing; in graphs (b) and (c), both of them have the acceleration values decrease only towards the end. Similar observations also exist for Z axis.

With the significant differences in accelerations, Dynamic Time Warping will just treat them as errors. Thus the errors are huge and resulting in incorrect classifications.

We have also tried including sample data of different orientations in both tree-based / rule-based classifiers, as well as Dynamic Time Warping. In tree-based / rule-based classifiers, some motions are classified wrongly still. As in Dynamic Time Warping, including those sample data can help, but it fails again once there are little variations in the orientation. Even though it is correctly classified, the DTW error is still quite high.

CHAPTER *10* – CURRENT LIMITATIONS

In this chapter we are going to describe some limitations of our current classifier at this moment.

This chapter comprises the following sections:

- 10.1 Button Required
- 10.2 Fixed Orientations
- 10.3 Same motion data for visually different motions

10.1 BUTTON REQUIRED

Currently, in order to indicate the start and the end of motion, users are required to press a button on Wii Remote while performing the motions.

10.2 FIXED ORIENTATIONS

As mentioned in section 9.2, the orientation of the Wii Remote will affect the pattern of the accelerations data. One possible solution is to apply transformation on the data. However, being not able to obtain the direction of the gravity, it is impossible to know what transformation we should apply. Therefore, during our study, we assume the orientation is the same as model motion. But of course, developer can include motion data of the same motion using different orientation.

10.3 SAME MOTION DATA FOR DIFFERENT MOTION

As mentioned in Section 2.2.1, Wii Remote captures motion data in terms of acceleration values only. It is possible for two visually different motions can have similar acceleration values. Since our classifier just classifies motion based on the acceleration values only, so if such case really occurs, then our classifier may not be able to classify between these two visually different motions accurately.

CHAPTER *11* – POTENTIAL FUTURE WORK

Although this project comes to the end, this does not mean that this is completed. In this chapter we are going to outline some rooms for future development of this project.

This chapter comprises the following sections:

- 11.1 "Button-less"
- 11.2 Different Weapons
- 11.3 A More Complete C++ Library for Wii Style
- 11.4 More Functions on Development Tool
- 11.5 Multiple Wii Remotes

11.1 "BUTTON-LESS"

To perform a motion, currently users have to press the button just before they start performing the motion, and release the button once the motion is completed. Yet, there is possibility to determine the start and the end of motion by looking at the overall acceleration on Wii Remote or consider the duration of the motions.

11.2 DIFFERENT WEAPONS AND NUNCHUK SUPPORT

In this project, we mainly use dao as the major weapon to demonstrate our motion classifier. To let users learn more about the 18 major weapons in Chinese martial arts, we can try to incorporate Wii Remote with more weapons, for example, Jian, Halberd, axe and so on.

Fig 11.1(a) Axe

Fig 11.1(b) Halberd

We can also try to incorporate Nunchuk together with Wii Remote into some two hand weapons; for example, Qiang.

11.3 A MORE COMPLETE C++ LIBRARY FOR WII STYLE

Wii Style is solely designed us, therefore it is possible that functions that is wanted by the developers are not included in our library. We can enhance the library to include more functions that are necessary for developers.

11.4 MORE FUNCTIONS ON DEVELOPMENT TOOL

In Wii Style Development Tool, although it is a complete development tool for developer to record their motion and building the classifier, there are still many functionalities that can be added to enhance the development tool. For example, average of motion data is done automatically in the data preprocessing module so as to increase the classification rate. To allow more customization and tuning for developer to further increase the classification rate with their own defined motions, more options can be included in the data preprocessing module like noise filtering or time matching.

11.5 MULTIPLE WII REMOTES

Playing games together with friends is more enjoyable than playing the game alone, and so do for learning. Therefore support of multiple Wii Remotes on game can also be considered to bring a greater happiness or learning outcome to the users.

CHAPTER *12* – CONCLUSION

To conclude our project, we have successfully built a motion classifier featuring Chinese martial arts. We compiled a C++ library for others to use the classifier in their own applications. On the other hand, we developed a development tool in graphical user interface for recording motions and building up the classifier.

To demonstrate the functionality of our classifier, we integrated the motion classifier into Onimusha 3 – a motion game with dao, one of the 18 major weapons, as one of the weapons for the game character. We also implemented a mini demo to demonstrate features of our classifier.

We achieved the objective of our project. With Onimusha 3, people can use a revolutionary approach to play the game. With our mini demo, people can use the similarity feature inside the demo to know how well they performed the motion. They can further improve and eventually master different forms. In both demos, people perform motions in front of the computer and they are then shown on the computer screen.

Throughout the project, we have acquired knowledge on different areas, including data mining, pattern classification, GUI programming and graphics rendering techniques.

Last but not the least, we have learnt to be self-motivated and fast-learner by studying new knowledge ourselves. This is essential when we begin our career path.

CHAPTER *13* – CONTRIBUTION OF WORK

In this chapter, my contribution of work will be described.

1. Preparation

   We have been preparing this project since the last summer. As our project is related with two areas – data mining, and Wii Remote, and we have a little knowledge on these two areas, so during the summer holiday we tried to study some related materials.

   We have tried a motion recognition neural network, which was developed from a research project; but unfortunately we failed to operate it and we were not able to get any support from the project team.

   Apart from this, I have read a book about data mining to get some basic understanding, as well as a number of papers specifically related to activity recognition using accelerometers, as well as different formats of data representation.

2. Semester 1

   After we have got some understanding on the background information, we decided our project objective. The first thing to do is to build up a motion classifier to classify player’s motion. Player’s motion can be classified based on acceleration values obtained from Wii Remote; my partner built an interface to get acceleration values at fixed interval of time.

After that, we started the construction of the motion classifier. I tried to build the classifier with decision trees. I first followed LiveMove Whitepaper to construct and compute certain attributes based on motion data, which are used for building up the decision tree, and later on around 40 attributes are constructed together with my partner.

I noticed that the classification result is quite good for discrete motion; however, this is not the case for complex motion like the dao motions. In order to deal with this, I tried to do some treatment on the motion data aimed at increasing the classification accuracy. Unfortunately, the improvement was not obvious, while Dynamic Time Warping, which was implemented by my partner, had a satisfactory result, and so we chose it as our motion classification algorithm.

Approaching the end of semester 1, we started preparing our project report and presentation. Since my partner focused on tuning the Dynamic Time Warping, so I have completed the majority of reports and presentations; still my partner finished the part regarding the tuning of Dynamic Time Warping.

3. Semester 2

   In semester 2, I mainly worked on the mini-demo featuring our motion classifier. Initially I and my partner expected to complete all the human skeletal animations within irrlicht graphics engine, and so I spent some time in trying to understand the skeletal system provided in irrlicht engine; however, we gave up this system and used built-in animations of models instead.

To construct the scene for the demo, I downloaded a gymnasium model from the internet and further edited it (included editing its polygon as well as applying texture mapping) so that it could be served as the scene for our demo. The model was discarded later on due to strange lighting of that model.

I also implemented the control of characters in Irrlicht engine by keyboards and Wii Remote, using Wii Style Library; as well as integrated Freetype Font Library into the irrlicht engine for better display.

Approaching the end of the semester, again I completed most of the contents of the presentations and reports. My partner did some proofreading on the report and modified some of contents in a better representation.

4. Conclusion

Throughout this project, I am exposed to different methods in pattern classification, from similarity search to decision trees. I also know that game development is not as easy as I thought before. The statement holds even for a mini-game. Even though in our mini-game demo we have to use a considerable amount of time to deal with different parts of it.

CHAPTER *14* – REFERENCE

[1]: “*The Games Industry: Past, Present & Future*”,
http://www.gamesinvestor.com/Research/History/history.htm

[2]: “*The Big Ideas Behind Nintendo's Wii*” - BusinessWeek November 16 2006,
http://www.businessweek.com/technology/content/nov2006/tc20061116_750580.htm

[3]:“*Breaking: Nintendo Announces New Revolution Name - 'Wii'.*”, Gamasutra.
http://www.gamasutra.com/php-bin/news_index.php?story=9075

[4]: “*Wii Remote*”, http://en.wikipedia.org/wiki/Wii_Remote

[5]: “*- WiiYourself! - gl.tter's native C++ Wiimote library.*” http://wiiyourself.gl.tter.org/

[6]: “*LiveMove Pro Director's Cut Manual*”
http://www.ailive.net/papers/directorsManual_en.pdf

[7]: Ian H. Witten and Eibe Frank (2005) "*Data Mining: Practical machine learning tools and techniques*", 2nd Edition, Morgan Kaufmann, San Francisco, 2005.

[8]: “*Euclidean Distance*”, http://en.wikipedia.org/wiki/Euclidean_distance

[9]: “*Dynamic Time Warping*”, http://en.wikipedia.org/wiki/Dynamic_time_warping

[10]: “*11.2. Dynamic Time Warping*”,
http://www.ics.mq.edu.au/~cassidy/comp449/html/ch11s02.html

[11]: “*Exact Indexing of Dynamic Time Warping*”,
http://www.cs.ucr.edu/~eamonn/vldb_keogh_2002.ppt

[12]: “*Wii Remote*”,
http://www.cse.cuhk.edu.hk/~csc3510/notes/3510_lecture11_Wii_Controller.pdf

[13]: Eamonn Keogh*, Selina Chu, David Hart, and Michael Pazzani (2001): “*An Online Algorithm for Segmenting Time Series*”,

http://www-scf.usc.edu/~selinach/segmentation-slides.pdf

[14]: "*Normalization* ", http://en.wikipedia.org/wiki/Normalization

[15]: Hagit Shatkay (1995) : "*The Fourier Transform – A Primer*",

http://www.phys.hawaii.edu/~jgl/p274/fourier_intro_Shatkay.pdf

[16]: "*Cooley-Tukey FFT algorithm* ",

http://en.wikipedia.org/wiki/Cooley-Tukey_FFT_algorithm

[17]: "*Dao (sword)*", http://en.wikipedia.org/wiki/Dao_(sword)

[18]: "*Irrlicht Engine - A free open source 3d engine*", http://irrlicht.sourceforge.net/

[19]: "*The FreeType Project*", http://www.freetype.org/

[20]: "*Irrlicht Engine :: View topic - TrueType font Support by FreeType Library*",

http://irrlicht.sourceforge.net/phpBB2/viewtopic.php?p=23386#23386

[21]: "*Irrlicht Engine :: View topic - TrueType font Support by FreeType Library*",

http://irrlicht.sourceforge.net/phpBB2/viewtopic.php?p=150888#150888

[22]: "*3D max Soccer Player Football*",

http://www.turbosquid.com/FullPreview/Index.cfm/ID/307330

[23]: "*Ascension Products - MotionStar Wireless 2*",

http://www.ascension-tech.com/products/motionstarwireless.php

[24]: "Chinese martial arts", http://en.wikipedia.org/wiki/Chinese_martial_arts

APPENDIX *1* – WII STYLE API

In this appendix, it will describe the API for Wii Style, our C++ library for motion recording and classification.

