Motion Recognition

Generating real-time feedback based upon movement of a gaming controller

FRANCISCO ALEO MONTEAGUDO

KTH Information and
Communication Technology

Degree project in
Communication Systems
Second level, 30.0 HEC
Stockholm, Sweden

Motion Recognition

Generating real–time feedback based upon movement of a gaming controller

Francisco Aleo Monteagudo
2011-08-25

Examiner: Gerald Q. Maguire Jr
School of Information and Communication Technology
Royal Institute of Technology (KTH)
Stockholm, Sweden

Abstract

Today motion recognition has become popular for human computer interaction in areas, such as health care, computer games, and robotics. Although many research projects have investigated this field, there are still some challenges remaining, especially in real-time environments.

In real-time environments, the amount of data needed to compute the user's motion and the time required to collect and process this data are crucial parameters in the performance of a motion recognition system. Moreover, the nature of the data (accelerometer, gyroscope, camera, . . .) determines the design of the motion recognition system. One of the most important challenges is to reduce the delay between sensing and recognizing the motion, while, at the same time, achieving acceptable levels of accuracy.

In this thesis we present a solution using Nintendo's Wii Remote that solves several problems, such as permitting multiple device interaction and synchronization. In addition, this thesis addresses the performance challenge of realizing motion recognition for such a device. Finally, this thesis introduces a Java architecture which contains a set of interfaces that can be re-used in future projects.

One of the most important achievements of this project is enabling interaction among different users and devices in a real-time environment, as, our application deals with multiple devices at the same time, with an acceptable delay. The resulting application provides smooth interaction to the user. As a consequence, our application enables collaborative and competitive activities which in this thesis project were evaluated in a educational process context. In this specific context, the main goal of the researchers with whom I was collaborating was to extend traditional methods of teaching children about some abstract concepts, such as energy.

In addition, this thesis shows how to achieve different levels of accuracy and performance, by implementing two different algorithms. The first one is a static algorithm based on heuristics. The second algorithm, called k-Means, is based on data clustering. The heuristics based algorithm provides a result in less than 2 milliseconds, while k-Means takes roughly 4 milliseconds to converge. A comparison of the performance and flexibility of these two algorithms is presented.

This project has resulted in a multi-threaded high level architecture based on Java, which enables interaction between Wiimote devices. The Application Programming Interface, can be easily extended for future projects, via several interfaces that provide basic mechanisms, such as an event listener, message delivery, and synchronization module. Moreover, the two different motion recognition algorithms offer different performances and different flexibility features, a crucial parameter closely related with motion recognition accuracy.

Sammanfattning

Idag rörelse erkännande har blivit populär för människa-dator interaktion områden, t.ex. hälsovård, dataspel, och robotik. även om många forskningsprojekt har undersökt detta område finns det fortfarande några utmaningar som återstår, framför allt i realtid miljöer.

I realtid miljöer, behövs den mängd data för att beräkna användarens rörelse och den tid som krävs för att samla in och bearbeta dessa data är avgörande parameter är i utförandet av en rörelse erkännande. Dessutom har typ av data (accelerometer, gyroskop, kamera, ...) bestämmer Utformningen av rörelse erkännande. En av de viktigaste utmaningarna är att minska fördröjningen mellan sensorer och erkänna rörelse, medan vid Samtidigt uppnå en acceptabel nivå av noggrannhet.

I denna avhandling presenterar vi en lösning med Nintendos Wii Remote som löser flera problem, som tillåter flera enheter samspel och synkronisering. Dessutom behandlar denna avhandling prestanda utmaningen förverkliga rörelse erkännande för en sådan enhet. Slutligen, denna avhandling introducerar en Java-arkitektur som innehåller en uppsättning gränssnitt som kan återanvändas i framtida projekt.

En av de viktigaste resultaten av detta projekt gör det möjligt för interaktion mellan olika användare och enheter i realtid miljö som är våra Ansökan handlar om flera enheter på samma gång, med en acceptabel dröjsmål. Den nya ansökan innehåller smidigt samspel med användaren. som en följd av detta gör att vår ansökan samarbete och konkurrens verksamheter som i detta examensarbete utvärderades i en pedagogisk processen sammanhang. I detta specifika sammanhang, det viktigaste målet för forskarna som jag har samarbetat var att utöka traditionella undervisningen barn om några abstrakta begrepp, såsom energi.