A 1.1 THE CLASS AND NAMESPACE LIST

Table A1.1 and A1.2 below shows the list of namespaces and classes with brief descriptions, respectively.

Namespaces	Descriptions
WiiStyle	The namespace for our Wii Style library, include three other namespaces
Data	Include all classes for storing motion information
Classifier	Include all classes for our motion classifier
Classifier::DTW	Include all classes and functions related to Dynamic Time Warping
Recording	Include all classes for recording motions

Table A1.1 List of namespace

Classes	Descriptions
Data::WiiDataList	Contains the raw motion data with vector operations
Data::WiiDataQueue	Contains the raw motion data with queue operations
Data::Motion	A class which stores the information of a motion
Data::MotionDB	A database which stores different motions and motion FSM
Classifier::DTW::DTWColumn	This class represents a column in the dynamic time warping cost matrix. It is a component used for computing the dynamic time warping cost matrix in a column by column manner
Classifier::ButtonOnline	Classifier for online classification with button pressing
Recording::MotionRecorder	A class for recording motions

Table A1.2 List of classes

A 1.2 CLASS WIISTYLE::DATA::WIIDATALIST

- **Public Member Attributes**
  - struct acceleration

```
struct acceleration {

    float x, y, z;

};
```

This is the data structure for storing three accelerations obtained from Wii Remote.

Variables:

`x` – Variable for storing x-acceleration.

`y` – Variable for storing y-acceleration.

`z` – Variable for storing z-acceleration.

  - struct nunchuk

```
struct nunchuk {

    struct acceleration Acceleration;

};
```

This is the data structure for Nunchuk related variables in one data frame.

Variables:

`Acceleration` – Variable for storing the three accelerations obtained from Nunchuk.

- struct wiiDataFrame

```
struct wiiDataFrame {

    struct acceleration Acceleration;

    struct nunchuk Nunchuk;

    long time;

};
```

This is the data structure for storing one data frame for Wii Remote and Nunchuk.

Variables:

`Acceleration` – Variable for storing the three accelerations obtained from Wii Remote.

`Nunchuk` – Variable for storing Nunchuk related variables.

`time` – the time for this data frame.

- **Constructor Documentation**

- WiiDataList();

  Class constructor

- ~WiiDataList();

  Class destructor

- **Public Member Functions**

- int length();

  Returns the total number of data frames in the data list.

  Return Value:

  - Total number of data frame in the data list.

- void add(float x, float y, float z, long time);

  Adds a new data frame to the data list, without Nunchuk acceleration information.

Parameters:

- `x` : x-acceleration obtained from Wii Remote
- `y` : y-acceleration obtained from Wii Remote
- `z` : z-acceleration obtained from Wii Remote
- time : the time of this data frame

■ void add(float x, float y, float z, float ex, float ey, float ez, long time);

Adds a new data frame to the data list, with Nunchuk acceleration information.

Parameters:

- `x` : x-acceleration obtained from Wii Remote
- `y` : y-acceleration obtained from Wii Remote
- `z` : z-acceleration obtained from Wii Remote
- `ex` : x-acceleration obtained from the Nunchuk connected with Wii Remote
- `ey` : y-acceleration obtained from the Nunchuk connected with Wii Remote
- `ez` : z-acceleration obtained from the Nunchuk connected with Wii Remote
- `time` : the time of this data frame

■ void clear();

Clears all data frames in the data list.

- wiiDataFrame& operator[](int index);

  Retrieves a data frame specified by `index`.

  Parameters:

  - `index`: the index of data frame to be retrieved.

  Return Value:

  - A `wiiDataFrame` structure which the index equal to `index`.

- std::string toString();

  Return a displayable representation of data excluding the Nunchuk accelerations values in the data list.

  Return Value:

  - A string which is the displayable representation.

- void saveState(FILE *file);

  Save the state of the data list to a destination file.

  Parameters:

  - `file`: the `FILE` pointer pointing to the destination file.

- void loadState(FILE *file);

  Load the state of the data frames from a source file.

  Parameters:

  `file`: the `FILE` pointer pointing to the source file.

A 1.3 CLASS WIISTYLE::DATA::WIIDATAQUEUE

- **Constructor Documentation**
  - WiiDataQueue(int initSize);

    Class constructor

    Parameters:

    - `initSize`: size of the queue.
  - ~ WiiDataQueue();

    Class destructor
- **Public Member Functions**
  - WiiDataList::wiiDataFrame pop();

    Pop the first data frame from the data queue. The data frame is then removed from the queue and returned.

    Return Value:

    - A `wiiDataFrame` structure which the index equal to 0.
  - void popUntil(unsigned int targetSize);

    Keep removing data frame from data queue until the queue size drop to `targetSize`.

    Parameters:

    `targetSize`: the target size of the queue

A 1.4 CLASS WIISTYLE::DATA::MOTION

- **Constructor Documentation**

  - Motion();

    Class constructor

  - Motion(std::string motionName);

    Class constructor

    Parameters:

    - `motionName`: name of the motion.

  - ~ Motion();

    Class destructor

- **Public Member Functions**

  - std::string name();

    This function returns the motion name

    Return Value:

    - A string containing the motion name.

  - int size();

    This function returns the number of raw motion data recorded in this motion.

    Return Value:

    - Total number of raw motion data.

  - void addRawData(const WiiDataList &data);

    This function adds a raw motion data “`data`” to the motion.

    Parameters:

    - `data`: the raw motion data to add.

  - void clearRawData();

    This function clears all the raw motion data for the motion.

- void deleteRawData(int index);

  This function deletes the raw motion data with index "`index`".

  Parameters:

  - `index`: index of the raw motion data to be delete.

- WiiDataList& operator [](int index);

  This operator returns a raw motion data of the motion at position "`index`".

  Parameters:

  - `index`: position of the target raw motion data.

  Return Value:

  - The raw motion data.

- bool addFollowedMotion(std::string name);

  This function adds a motion that is going to be followed after this motion with name specified by "`name`". Part of the motion finite state machine.

  Parameters:

  - `name`: name of the following motion.

  Return Value:

  - Boolean value. `true` indicates the addition is successful; `false` otherwise.

- std::vector<std::string> getFollowedMotions();

  This function returns a list of following motions after this motion. Part of the motion finite state machine.

  Return Value:

  - Vector list of names of following motions.

- void removeFollowedMotion(std::string name);

  This function removes a following motion with name "`name`" from this motion. Part of the motion finite state machine.

  Parameters:

  - `name`: name of the following motion to be removed.

- void clearFollowedMotions();

  This function removes all the following motions of this motion. Part of the motion finite state machine.

- void saveState(FILE *file);

  Save the state of the motion to a destination file, except for `processedData`.

  Parameters:

  - `file`: the `FILE` pointer pointing to the destination file.

- void loadState(FILE *file);

  Load the state of the motion from a source file, except for `processedData`.

  Parameters:

  - `file`: the `FILE` pointer pointing to the source file.

- **Public Member Variable**

  - `std::vector<WiiDataList> processedData;`

    - A vector of WiiDataList is used to store all the processed data. This vector will not be saved by using `saveState`.

A 1.5 CLASS WIISTYLE::DATA::MOTIONDB

- **Constructor Documentation**
  - MotionDB();

    Class constructor
  - ~MotionDB();

    Class destructor

- **Public Member Functions**
  - bool addNew(std::string motionName);

    This function adds a new motion to the motion database with name specified by `motionName` parameter.

    Parameter:
    - `motionName`: name of the new motion.

    Return Value:
    - Boolean value. `true` indicates the addition is successful; `false` otherwise.
  - bool remove(std::string motionName);

    This function removes a motion from the motion database specified by "`motionName`" parameter.

    Parameter:
    - `motionName`: name of the motion.

    Return Value:
    - Boolean value. `true` indicates the removal is successful; `false` otherwise.

- int indexOf(std::string motionName);

  This function returns the index of the motion in the database specified by "`motionName`" parameter.

  Parameter:

  - `motionName`: name of the motion to query.

  Return value:

  - `index` of the specified motion. -1 if the specified motion is not found.

- Motion& operator [] (std::string motionName);

  This operator returns the motion in the database specified by “`motionName`” parameter.

  Parameter:

  - `motionName`: name of the motion.

  Return value:

  - The motion requested.

- Motion& operator [] (int index)

  This operator returns the motion data of the motion in the database specified by “`index`” parameter.

  Parameter:

  - `index`: position of the motion in the database.

  Return value:

  - The motion requested.

- bool addStartupMotion(std::string name);

  This function adds a startup motion specified by "`name`" parameter to the database. Part of the motion finite state machine.

  Parameters:

  - `name`: name of the startup motion.

  Return value:

  - Boolean value. `true` indicates the addition is successful; `false` otherwise.

- std::vector<std::string> getStartupMotions();

  This function returns the list of startup motions in the database. Part of the motion finite state machine.

  Return value:

  - A vector list of names of the startup motions.

- void clearStartupMotions();

  This operator removes all the startup motions in the database. Part of the motion finite state machine.

- bool saveState(std::string filePath);

  Save the state of the motion database to a destination file.

  Parameters:

  - `filePath`: path of the destination file.

  Return value:

  - Boolean value. `true` indicates the saving is successful; `false` otherwise.

- bool loadState(std::string filePath);

  Load the state of the motion database from a source file.

  Parameters:

  - `filePath`: path of the source file.

  Return value:

  - Boolean value. `true` indicates the loading is successful; `false` otherwise.

A 1.6 FUNCTIONS IN WIISTYLE::CLASSIFIER::DTW

- **Public Member Functions**

  - float getThreeDimensionDistance(

    Data::WiiDataList &dataA, Data::WiiDataList &dataB, int indexA, int indexB);

    This function takes in two data list “`dataA`” and “`dataB`”, and computes the three-dimensional distance between two points with position specified by “`indexA`” and “`indexB`” respectively.

    Parameters:

    - `dataA`: the data list involved.
    - `dataB`: the data list involved.
    - `indexA`: the position of the point in data list “`dataA`”
    - `indexB`: the position of the point in data list “`dataB`”

    Return Value:

    - The three-dimensional Euclidean distance between these two points.

- float getSixDimensionDistance(

  Data::WiiDataList &dataA, Data::WiiDataList &dataB, int indexA, int indexB);

This function takes in two data list "`dataA`" and "`dataB`", and computes the six-dimensional distance between two points with position specified by "`indexA`" and "`indexB`".

Parameters:

- `dataA`: the data list involved.
- `dataB`: the data list involved.
- `indexA`: the position of the point in data list "`dataA`"
- `indexB`: the position of the point in data list "`dataB`"

Return Value:

- The six-dimensional distance Euclidean distance between these two points.

■ float **dtwMatrix(Data::WiiDataList &dataA, Data::WiiDataList &dataB, float(*getDistance)(
Data::WiiDataList &dataA, Data::WiiDataList &dataB, int indexA, int indexB));

This function allocates memory, and returns the 2-dimensional dynamic time warping cost matrix of the given two data list. Please call `freeDTWMatrix` when the matrix will no longer be used to free the memory.

Parameters:

- `dataA`: the data list involved.
- `dataB`: the data list involved.
- `getDistance`: a function pointer pointing to the function that is used to calculate the Euclidean distance of two points in the two data list.

Return Value:

- The two-dimensional dynamic time warping cost matrix.

■ void freeDTWMatrix(float **dtwMatrix, Data::WiiDataList &dataA);

This function de-allocates the memory used in the dynamic time warping cost matrix.

Parameters:

- `dtwMatrix`: the dynamic time warping matrix that is going to free memory.
- `dataA`: the "`dataA`" used in constructing the "`dtwMatrix`".

- std::vector<int> *getDTWMapping(

  Data::WiiDataList &dataA, Data::WiiDataList &dataB,

  float(*getDistance)(

  Data::WiiDataList &dataA, Data::WiiDataList &dataB, int indexA, int indexB));

  This function returns the point-to-point mapping using dynamic time warping between data lists "`dataA`" and "`dataB`".

  Parameter:

  - `dataA`: the data list involved.
  - `dataB`: the data list involved.
  - `getDistance`: a function pointer pointing to the function that is used to calculate the Euclidean distance of two points in the two data list.

  Return value:

  - A vector list of integer which is of size equal to number of data frame in `dataA`. This is a mapping of points of `dataA` to `dataB`. The integer values are the index of data frame in `dataB`.

- float dtwError(

  Data::WiiDataList &dataA, Data::WiiDataList &dataB,

  float(*getDistance)(

  Data::WiiDataList &dataA, Data::WiiDataList &dataB, int indexA, int indexB));

  This function returns the dynamic time warping error between two data lists “`dataA`” and “`dataB`”.

  Parameters:

  - `dataA`: the data list involved.
  - `dataB`: the data list involved.
  - `getDistance`: a function pointer pointing to the function that is used to calculate the Euclidean distance of two points in the two data list.

  Return value:

  - The dynamic time warping error.

- char maxDuration(Data::WiiDataList &dataA, Data::WiiDataList &dataB, Data::WiiDataList &dataC);

  This function compares three data lists “`dataA`”, “`dataB`” and “`dataC`” and find out the data list with longest duration.

  Parameters:

  - `dataA`: the data list involved.
  - `dataB`: the data list involved.
  - `dataC`: the data list involved.

  Return value:

  - A character which is either “A”, “B”, “C”. Returning “A” indicates that data list “`dataA`” has the longest duration, and similarly for “B” and “C”.

- void generateAverageOfThree(Data::MotionDB &db);

  This generates the average of every three motion data of every motion in the motion database "`db`" and stores it in "`processedData`" in each motion. In case there is less than three motion data in the motion, all the motion will be store in "`processedData`".

  Parameters:

  - `db`: the provided motion database for generating the average motions. This variable will be modified.

A 1.7 CLASS WIISTYLE::CLASSIFIER::DTW::DTWCOLUMN

- **Constructor Documentation**

  - DTWColumn(int height, bool nunchuk);

    Construct an empty `DTWColumn` object with height specified by "`height`".

  Parameters:

  - `height`: height of the `DTWColumn`, usually the number of data frame of the model motion data.
  - `nunchuk`: indicates the usage of Nunchuk. `true` if Nunchuk is used, `false` otherwise.

- DTWColumn(const DTWColumn &val);

  Copy the content of "`val`" to a new `DTWColumn` object.

  Parameters:

  - `val`: the `DTWColumn` object to be copied.

- ~DTWColumn();

  Class destructor

- **Public Member Functions**

  - void setModelList(Data::WiiDataList val);

    This function sets "`val`" as the model motion data list.

    Parameter:

    - `val`: the model motion data list.

  - void setClassifyList(Data::WiiDataList &val);

    This function sets "`val`" as the motion data list that is going to be classified.

    Parameter:

    - `val`: the motion data list that is going to the classified.

  - void iterate();

    This function computes the next column in the dynamic time warping cost matrix.

  - float getValueAtTop();

    This function returns the top entry in the `DTWColumn`.

    Return value:

    - The value of the top entry in the `DTWColumn`.

  - void reset(bool nunchuk);

    This function reset the state of the `DTWColumn` so that it can be used in the next classification.

    Parameter:

    - `nunchuk`: indicates the usage of Nunchuk. `true` if Nunchuk is used, `false` otherwise.

  - std::string toString();

    Return a displayable representation of the `DTWColumn`.

    Return Value:

    - A string which is the displayable representation.

A 1.8 CLASS WIISTYLE::CLASSIFIER::BUTTONONLINE

- **Public Member Attributes**

  - struct report

```
struct report {
    std::string motionName;
    int motionID;
    float error;

    bool operator > (report toCMP) {
        return (this->error > toCMP.error);
    }

    bool operator < (report toCMP) {
        return (this->error < toCMP.error);
    }
};
```

This is the data structure of a report for storing motion name and the associated DTW error after motion classification.

Variables:

`motionName` – name of the motion.

`motionID` – ID of the motion.

`error` – the DTW error after motion classification.

- **Constructor Documentation**

  - ButtonOnline(wiimote *wii, bool nunchuk);

Class constructor

Parameters:

- `wii`: pointer to Wii Remote object.
- `nunchuk`: indicates the usage of Nunchuk. `true` if Nunchuk is used, `false` otherwise.

  - ~ButtonOnline();

Class destructor

- **Public Member Functions**

  - void buildClassifier(Data::MotionDB &db);

    This function builds a motion classifier to classify all the motions within motion database "`db`". All `processedData` in each motion is used as model motion data.

    Parameter:

    - `db`: the motion database.

  - int getNumOfMotions();

    This function returns the total number of motion in the motion database.

    Return value:

    - Number of motions in the motion database.

  - Data::MotionDB getTeachingDB();

    This function return a database with all model motions insides the classifier. Model motion data are included as raw motion data. Motion FSM is not included.

    Return value:

    - A motion database containing all the model motions, with model motion data but not motion FSM.

- bool appendMotionData(Data::Motion &motion);

This function appends all the motion data stored in `processedData` stored in the motion specified by "`motion`" to the motion of the same name in the classifier database as model motion data.

Parameter:

- `motion`: motion containing the motion data in `processedData` to be appended as model motion data.

Return value:

- Boolean value. `true` indicates the motion data is appended is successfully; `false` otherwise.

- bool replaceMotionData(Data::Motion &motion);

This function replace all the model motion data of the same name in the motion database with all the motion data stored in `processedData` stored in the motion specified by "`motion`".

Parameter:

- `motion`: motion containing the motion data in `processedData` that is going to replace the model motion data.

Return value:

- Boolean value. `true` indicates the motion is replaced is successfully; `false` otherwise.

- void startClassify();

This function starts the motion classification.

- bool isClassifying();

  This function tells the status of the classifier – whether it is classifying or not.

  Return value:

  - Boolean value. `true` indicates the classifier is classifying; `false` otherwise.

- report getPrevClassified();

  This function returns a report of the last classified motion.

  Return value:

  - The `report` structure storing the name, id, and error of the last classified motion.

- std::vector<report> getAllPrevClassified(bool sorted);

  This function returns reports of all motions in the last classification.

  Return value:

  - The vector structure of `report` structure which storing the name, id and error of all motions in the last classification.

- std::string getCurrentState();

  This function gets the current state in the motion FSM of the motion classifier. Part of the motion finite state machine.

  Return value:

  - Name of the previously motion in the FSM. It returns an empty string if there is no previous motion. For example when it is at the start state.

- bool proceed(bool autoReset);

  This function moves one step in FSM. Call this function to accept the classified motion and proceed to classify next set of motions that is followed by the classified motion. Usually this function is called when the motion is correctly classified. "`autoReset`" specifies whether "`motionReset`" should be called when there is no motion to be classify for the next set of motions. Part of the motion finite state machine.

  Parameter:

  - `autoReset`: Boolean value, specifies if auto reset of to start state is allowed or not. `true` value means allowed; `false` otherwise.

  Return value:

  - Boolean value. If the motion FSM is restarted, `true` is returned; `false` otherwise.

- bool proceedTo(std::string motionName);

  This function rejects the classified motion and updates the next set of motions to classify according to the setting in the specified motion with name "`motionName`". Part of the motion finite state machine.

  Parameter:

  - `motionName`: name of the specified motion that should proceed to in the next state in motion FSM.

  Return value:

  - Boolean value. `true` if the function call is successes, `false` otherwise.

- void motionReset();

  This function resets the motion FSM to classify to initial set of motions.

A 1.9 CLASS WIISTYLE::RECORDING::MOTIONRECORDER

- **Constructor Documentation**
  - MotionRecorder(wiimote &wii);

    Class constructor

    Parameters:

    - `wii`: Wii Remote object.
  - ~ MotionRecorder();

    Class destructor

- **Public Member Functions**
  - void startRecord(int interval);

    This function starts recording data from Wii Remote. Data is reported at an interval specified by “`interval`”.

    Parameter:

    - `interval`: The interval between each recorded data, in millisecond. Value of zero means data is recorded only when it is changed.
  - void stopRecord();

    This function stops recording data from Wii Remote.
  - bool isRecording();

    This function returns the status of motion recorder – whether the recorder is recording motion.

    Return Value:

    - Boolean value. `true` indicates the recorder is recording motion at that moment; `false` otherwise.

- Data::WiiDataList getRecordedData();

  This function returns recorded motion data.