Dessutom visar avhandlingen hur man kan uppnå olika nivåer av noggrannhet och prestanda genom att införa två olika algoritmer. Den första är en statisk algoritm baserad på heuristik. Den andra algoritmen, kallade K-medel, är baseras på data klustring. Den heuristik baserad algoritm ger ett resultat i mindre än 2 millisekunder, medan k-Betyder tar ungefär 4 millisekunder att konvergerar. En jämförelse av prestanda och flexibilitet för dessa två algoritmer presenteras.

Detta projekt har resulterat i en flertrådad hög nivå arkitektur baserad på Java, som möjliggör interaktion mellan Wiimote enheter. Ansökan Programming Interface, kan enkelt byggas ut för framtida projekt, via flera gränssnitt som ger grundläggande mekanismer, såsom en händelseavlyssnare meddelande leverans, och synkronisering modul. Dessutom har två olika rörelser erkännande algoritmer erbjuder olika föreställningar och olika flexibilitet funktioner, en avgörande parameter nära besläktad med rörelse erkännande noggrannhet.

Contents

Contents iii

List of Figures v

List of Tables vi

1 Introduction 1

2 Background 5
- 2.1 Mathematical approaches to processing the acceleration data 5
  - 2.1.1 Hidden Markov Model 5
  - 2.1.2 Kalman filter 7
  - 2.1.3 k-Means Clustering 9
- 2.2 Related work 11
  - 2.2.1 HMM based 11
  - 2.2.2 Kalman based 11
- 2.3 System Architecture 12
  - 2.3.1 Nintendo Wii remote device (Wiimote) 12
  - 2.3.2 Human Interface Device 13
  - 2.3.3 Software Architecture 14
- 2.4 WiiuseJ 16
  - 2.4.1 Big Picture - Model Diagram 17
  - 2.4.2 Model-View-Controller Architecture 18
  - 2.4.3 Event Handler System 24

3 Method 29
- 3.1 Project Context 29
  - 3.1.1 Design Process - Workshops 29
  - 3.1.2 The Game 30
  - 3.1.3 Conclusions 32
- 3.2 The architecture from a design point of view 33
  - 3.2.1 Overview - Model Diagram 33
  - 3.2.2 Multithreaded Application 33
  - 3.2.3 Model-View-Controller Architecture 35

3.2.4 Whole Process 43
3.3 Heuristic algorithm 46

4 Analysis 49
4.1 Testing 49
4.2 System Modes 49
4.3 Performance 50
4.3.1 Network Delay 50
4.3.2 Architecture Delay 54
4.3.3 End-to-End Delay 55
4.3.4 Statistical analysis of experimental results regarding correct motion recognition 57
4.4 Data process 59

5 Conclusions and future work 61
5.1 Conclusions 61
5.1.1 Goals and insights 61
5.1.2 Suggestions and hints 62
5.1.3 Modifications 62
5.2 Future work 63
5.2.1 Remaining work and next steps 63

Bibliography 65

List of Figures

2.1 Markov chain 6
2.2 HMM latent variables 6
2.3 Latent variables state transition diagram 7
2.4 Kalman filtering process 9
2.5 k-Means process sample 10
2.6 Human Interface Device stack 13
2.7 Nintendo Wii remote device (Wiimote) 14
2.8 Software Architecture 15
2.9 System architecture shown as a Model View Controller Diagram 17
2.10 Events Class Diagram 18
2.11 Wiimote Event Class Diagram 19
2.12 GUI objects Class Diagram 20
2.13 Graphical User Interface Testing 21
2.14 GUI objects and Listerners Class Diagram 22
2.15 WiimoteApiManager Class Diagram 23
2.16 Wiimote Connection Sequence Diagram 24
2.17 Listener Hierarchy Diagram 25
2.18 Event Handling Sequence Diagram 26
2.19 Event Distribution Sequence Diagram 27

3.1 Attached wiimotes 31
3.2 Architecture - overview 34
3.3 Event Handling classes 38
3.4 Event Handling interfaces 40
3.5 Hardware Controller hierarchy 41
3.6 Software Controller hierarchy 42
3.7 Message Delivery system 43
3.8 Controllers Part 1 44
3.9 Controllers Part 2 45