  Return Value:

  - A data list which contains the recorded motion data.

APPENDIX *2* – SUBVERSION LOG

To keep a record on our project coding, we have used Subversion, a revision control system to manage our code. The following pages show our subversion log since last November, the time when we started using this system.

```
------------------------------------------------------------------------
r125 | 2008-04-14 10:46:45 +0800 (Mon, 14 Apr 2008) | 1 line

- This is the code used in the second term presentation.
------------------------------------------------------------------------
r124 | 2008-04-12 22:14:31 +0800 (Sat, 12 Apr 2008) | 5 lines

WiiStyleDevTool/DlgCurveViewer.h:
- Interchanged the color of Y and Z axis. Now the color for X axis is red, color for Y axis is green, color for Z axis is blue

WiiStyleTeacher:
- Fixed release build error
------------------------------------------------------------------------
r123 | 2008-04-07 23:54:56 +0800 (Mon, 07 Apr 2008) | 9 lines

WiiStyle/include/ButtonOnline.h
WiiStyle/src/ButtonOnline.cpp:
- Changed a function "getEmptyDB" to "getTeachingDB" which will include the model motions as well

WiiStyle/include/Motion.h:
- Changed argument with "const" in function "addRawData"

WiiStyleTeacher/FrmMain.h:
- Finished implementing the Wii Style Classifer Teacher
------------------------------------------------------------------------
```

r122 | 2008-04-07 12:40:14 +0800 (Mon, 07 Apr 2008) | 11 lines

WiiStyle/include/ButtonOnline.h
WiiStyle/src/ButtonOnline.cpp:
- Added functions "getEmptyDB", "appendMotionData", replaceMotionData"
- Renamed a function "_motionID" to "_getMotionID"

WiiStyleDevTool/DlgRecorder.h:
- Fixed some warnings
- Fixed a possible application hang when the form is minimize

WiiStyleTeacher:
- A new project which will take in a classifier, and change the "model motion", not yet finish implement
----
r121 | 2008-04-07 00:08:22 +0800 (Mon, 07 Apr 2008) | 9 lines

WiiStyle/include/MotionRecorder.h
WiiStyle/src/MotionRecorder.cpp:
- Fixed a bug in which the time entry is incorrect for the recorded data

WiiStyleDevTool/DlgRecorder.h:
- Implemented the Wii Remote Data displaying graph

WiiStyleDevTool/FrmMain.h
- Integrated with the "DlgRecorder" form
----
r120 | 2008-04-06 19:48:50 +0800 (Sun, 06 Apr 2008) | 12 lines

DemoGame/Main.cpp:
- Updated include header

WiiStyle:
- Added a new class "MotionRecorder" for recording motions
- Added a Wii Style interface header file for easy including of all header files in WiiStyle

WiiStyleDevTool:
- Added a new form "DlgRecorder" for displaying data from Wii Remote during recording motion, but not yet complete implementing

```
WiiToONIMUSHA3/Main.cpp:
- Updated include header
------------------------------------------------------------------------
r119 | 2008-04-02 16:03:05 +0800 (Wed, 02 Apr 2008) | 7 lines

DemoGame.txt
FreeFight.txt
ONIMUSHA3.txt:
- Updated to match the new database name

DMC3.txt:
- Removed as no longer in use
------------------------------------------------------------------------
r118 | 2008-03-30 12:13:33 +0800 (Sun, 30 Mar 2008) | 3 lines

- Changed some name space in project "WiiStyle"
- Renamed "DTWButtonOnline" to "ButtonOnline"
- Updated all other projects for the change in name space
------------------------------------------------------------------------
r117 | 2008-03-30 01:11:20 +0800 (Sun, 30 Mar 2008) | 15 lines

DemoGame/Main.cpp:
- Added HP text to the dummy
- Dummy now will move randomly, with collision detection with the wall
- Player will automatically facing the dummy when attacking
- Player and dummy will no other move vertically

DemoGame/Player.h
DemoGame/Player.cpp:
- Added a function "setYPos"

DemoGame/StaticModel.h
DemoGame/StaticModel.cpp:
- Added functions "setYPos", "addHitter"
- Added multiple text node support
- Added active collision detection support
------------------------------------------------------------------------
```

```
r116 | 2008-03-29 23:39:51 +0800 (Sat, 29 Mar 2008) | 2 lines

DemoGame:
- Added free fight mode support
------------------------------------------------------------------------
r115 | 2008-03-29 20:41:01 +0800 (Sat, 29 Mar 2008) | 2 lines

WiiStyle/src/MotionDB.cpp:
- Fixed a bug in which the start up motion is not clear when "loadState" is called
------------------------------------------------------------------------
r114 | 2008-03-29 20:27:03 +0800 (Sat, 29 Mar 2008) | 2 lines

DemoGame/media:
- Removed as no longer in use
------------------------------------------------------------------------
r113 | 2008-03-29 20:24:53 +0800 (Sat, 29 Mar 2008) | 5 lines

- Fixed all "Release" version build error

WiiStyleDevTool/FrmMain.h:
WiiStyleDevTool/FrmMain.resx:
- Added "Reset" button in tab "Motion FSM"
------------------------------------------------------------------------
r112 | 2008-03-29 19:43:18 +0800 (Sat, 29 Mar 2008) | 12 lines

WiiStyle:
- Suppressed some warnings

WiiStyle/include/Motion.h
WiiStyle/src/Motion.cpp:
- Added a function "removeFollowedMotion"

WiiStyle/src/MotionDB.cpp:
- Fixed a bug in which removing a motion did not remove the related motion in startup
motion and followed motions

WiiStyleDevTool/DlgCurveViewer.h:
- Fixed some warnings
------------------------------------------------------------------------
```

```
r111 | 2008-03-27 10:45:38 +0800 (Thu, 27 Mar 2008) | 2 lines

DemoGame/Main.cpp:
- Added sound when the dummy is being hit
------------------------------------------------------------------------
r110 | 2008-03-27 10:41:53 +0800 (Thu, 27 Mar 2008) | 1 line

- Renamed folder and file names from "WiiDevelopmentTool" to "WiiStyleDevTool"
------------------------------------------------------------------------
r109 | 2008-03-27 10:35:04 +0800 (Thu, 27 Mar 2008) | 2 lines

- Renamed project "WiiDevelopmentTool" to "WiiStyleDevTool"
- Removed project "WiiToDMC3"
------------------------------------------------------------------------
r108 | 2008-03-27 10:13:12 +0800 (Thu, 27 Mar 2008) | 2 lines

DemoGame:
- Fixed the incorrect position of the dao in the animation
------------------------------------------------------------------------
r107 | 2008-03-24 22:11:29 +0800 (Mon, 24 Mar 2008) | 19 lines

Main.cpp:
- Larger font is used
- Fixed some warnings
- Added HP to the dummy
- Font is changed to red color
- Changed initial text displayed at the top of the dummy
- The start position of the dummy is random now
- Now able to attack dummy
- The dummy will appear randomly after it dead

DemoGame/Player.h
DemoGame/Player.cpp:
- Added a function "isAttacking"
```

```
DemoGame/StaticModel.h
DemoGame/StaticModel.cpp:
- Fixed some warnings
- Added a overloaded function "setTopText"
- Added a function "getBoundingBox"
------------------------------------------------------------------------
r106 | 2008-03-24 19:00:16 +0800 (Mon, 24 Mar 2008) | 11 lines

- Integrated with Alex work

DemoGame/Main.cpp:
- Added TrueType Font Support
- Changed lighting
- Added gym
- Replaced dummy to wooden man

DemoGame/StaticModel.h
DemoGame/StaticModel.cpp:
- Added functions "addCubeToScene", "setTexture", "createTopTextNode", "setTopText"
------------------------------------------------------------------------
r105 | 2008-03-24 16:23:38 +0800 (Mon, 24 Mar 2008) | 8 lines

Main.cpp:
- Position of the dao is updated
- Fixed the alternate run and stand animation when using Wii Remote for moving

Player.h
Player.cpp:
- Fixed the dao flashing problem
- Added checking for "Invaild" motion type in function "updateAnimState"
------------------------------------------------------------------------
r104 | 2008-03-24 13:50:35 +0800 (Mon, 24 Mar 2008) | 19 lines

- Removed GYM model and Ground model
- Removed old skeleton model and old man model
- Used a new man model
```

Main.cpp:
- Removed GYM
- Attached Dao onto the player but still have problem
- Player now able to display running animation when running

Player.h
Player.cpp:
- Used a new model
- Added "Run" animation support
- Increased the moving speed
- Added a function "getCurrentAnim"
- Modified the ellipsoid size for collision detection

StaticModel.cpp:
- Added display of the bounding box

----

r103 | 2008-03-18 01:24:26 +0800 (Tue, 18 Mar 2008) | 3 lines

WiiStyle/include/DTWButtonOnline.h
WiiStyle/src/DTWButtonOnline.cpp:
- Changed the return type of function "proceed" from "void" to "bool". True will be returned if motion is reset

----

r102 | 2008-03-18 01:23:10 +0800 (Tue, 18 Mar 2008) | 16 lines

Main.cpp:
- Limited FPS of the game
- Classification can be done when a complete set of motion is animated
- Added a man and a dao into the scene

Player.h
Player.cpp:
- Modified the moving speed and rotation degree of the player to match the new limited FPS
- Added an item "Invalid" in enum Animation
- Added a function "getLastAnimInQueue"
- The last animation of a set of motion will not be cut
- Added to display bounding box

StaticModel.h
StaticModel.cpp:
- Added function "setRotation"
----
r101 | 2008-03-17 01:56:08 +0800 (Mon, 17 Mar 2008) | 11 lines

DemoGame/Main.cpp:
- Added motion reset after the player press move button

DemoGame/Player.h
DemoGame/Player.cpp:
- Changed function "playAnimation" to private
- Added motion queue support
- Added function "appendAnimation", "updateAnimState", "_isAnimCompleted", "_isLoopAnimation"
- Fixed a bug in not able to compile due to missing ';'
- Default animation played is "Stand"
- Next motion will only be played after the current motion is completed 90% playing
----
r100 | 2008-03-16 14:30:18 +0800 (Sun, 16 Mar 2008) | 5 lines

DemoGame/Player.cpp:
- Fixed possible memory leak

DemoGame/StaticModel.cpp:
- Fixed possible memory leak
----
r99 | 2008-03-16 13:59:16 +0800 (Sun, 16 Mar 2008) | 14 lines

DemoGame/Main.cpp:
- Added collision detection support

DemoGame/Player.h
DemoGame/Player.cpp:
- Added one more parameter to "addToScene" for setting the initial position
- Added collision detection support
- Added functions "addHitter", "getTriSelector", "_initCollisionDetection"

```
DemoGame/StaticModel.h
DemoGame/StaticModel.cpp:
- Added one more parameter to "addToScene" for specify using OctTree scene node or not
- Added collision detection support
- Added functions "createTriSelector" and "getTriSelector"
------------------------------------------------------------------------
r98 | 2008-03-15 22:56:14 +0800 (Sat, 15 Mar 2008) | 2 lines

- Fixed a corrupted animation "LeftVDown"
- Added the animation "Stand"
------------------------------------------------------------------------
r97 | 2008-03-15 22:07:18 +0800 (Sat, 15 Mar 2008) | 20 lines

DemoGame/Main.cpp:
- Included Wii Remote support

DemoGame/Player.h
DemoGame/Player.cpp:
- Renamed function from "doAnimation" to "playAnimation"
- Further decease the moving speed
- "playAnimation" will only play the animation once only
- Fixed a bug in interchanged animation
- No default motion is played on the start

WiiStyle/include/DTWButtonOnline.h
WiiStyle/src/DTWButtonOnline.cpp:
- Motion ID is added into the structure "report"

Classifier/DemoGame.txt:
- Classifier for the demo game

MotionDB/DemoGame.txt:
- Motion data of the demo game
------------------------------------------------------------------------
r96 | 2008-03-15 18:51:53 +0800 (Sat, 15 Mar 2008) | 2 lines

WiiDevelopmentTool/FrmMain.h:
- In tab "Motion Register", now able to add motion name by just pressing "Enter"
------------------------------------------------------------------------
```

```
r95 | 2008-03-15 17:52:15 +0800 (Sat, 15 Mar 2008) | 16 lines

DemoGame/DemoGame.vcproj:
- Excluded "wiimote.cpp"

DemoGame/Main.cpp:
- Added Tomb Raider like player movement support
- Updated due to changes in "ThirdPersonCamera"

DemoGame/Player.h
DemoGame/Player.cpp:
- Added a function "moveRelativeTo"
- Added Tomb Raider like player movement support

DemoGame/ThirdPersonCamera.h
DemoGame/ThirdPersonCamera.cpp:
- Added functions "setTargetOffset" and getNode"
- Revised the arguments pass to function "addToScene" and "attachToTarget" to avoid hard coding of the value into the class
------------------------------------------------------------------------
r94 | 2008-03-15 13:41:32 +0800 (Sat, 15 Mar 2008) | 5 lines

DemoGame/Player.h
DemoGame/Player.cpp:
- Added a function "doAnimation"
- Fixed incorrect initial facing direction
- Fixed incorrect moving direction
------------------------------------------------------------------------
r93 | 2008-03-15 13:20:12 +0800 (Sat, 15 Mar 2008) | 9 lines

DemoGame/Main.cpp:
- Tidied code

DemoGame/ThirdPersonCamera.cpp:
- Enhanced view angle of the camera

DemoGame/StaticModel.h
DemoGame/StaticModel.cpp:
- A new class represent a model without animation
```

```
------------------------------------------------------------------------
r92 | 2008-03-15 12:14:39 +0800 (Sat, 15 Mar 2008) | 10 lines

DemoGame/InputReceiver.cpp:
- Fixed a bug in returning job done for undone job

DemoGame/Main.cpp:
- Added camera to follow the player

DemoGame/Player.h
DemoGame/Player.cpp:
- Added a function "getNode"
- Reduced the moving and rotate speed
------------------------------------------------------------------------
r91 | 2008-03-15 11:49:18 +0800 (Sat, 15 Mar 2008) | 2 lines

DemoGame:
- Continuing implementation
------------------------------------------------------------------------
r90 | 2008-03-15 10:19:10 +0800 (Sat, 15 Mar 2008) | 2 lines

DemoGame:
- Imported with Alex work
------------------------------------------------------------------------
r89 | 2008-03-09 00:09:00 +0800 (Sun, 09 Mar 2008) | 6 lines

WiiToONIMUSHA3/Keyboard.h
WiiToONIMUSHA3/Keyboard.cpp:
- Added support for left control key

WiiToONIMUSHA3/Main.cpp:
- Added support for game enhancement when Wii Nunchuk is connected
------------------------------------------------------------------------
r88 | 2008-03-08 21:36:12 +0800 (Sat, 08 Mar 2008) | 2 lines

WiiToONIMUSHA3/Main.cpp:
- Added the detection of guard motion
------------------------------------------------------------------------
```

r87 | 2008-03-08 21:27:38 +0800 (Sat, 08 Mar 2008) | 26 lines

WiiStyle/include/DTWButtonOnline.h:
- Fixed the displayed graph is not the same after resize problem

WiiDevelopmentTool/FrmMain.h:
- After registered new motion, the text will be cleared in the input text box

WiiStyle/include/DTWButtonOnline.h:
- Commented useless functions

WiiStyle/include/Motion.h:
- Added multiple processed data (average data) support

WiiStyle/include/DTWButtonOnline.cpp:
- Added multiple average data support
- Fixed a bug due to incorrect index in function "startClassify"
- Changed the calculation method of the error for higher accuracy under multiple average data

WiiStyle/src/DTWClassifier.cpp:
- Modified function "generateAverageOfThree" for supporting multiple average data

WiiToONIMUSHA3/Keyboard.h
WiiToONIMUSHA3/Keyboard.cpp:
- Added one more argument "forced" in function "keyDown" to specify if the key down event is forced to happen or not

WiiToONIMUSHA3/Main.cpp:
- Added a force key down so that guard in the game will perform in a better way
----
r86 | 2008-03-08 19:46:49 +0800 (Sat, 08 Mar 2008) | 2 lines

WiiStyle/src/DTWButtonOnline.cpp:
- Implemented another version of recording procedure, but commented
----

r85 | 2008-03-08 14:43:44 +0800 (Sat, 08 Mar 2008) | 2 lines

WiiDevelopmentTool/FrmMain.h:
- Fixed a bug in "Test Classifier" where the loop is run continuously instead of having a time interval

----
r84 | 2008-03-08 13:07:15 +0800 (Sat, 08 Mar 2008) | 7 lines

WiiStyle/include/DTWButtonOnline.h
WiiStyle/src/DTWButtonOnline.cpp:
- Added a function "getCurrentState" which return the previous motion in the FSM
- Fixed a bug in not reseting the current state and classified motion ID when calling "_cleanDatabase"

WiiToONIMUSHA3/Main.cpp:
- Included motion timing checking which ensure current motion is performed within the time limit

----
r83 | 2008-03-07 12:01:56 +0800 (Fri, 07 Mar 2008) | 8 lines

WiiStyle/include/DTWButtonOnline.h
WiiStyle/src/DTWButtonOnline.cpp:
- Fixed a bug when the motion FSM is not generated correctly when calling "buildClassifier"
- Added one argument "autoReset" to function "proceed" to auto reset FSM when "_motionToClassify" is empty
- Added a function "_motionID" for getting the motion ID given a motion name

WiiToONIMUSHA3:
- New project which try to play the game "ONIMUSHA 3" using Wii Remote

----

```
r82 | 2008-03-06 23:26:19 +0800 (Thu, 06 Mar 2008) | 13 lines

MotionDB/Dao.txt
MotionDB/Direction.txt
MotionDB/DMC3.txt
MotionDB/Number.txt:
- Updated database file format to version 0.2

WiiDevelopmentTool/FrmMain.h:
- Added startup motions support

WiiStyle/include/MotionDB.h
WiiStyle/src/MotionDB.cpp:
- Added startup motions support
- Changed the database file format to version 0.2
------------------------------------------------------------------------
r81 | 2008-03-06 11:53:50 +0800 (Thu, 06 Mar 2008) | 2 lines

WiiStyle/src/Motion.cpp:
- Fixed a bug in not saving the names of followed motions
------------------------------------------------------------------------
r80 | 2008-03-06 11:09:27 +0800 (Thu, 06 Mar 2008) | 7 lines

WiiDevelopmentTool/FrmMain.h:
- Implemented loading and saving motion FSM in tab "Motion FSM"
- Changed the error message to display "Exclamation" instead of "Information"

WiiStyle/include/Motion.h
WiiStyle/src/Motion.cpp:
- Renamed function from "addFollowedMotions" to "addFollowedMotion"
------------------------------------------------------------------------
r79 | 2008-03-06 10:35:25 +0800 (Thu, 06 Mar 2008) | 2 lines

WiiDevelopmentTool/FrmMain.h:
- Reorganized code
------------------------------------------------------------------------
```

```
r78 | 2008-03-06 10:29:36 +0800 (Thu, 06 Mar 2008) | 10 lines

WiiStyle:
- Included followed motion support in motion database
- Motion database version included in motion database file

MotionDB/Dao.txt
MotionDB/Direction.txt
MotionDB/DMC3.txt
MotionDB/Number.txt:
- Updated file format

------------------------------------------------------------------------
r77 | 2008-03-06 01:34:44 +0800 (Thu, 06 Mar 2008) | 2 lines

WiiDevelopmentTool/FrmMain:
- Implementing GUI for motion FSM
------------------------------------------------------------------------
r76 | 2008-03-05 22:44:42 +0800 (Wed, 05 Mar 2008) | 6 lines

WiiDevelopmentTool/FrmMain.h:
- Added the display of number of motions in the classifier

WiiStyle/include/DTWButtonOnline.h
WiiStyle/src/DTWButtonOnline.cpp:
- Added function "getNumOfMotions"
------------------------------------------------------------------------
r75 | 2008-03-05 22:22:30 +0800 (Wed, 05 Mar 2008) | 3 lines

WiiStyle/include/DTWButtonOnline.h
WiiStyle/src/DTWButtonOnline.cpp:
- Implemented motion FSM
------------------------------------------------------------------------
```

```
r74 | 2008-03-05 20:05:49 +0800 (Wed, 05 Mar 2008) | 7 lines

WiiDevelopmentTool/FrmMain.h:
- Minor change to GUI

WiiStyle/include/DTWButtonOnline.h
WiiStyle/src/DTWButtonOnline.cpp:
- Implemented part of the motion FSM
- Updated the database version to 0.2.1 due to the motion FSM
------------------------------------------------------------------------
r73 | 2008-03-05 19:23:49 +0800 (Wed, 05 Mar 2008) | 7 lines

WiiStyle/include/DTWButtonOnline.h
WiiStyle/src/DTWButtonOnline.cpp:
- Replaced function "resetClassifier" by "_cleanDatabase"
- Fixed memory leak when calling "buildClassifier" and "loadState"
- Implemented function "getAllPrevClassified"
- Added support of multiple average data
- Updated the database version to 0.2 due to the support of multiple average data
------------------------------------------------------------------------
r72 | 2008-02-26 22:48:12 +0800 (Tue, 26 Feb 2008) | 19 lines

- Reorganized the solution structure
- Useless "DownloadedSource" are removed
- Updated WiiYourself! library from 0.96b to 1.0

WiiDevelopmentTool/FrmMain.h:
- Fixed a warning

WiiYourself!Demo:
- Updated the demo to using 1.0 WiiYourself! library

WiiStyle:
- A new static library project which generate our classifier in library form

Classifier:
- A folder which store the classifier data
```

```
MotionDB:
- A folder which contain all the motion database files

------------------------------------------------------------------------
r71 | 2008-02-26 22:37:49 +0800 (Tue, 26 Feb 2008) | 1 line

- Initial folder import
------------------------------------------------------------------------
r70 | 2008-02-26 03:40:41 +0800 (Tue, 26 Feb 2008) | 5 lines

DemoGame/Main.cpp:
- Fixed a warning

WiiDevelopmentTool:
- Added data curve viewer
------------------------------------------------------------------------
r69 | 2008-02-26 02:21:04 +0800 (Tue, 26 Feb 2008) | 2 lines

WiiDevelopmentTool:
- Added limited support for Nunchuk
------------------------------------------------------------------------
r68 | 2008-02-26 01:24:31 +0800 (Tue, 26 Feb 2008) | 12 lines

WiiDevelopmentTool/Classifier/DTWButtonOnline.cpp:
- Fixed a bug in not saving the motion name when saving classifier
- Fixed a bug in not closing the file after saved classifier
- Fixed a bug in reporting wrong success result when saving classifier
- Fixed a bug in not able to load classifier due to the line feed character in the version
- Fixed a bug in corrupted motion name after loaded the classifier due to the line feed character

WiiDevelopmentTool/FrmMain.h:
- Added loading and saving of the classifier

WiiToDMC3:
- A new project that use Wii Remote in playing Devil May Cry 3 Special Edition
------------------------------------------------------------------------
```

```
r67 | 2008-02-26 01:16:05 +0800 (Tue, 26 Feb 2008) | 1 line

- Initial folder structure import
------------------------------------------------------------------------
r66 | 2008-02-19 02:00:09 +0800 (Tue, 19 Feb 2008) | 4 lines

WiiDevelopmentTool/FrmMain:
- Motion will be deleted in "Motion Register" according to the selected row in the table, rather than the input in the text box now
- Confirmation is added when deleting motion
- Added error and similarity output in "Test Classifier"
------------------------------------------------------------------------
r65 | 2008-02-18 23:59:48 +0800 (Mon, 18 Feb 2008) | 3 lines

WiiDevelopmentTool:
- Completed implementing the prototype of the Wiimote Development Tool
- Wiimote report type is set in "btmConnect_Click" instead of "btmRecord_Click" now
------------------------------------------------------------------------
r64 | 2008-02-18 17:28:56 +0800 (Mon, 18 Feb 2008) | 8 lines

WiiDevelopmentTool/FrmMain.h:
- Implemented tab "Data Management"
- Added "New Database", "Exit" command in the menu
- Fixed a bug in application crash when deleting last left motion
- Added "About" dialog in "Help" menu
- Fixed a bug in the recorded data where the time is not start from zero
- Fixed a bug where motion name can be empty
- Added warning when exiting program when Wiimote is connected
------------------------------------------------------------------------
r63 | 2008-02-18 14:06:55 +0800 (Mon, 18 Feb 2008) | 5 lines

WiiDevelopmentTool:
- Updated WiiYourself! library to 1.0
- Renamed tab from "Data Record" to "Data Management"
- Continued implement "Data Management" tab
- Used class "ES_FlashTimer" for timing in recording motion
------------------------------------------------------------------------
```

```
r62 | 2008-02-05 00:28:11 +0800 (Tue, 05 Feb 2008) | 2 lines

DemoGame/Main.cpp:
- Successfully included wiimote button in the demo
------------------------------------------------------------------------
r61 | 2008-02-05 00:05:46 +0800 (Tue, 05 Feb 2008) | 3 lines

- Added new project "DemoGame"
- Removed "DownloadedSource/OpenGL" as it is not used
- Added new source "DownloadedSource/Irrlicht 1.4"
------------------------------------------------------------------------
r60 | 2008-02-05 00:02:06 +0800 (Tue, 05 Feb 2008) | 1 line

- Initial folder structure import for project "DemoGame"
------------------------------------------------------------------------
r59 | 2008-02-04 22:24:20 +0800 (Mon, 04 Feb 2008) | 4 lines

DTWButtonOnline.cpp:
- Implemented "getPrevClassified"
- Implemented "saveState"
- Added database version checking in "loadState"
------------------------------------------------------------------------
r58 | 2008-02-04 22:06:56 +0800 (Mon, 04 Feb 2008) | 2 lines

WiiDevelopmentTool/FrmMain.h:
- Continued implementation
------------------------------------------------------------------------
r57 | 2008-02-04 17:25:09 +0800 (Mon, 04 Feb 2008) | 2 lines

WiiDevelopmentTool/FrmMain.h:
- Continued in implementing GUI
------------------------------------------------------------------------
r56 | 2008-02-04 12:33:54 +0800 (Mon, 04 Feb 2008) | 9 lines

- Added Wiimote source code into project "WiiDevelopmentTool"

NewClassifier/Motion.h
NewClassifier/Motion.cpp:
- Useless variable "_processedData" removed
```

```
WiiDevelopmentTool/Classifier/Motion.h
WiiDevelopmentTool/Classifier/Motion.cpp:
- Useless variable "_processedData" removed
------------------------------------------------------------------------
r55 | 2008-02-04 12:04:07 +0800 (Mon, 04 Feb 2008) | 1 line

- New project "WiiDevelopmentTool" is added
------------------------------------------------------------------------
r54 | 2008-02-04 12:02:30 +0800 (Mon, 04 Feb 2008) | 1 line

- Import folder structure
------------------------------------------------------------------------
r53 | 2008-02-04 11:53:16 +0800 (Mon, 04 Feb 2008) | 2 lines

WiiDataQueue.h:
- Fixed a compilation error because of not updating the header file
------------------------------------------------------------------------
r52 | 2008-02-04 11:48:45 +0800 (Mon, 04 Feb 2008) | 2 lines

WiiDataQueue.cpp:
- Fixed a warning because of signed and unsigned comparison
------------------------------------------------------------------------
r51 | 2008-02-03 22:53:37 +0800 (Sun, 03 Feb 2008) | 6 lines

NewClassifier/DTWButtonOnline.h
NewClassifier/DTWButtonOnline.cpp:
- Implemented "buildClassifier"
- Added modifier "const" to argument in functions "saveState" and "loadState"
- Changed variable "_db" to non-pointer
- Added initialization of "_recordedData"
------------------------------------------------------------------------
r50 | 2008-02-03 22:01:39 +0800 (Sun, 03 Feb 2008) | 9 lines

NewClassifier/DTWButtonOnline.h
NewClassifier/DTWButtonOnline.cpp:
- Variable "col" in structure "_motionInfo" is changed to pointer type
- Instance variable "_recordedData" is added
- A function "_getThreeDimensionDistance" is added
```

```
NewClassifier/DTWColumn.h
NewClassifier/DTWColumn.cpp:
- The model list now only store as reference instead of data
------------------------------------------------------------------------
r49 | 2008-02-03 19:13:36 +0800 (Sun, 03 Feb 2008) | 6 lines

NewClassifier/DTWButtonOnline.h
NewClassifier/DTWButtonOnline.cpp:
- Changed the return type to "void" for functions "startClassify" and "endClassify"
- Added one variable "_recordThread" for holding the handle of recording thread
- Added one class function "_recordingProc" for the recording thread function
- Implemented some functions in the class
------------------------------------------------------------------------
r48 | 2008-01-29 01:03:39 +0800 (Tue, 29 Jan 2008) | 6 lines

DTWButtonOnline.h
DTWButtonOnline.cpp:
- Added one new function "isClassifying"
- Added one new instance variable "_isClassifying"
- Renamed the instance variable from "wii" to "_wii"
- Added missed initialization of the variables in the constructor
------------------------------------------------------------------------
r47 | 2008-01-29 00:56:34 +0800 (Tue, 29 Jan 2008) | 2 lines

DTWButtonOnline.h:
- Added the missing argument name
------------------------------------------------------------------------
r46 | 2008-01-29 00:50:08 +0800 (Tue, 29 Jan 2008) | 3 lines

DTWButtonOnline.h
DTWButtonOnline.cpp:
- A new class which represents the the DTW button online classifier
------------------------------------------------------------------------
r45 | 2008-01-28 21:55:10 +0800 (Mon, 28 Jan 2008) | 5 lines

- Namespace "Motions::Classify" is renamed to "Motion::Classifier""

- All DTW related class are put under namespace "DTW" in "Motions::Classifier"
- "Mathematics.h" and "Mathematics.cpp" are deleted, as no longer in use
------------------------------------------------------------------------
r44 | 2008-01-20 19:59:31 +0800 (Sun, 20 Jan 2008) | 8 lines

NewClassifier/DTWOnline.cpp:
- Removed the code for improper values checking in "_preClassify" since it seems not effective
- Added display of the x-acceleration value when time out happen in "_preClassify"

NewClassifier/MotionClassify.h
NewClassifier/MotionClassify.cpp:
- Updated function names
- Updated menu to display all classification
------------------------------------------------------------------------
r43 | 2008-01-20 19:39:04 +0800 (Sun, 20 Jan 2008) | 7 lines

NewClassifier/DTWClassifier.cpp:
- Enabled the flag "USE_NORMALIZE"

NewClassifier/DTWOnline.cpp:
- The average data for classification now use the average of the first three data in the database
- Added code in "_preClassify" to ensure no improper value from Wii Remote
- "_preClassify" will now focus on marking the end of the motion first
------------------------------------------------------------------------
r42 | 2008-01-20 15:20:31 +0800 (Sun, 20 Jan 2008) | 10 lines

NewClassifier/DTWColumn.h
NewClassifier/DTWColumn.cpp:
- Function name "getValueAtMIdx" changed to "getValueAtTop"

NewClassifier/DTWOnline.cpp:
- Implemented the prototype of buttonless classification

NewClassifier/WiiDataQueue.h
NewClassifier/WiiDataQueue.cpp:
- Added the function "popUntil"
------------------------------------------------------------------------
```

```
r41 | 2008-01-20 15:15:24 +0800 (Sun, 20 Jan 2008) | 1 line

- Added the folder "branches"
------------------------------------------------------------------------
r40 | 2008-01-20 11:29:49 +0800 (Sun, 20 Jan 2008) | 16 lines

NewClassifier/DTWClassifier.cpp:
- Changed the time calculation, yet the proportion and the result is the same

NewClassifier/DTWOnline.h
NewClassifier/DTWOnline.cpp:
- Added a function "_preClassify" for pre-classification of the motions before going to the actual classification

NewClassifier/MotionClassify.cpp:
- "On the Fly" mode now using "_preClassify"

NewClassifier/WiiDataList.h:
- Changed the variable "_data" from private to protected

NewClassifier/WiiDataQueue.h
NewClassifier/WiiDataQueue.cpp:
- A new class representating a queue, which is inherited from "WiiDataList"
------------------------------------------------------------------------
r39 | 2008-01-14 11:52:43 +0800 (Mon, 14 Jan 2008) | 2 lines

NewClassifier/DTWOnline.cpp:
- Commented the code for printing information
------------------------------------------------------------------------
```

```
r38 | 2008-01-14 11:39:10 +0800 (Mon, 14 Jan 2008) | 10 lines

NewClassifier/DTWClassifier.cpp:
- Added the MARCO to hide the useless code when "WITH_SUB_ERROR" is not defined
- Added a commented code for printing the DTWMatrix

NewClassifier/DTWColumn.cpp:
- Removed useless code in function "iterate"
- Function "getValueAtMIdx" is changed to increase the classification rate

NewClassifier/DTWOnline.cpp:
- Commented code is added for print out the DTW Column
------------------------------------------------------------------------
r37 | 2008-01-13 23:20:44 +0800 (Sun, 13 Jan 2008) | 8 lines

NewClassifier/DTWColumn.h
NewClassifier/DTWColumn.cpp:
- Data are now stored in two arrays instead of two vector
- Parameter of "(*_getThreeDimensionDistance)" is changed to pass by reference to enhance speed

NewClassifier/DTWOnline.h
NewClassifier/DTWOnline.cpp:
- Parameters of "_getThreeDimensionDistance" is changed to pass by reference to enhance speed
------------------------------------------------------------------------
r36 | 2008-01-13 22:20:14 +0800 (Sun, 13 Jan 2008) | 8 lines

- Added "const" into instance functions that will not change the data of the instance

NewClassifier/DTWColumn.h
NewClassifier/DTWColumn.cpp:
- Rewrote the iteration and related methods by using two buffers

NewClassifier/MotionClassify.cpp:
- Benchmarks will now output the time used
------------------------------------------------------------------------
```

```
r35 | 2008-01-13 18:25:41 +0800 (Sun, 13 Jan 2008) | 11 lines

NewClassifier/DTWColumn.h
NewClassifier/DTWColumn.cpp:
- Added a function "toString"
- Fixed a bug in calculating the length of the time

NewClassifier/DTWOnline.h
NewClassifier/DTWOnline.cpp:
- Added a function "benchmark" for online classification

NewClassifier/MotionClassify.cpp:
- Included benchmark on online classification in the menu
------------------------------------------------------------------------
r34 | 2008-01-10 00:09:59 +0800 (Thu, 10 Jan 2008) | 7 lines

- DTW online mode implementation completed, not yet debug

NewClassifier/WiiDataList.cpp:
- Removed unused include and define MARCO

UnusedAttribute.txt:
- Removed as no longer in use
------------------------------------------------------------------------
r33 | 2008-01-09 16:13:52 +0800 (Wed, 09 Jan 2008) | 2 lines

- Removed the old DTWLMatrix design
- Added the new DTWColumn design template
------------------------------------------------------------------------
r32 | 2008-01-09 15:29:58 +0800 (Wed, 09 Jan 2008) | 1 line

- Integrated the on the fly classification. However, since the design of using DTWLMatrix is somehow inappropriate, a new design is required. Thus in this revision it only serve as an record of the DTWLMatrix
------------------------------------------------------------------------
```

```
r31 | 2008-01-07 16:00:05 +0800 (Mon, 07 Jan 2008) | 16 lines

NewClassifier/DTWClassifier.cpp:
- Fixed a compilation warning due to truncation of int to float

NewClassifier/Motion.cpp:
- Fixed a compilation warning due to comparison between unsigned int and int

NewClassifier/DTWLMatrix.h
NewClassifier/DTWLMatrix.cpp:
- A new class representing the L shape row & column structure in the DTW matrix

NewClassifier/OnTheFly.cpp:
- Code for on the fly DTW classification, not yet complete

NewClassifier/WiiDataQueue.h
NewClassifier/WiiDataQueue.cpp:
- A new class representing a queue on data frame
------------------------------------------------------------------------
r30 | 2008-01-06 21:19:44 +0800 (Sun, 06 Jan 2008) | 1 line

- Removed old and unused projects and source code
------------------------------------------------------------------------
r29 | 2008-01-06 21:09:23 +0800 (Sun, 06 Jan 2008) | 1 line

- Moved all files to upper level
------------------------------------------------------------------------
r28 | 2007-11-29 23:37:46 +0800 (Thu, 29 Nov 2007) | 1 line

This is the code used in the first term presentation.
------------------------------------------------------------------------
```

```
r27 | 2007-11-26 21:07:08 +0800 (Mon, 26 Nov 2007) | 7 lines

NewClassifier/Motion.h:
- Instance variable "NormalizedData" is renamed to "AverageData"

NewClassifier/DTWClassifier.h
NewClassifier/DTWClassifier.cpp:
- Changes are made to cope with the rename of the instance variable renaming
- Useless part of the functions declaration are removed in the header file
------------------------------------------------------------------------
r26 | 2007-11-26 11:59:40 +0800 (Mon, 26 Nov 2007) | 7 lines

NewClassifier/DTWClassifier.cpp:
- A bug was discovered in function "_getDTWMapping". Backward induction should be used instead of forward induction
- "_getDTWMapping" will return the mapping of the second WiiDataList to the first WiiDataList
- Flags "WITH_DIV_TIME" and "WITH_DIV_ERROR" are removed

NewClassifier/MotionClassify.cpp:
- The default "NUM_OF_SAMPLE" is changed to 5
------------------------------------------------------------------------
r25 | 2007-11-21 20:59:49 +0800 (Wed, 21 Nov 2007) | 2 lines

NewClassifier/DTWClassifier.cpp:
- Fixed a bug in "prepareDatabase" where "sample[0]" is not the longest raw data when "_maxDuration" return "C"
------------------------------------------------------------------------
r24 | 2007-11-21 20:33:59 +0800 (Wed, 21 Nov 2007) | 7 lines

NewClassifier/DTWClassifier.h
NewClassifier/DTWClassifier.cpp:
- Functions "_maxDuration" and "_getDTWMapping" are added back for generating the average data
- A flag "USE_NORMALIZE" is added, when it is on the classification will compare the result with the average data. "prepareDatabase" is modified to cope with the new classification method.
```

NewClassifier/Motion.h:
- Instance variable "NormalizedData" is added back to cope with the new classification method.
----
r23 | 2007-11-20 02:01:33 +0800 (Tue, 20 Nov 2007) | 2 lines

NewClassifier/MotionClassify.cpp:
- Added a definition "NUM_OF_SAMPLE" which specify the number of sample used in classifier. Mainly used for benchmark of different configuration
----
r22 | 2007-11-19 23:39:16 +0800 (Mon, 19 Nov 2007) | 4 lines

NewClassifier/MotionClassify.h
NewClassifier/MotionClassify.cpp:
- Removed the parameter "wii" in "dtwBenchmark" as it is useless
- New function "dtwFileBenchmark" which able to take the data in a file for classification with existing motion database
----
r21 | 2007-11-19 23:02:12 +0800 (Mon, 19 Nov 2007) | 14 lines

NewClassifier/DTWClassifier.h
NewClassifier/DTWClassifier.cpp:
- Functions "_getXDistance", "_getYDistance", "_getZDistance" and "_generateReport" are removed as no longer used
- Functions "_getThreeDimensionDistance" and "_generateError" are added
- Code rewritten. Classifier now based on the 3-D distance for error generation. Dynamic time warping applied on separated 3 axes of acceleration is logically incorrect. Since the mapping of the point on each axis may not be consistence with other axis
- Defined different flag for benchmark with different settings

NewClassifier/ErrorReport.h
NewClassifier/ErrorReport.cpp:
- Deleted as the classifier no longer based on 3 axes of acceleration

NewClassifier/Motion.h:
- Instance variables "AverageError" and "NormalizedData" are removed as the classifier no longer based on 3 axes of acceleration
- Instance variable "AverageError" of type "float" added to record the error based on 3-D distance error generation

```
------------------------------------------------------------------------
r20 | 2007-11-19 19:29:48 +0800 (Mon, 19 Nov 2007) | 27 lines

NewClassifier/DBOperation.h
NewClassifier/DBOperation.cpp:
- Commented functions "getThreeMotion" and "recordThreeMotion" are removed
- Commented code in "registerNewMotion" in removed
- Fixed a bug on displaying second and third recorded motions
- Fixed a bug on re-recording second and third recored motions

NewClassifier/DTWClassifier.h
NewClassifier/DTWClassifier.cpp:
- Functions "_maxDuration" and "getDTWMapping" are removed
- Functions "_getXAngleDistance", "_getYAngleDistance", "_getZAngleDistance", "_getCombinedAccelerationDistance" are removed
- Commented code in "prepareDatabase" is removed
- Commented code in "classify" is removed
- Commented code in "_generateReport" is removed

NewClassifier/ErrorReport.h
NewClassifier/ErrorReport.cpp:
- Instance variables "errorAngleX", "errorAngleY", "errorAngleZ", "errorCombinedAcceleration" are removed.
- Function "sumOfSquareOfAngle" is removed

NewClassifier/Main.cpp:
- "Answer: " will be outputed to screen when ask for confirmation of exiting the program

NewClassifier/WiiDataList.h
NewClassifier/WiiDataList.cpp:
- Structure "polarForm" in structure "wiiDataFrame" is removed
- Function "calculatePolarForm" is removed
------------------------------------------------------------------------
r19 | 2007-11-19 19:03:15 +0800 (Mon, 19 Nov 2007) | 1 line

This is the last revision that contains all the old and useless functions.
------------------------------------------------------------------------
```

```
r18 | 2007-11-19 19:01:48 +0800 (Mon, 19 Nov 2007) | 1 line

Create tags folder.
------------------------------------------------------------------------
r17 | 2007-11-19 18:54:55 +0800 (Mon, 19 Nov 2007) | 7 lines

NewClassifier/DBOperation.h
NewClassifier/DBOperation.cpp:
- Motion recording after register motion name is removed. Related functions not in use are commented.
- "deleteRecordedMotion" missing in header file is added

NewClassifier/Main.cpp:
- Added a confirmation when closing the program
------------------------------------------------------------------------
r16 | 2007-11-19 14:55:36 +0800 (Mon, 19 Nov 2007) | 9 lines

NewClassifier/DTWClassifier.h
NewClassifier/DTWClassifier.cpp:
- New function "_maxDuration" which return which "WiiDataList" has the maximum duration
- New function "_getDTWMapping" which return the mapping of the points in two "WiiDataList"
- Commented code on prototype of normalizing curve

NewClassifier/ErrorReport.h
NewClassifier/ErrorReport.cpp:
- New function "sumOfSquareOfAngle" to calculate the sum of square of the angle error
------------------------------------------------------------------------
r15 | 2007-11-18 18:31:38 +0800 (Sun, 18 Nov 2007) | 6 lines

NewClassifier/DTWClassifier.h
NewClassifier/DTWClassifier.cpp:
- "_dTWDistance" renamed to "_dtwDistance"
- New function "_dtwMatrix", which able to return the dynamic time warping matrix
- New function "_freeDTWMatrix", which able to free the memory used by the dynamic time warping matrix
- "_dtwDistance" modified to use the new functions
------------------------------------------------------------------------
```

r14 | 2007-11-18 18:03:35 +0800 (Sun, 18 Nov 2007) | 23 lines

NewClassifier/DBOperation.h
NewClassifier/DBOperation.cpp:
- Able to display other recorded motions other than first, second and third
- Able to re-record other recorded motions other than first, second and third
- Able to remove recored motions
- New function for getting number from user

NewClassifier/DTWClassifier.h
NewClassifier/DTWClassifier.cpp:
- Fixed memory leak in "_dTWDistance"
- Optional to display details in "classify"

NewClassifier/Main.cpp:
- Not require a connection of wiimote when start the program

NewClassifier/Motion.h:
- Now able to delete raw data

NewClassifier/MotionClassify.cpp:
- Benchmark won't display useless information anymore

NewClassifier/MotionRecorder.cpp:
- "recordSignal" will now connect to the wiimote first before start recording. It will return -1 if failed in connection
----
r13 | 2007-11-18 13:12:13 +0800 (Sun, 18 Nov 2007) | 21 lines

NewClassifier/DBOperation.cpp:
- Now will display the number of raw data when displaying motion details

NewClassifier/DTWClassifier.h
NewClassifier/DTWClassifier.cpp:
- "classify" will now return the name of the classified result

NewClassifier/Main.cpp:
- "DTW Classification" is put into the group "Classification"

NewClassifier/Motion.h:
- Function name "length" is changed to "size"

NewClassifier/MotionClassify.h
NewClassifier/MotionClassify.cpp:
- Missed function declarations are added in header file
- A sub menu function is added
- A benchmark mode is added

NewClassifier/RecordingOperation.cpp:
- "recordSequentially" now will display number of motions recorded

----
r12 | 2007-11-14 13:24:57 +0800 (Wed, 14 Nov 2007) | 9 lines

NewClassifier/Main.cpp:
- Updated to match the new name of functions and files

NewClassifier/SampleOperation.h
NewClassifier/SampleOperation.cpp
NewClassifier/RecordingOperation.h
NewClassifier/RecordingOperation.cpp:
- Files renamed from "SampleOperation" to "RecordingOperation"
- Rewrote the functions so that the recorded motion are stored in the Motion Database in Database Operation

----
r11 | 2007-11-14 02:17:12 +0800 (Wed, 14 Nov 2007) | 20 lines

NewClassifier/DBOperation.cpp:
- Updated for matching the new "Motion" class

NewClassifier/DTWClassifier.cpp:
- Changed the classification criteria which include consideration of duration and over subtraction problem

NewClassifier/ErrorReport.h
NewClassifier/ErrorReport.cpp:
- Added several operator functions

```
NewClassifier/Main.cpp:
- Added and implemented an new option "Record Sample Motions to File"

NewClassifier/Motion.h
NewClassifier/Motion.cpp:
- The storing of WiiDataList is changed from array to vector

NewClassifier/SampleOperation.h
NewClassifier/SampleOperation.cpp:
- New set of functions for the console in doing sample data related job
------------------------------------------------------------------------
r10 | 2007-11-10 17:27:00 +0800 (Sat, 10 Nov 2007) | 24 lines

NewClassifier/DBOperation.cpp:
- Number of motions will be printed when printing motion names

NewClassifier/DTWClassifier.h
NewClassifier/DTWClassifier.cpp:
- New class for classification using dynamic time warping

NewClassifier/ErrorReport.h
NewClassifier/ErrorReport.cpp:
- Now in namespace "Motions.Classify"
- "removeNegative" now will return the result instead of modify its value
- New function "sumOfSquareOfAcceleration"
- New private function "_toZeroIfNegative"

NewClassifier/Main.cpp:
- Classification with DTW now working

NewClassifier/Motion.h
NewClassifier/Motion.cpp:
- Removed "learn" and "calculateError" functions

NewClassifier/MotionClassify.h
NewClassifier/MotionClassify.cpp:
- New set of functions for displaying classification option in console
------------------------------------------------------------------------
```

```
r9 | 2007-11-10 15:41:17 +0800 (Sat, 10 Nov 2007) | 27 lines

NewClassifier/DBOperation.h
NewClassifier/DBOperation.cpp:
- Updated to match the new namespace
- Added confirmation when saving the database

NewClassifier/DTWReport.h
NewClassifier/DTWReport.cpp:
- Deleted as it is no longer in use

NewClassifier/Main.cpp:
- Added an option "Classify"

NewClassifier/Motion.h
NewClassifier/Motion.cpp:
- Modified namespace to "Motions.Data"

NewClassifier/MotionDB.h
NewClassifier/MotionDB.cpp:
- Modified namespace to "Motions.Data"

NewClassifier/MotionRecorder.h
NewClassifier/MotionRecorder.cpp:
- Modified namespace to "Motions.Recording"

NewClassifier/WiiDataList.h
NewClassifier/WiiDataList.cpp:
- Modified namespace to "Motions.Data"
------------------------------------------------------------------------
r8 | 2007-11-10 14:13:09 +0800 (Sat, 10 Nov 2007) | 36 lines

NewClassifier/DBOperation.h
NewClassifier/DBOperation.cpp:
- New set of functions for console control of the motion database

NewClassifier/DTWReport.h:
- Modified for matching the new namespace
```

NewClassifier/Main.cpp:
- Rewrite for controlling the new design of the program

NewClassifier/Mathematics.h
NewClassifier/Mathematics.cpp:
- Modified to fix the warning when compile

NewClassifier/Motion.h
NewClassifier/Motion.cpp:
- Now in new namespace "MotionData"
- Implemented "saveState" and "loadState"

NewClassifier/MotionDB.h
NewClassifier/MotionDB.cpp:
- Now in new namespace "MotionData"
- Implemented "saveState" and "loadState"
- Modified to fix the warning when compile

NewClassifier/MotionRecorder.h
NewClassifier/MotionRecorder.cpp:
- Now in new namespace "MotionRecording"
- Modified to match the new design of "WiiDataList"

NewClassifier/WiiDataList.h
NewClassifier/WiiDataList.cpp:
- Now in new namespace "MotionData"
- Redesigned. Data are stored in a "vector"
- Will not store "name" anymore
- Useless functions are removed

```
r7 | 2007-11-08 13:05:35 +0800 (Thu, 08 Nov 2007) | 27 lines

NewClassifier.vcproj:
- Updated structure of the project

NewClassifier/ErrorReport.h
NewClassifier/ErrorReport.cpp:
- New class for storing error report

NewClassifier/Main.cpp:
- Updated function call name

NewClassifier/Mathematics.h
NewClassifier/Mathematics.cpp:
- Copies from the "DevConsole", with new namespace "Math"

NewClassifier/Motion.h
NewClassifier/Motion.cpp:
- Added a new constructor with parameter "MotionName"
- Added a new variable for storing the motion name

NewClassifier/MotionDB.h
NewClassifier/MotionDB.cpp:
- New class for storing a list of motions

NewClassifier/WiiDataList.h
NewClassifier/WiiDataList.cpp:
- Copies from "DevConsole"

------------------------------------------------------------------------
r6 | 2007-11-07 19:18:48 +0800 (Wed, 07 Nov 2007) | 1 line

Updated SVN Ignore Rule
------------------------------------------------------------------------
r5 | 2007-11-07 19:10:37 +0800 (Wed, 07 Nov 2007) | 2 lines

CS FYP.suo:
Removed as it contain user options only
------------------------------------------------------------------------
```

```
r4 | 2007-11-07 18:15:08 +0800 (Wed, 07 Nov 2007) | 2 lines

NewClassifier/Main.cpp:
- The new classification algorithm able to include 5 sets of motions
------------------------------------------------------------------------
r3 | 2007-11-07 17:40:03 +0800 (Wed, 07 Nov 2007) | 10 lines

NewClassifier/Main.cpp:
- Included the new classification algorithm for testing

NewClassifier/Motion.h:
NewClassifier/Motion.cpp:
- Fixed the operator [] problem
- classify now return the error instead of just yes and no

NewClassifier/MotionRecorder.cpp:
- Fixed the problem of skipping "recordSignal" function call when "recordSignal" are called
sequentially.
------------------------------------------------------------------------
r2 | 2007-11-07 16:19:57 +0800 (Wed, 07 Nov 2007) | 1 line

Updated Ignore Files List.
------------------------------------------------------------------------
r1 | 2007-11-07 16:14:10 +0800 (Wed, 07 Nov 2007) | 1 line

Initial Import
------------------------------------------------------------------------
```