4.1 Architecture data flow 56

List of Tables

4.1 Network Performance 52
4.2 Network Performance statistics 53
4.3 Packet Lost 53
4.4 Motion recognition algorithms performance (with all times in milliseconds) 54
4.5 Acceleration packet delay in milliseconds 55
4.6 End-to-end delay in milliseconds 57
4.7 Motion recognition efficiency in % 58
4.8 Robotic recognition efficiency interferences in % 58

List of Algorithms

1 Heuristic algorithm 46

Acronyms and Abbreviations

ADPCM Adaptive Differential Pulse Code Modulation

API Application Programming Interface

GUI Graphical User Interface

HCI Human Computer Interaction

HID Human Interface Device

HMM Hidden Markov Model

IMU Inertial Measurement Unit

IR Infra-Red

J2EE Java Enterprise Edition

JNI Java Native Interface

LED Light-Emitting Diode

L2CAP Logical Link Control and Adaptation Protocol

MVC Model-View-Controller

PCM Pulse Code Modulation

RTT Road Trip Time

UML Unified Modeling Language

USB Universal Serial Bus

Wiimote Nintendo Wii remote device

Acknowledgements

I would like to express my appreciation to my supervisor, Gerald Q. Maguire Jr., for providing me an opportunity to conduct my master's research under him and for his guidance and support.

I thank Mobile Life VINN Excellence Center, which gave me the opportunity to be part of this research.

I also wish to thank to all my friends, who helped me get through this thesis and my masther's degree.

The most special thanks goes to my family, specially, to my brother Victor. He gave me his unconditional support through all this long process.

Chapter 1

Introduction

Motion recognition has become popular in recent years. A great deal of research effort has been conducted in this field, especially in Human Computer Interaction (HCI), where the user is one of the most significant parts of the system. Hence, the possibility to recognize user's gestures, and, consequently enable a interaction based upon gestures is an important challenges today.

The main problem we have to deal with is the definition of a specific gesture, and then, understand how to recognize it. In this project we will recognize gestures based upon the movement of a specific handheld device. Therefore we have to communicate with the sensors that are embedded into this device and from this sensor data recognize the user's gestures.

The context of this work is a Mobile Life Excellence Center research project called Generalized Interaction Models [21], lead by Jakob Tholander. The main focus of the project is adapting the desktop metaphor for the mobile phone, but not simply imitating the models invented for stationary workstations. Additionally, this motion recognition project was done in a collaboration with another research project called Wii Science [8], which is focused on introducing education to learners through the use of computer and video games.

Once the context was defined we started the motion design process. Carolina Johansson was the main motion designer, and consequently, she led this process. In order to define these motions and specify the constraints on these motions, we started with some conclusions from her studies of sports, such as skateboarding, and golf.

We decided that we wanted to deal with three simple and quite different motions. The requirements for those motions were:

- **Full body movement**, so, users could perform any of the motions without restrictions in terms of space, shape, or velocity.

- **Sensing**, so that the user could not cheat by doing a different motion, but obtain the same result. For instance, making smaller motions when big motions are required.
- **Awareness**, defined as the relationship that the user has with a thing when the user is aware of the thing, but the thing itself is not the focus of attention. Therefore, the thing (in this case the device with sensors) should allow to the user to focus on something else.

This criteria was adopted due the nature of the designed experiment, exposed in chapter 3, "Weather Gods and Fruit Kids Game", as well as the collaboration with Wii Science research project [8].

The results of these requirements and the motion design process were the following motions:

Slow&Soft
: The motion should not have high variations in terms of velocity or acceleration. Hence, the motion should be as smooth as possible.

Big&Large
: In this case the user should perform motions with high variations in terms of velocity, i.e., acceleration. The volume of space used is, consequently wider than in previous motion type.

Robotic
: The motions are short and exhibit a high variation in acceleration followed by non-activity for a defined minimum period of time. Therefore, after every motion the user should remain frozen for a short period of time.

The designed motions pretend to be easily distinguishable each from other. However, as it is possible to observe in chapter 4, Robotic motion presents poor motion recognition results. So, the solution would be either to make a deeper analysis in order to detect this lack of motion recognition (and avoid the use of this motion) or implement a more suitable motion recognition algorithm, such as the methods descibed in chapter 2.

After defining these motions, we needed to choose the most suitable device for enabling the interaction with the user. Initially, we thought about two different devices: a mobile phone and Nintendo Wii remote device (Wiimote). After evaluation of these alternatives, and an estimate of the time that would be required time to implement the system, we decided to focus on the Wiimote.

The Wiimote is the main interface device we used during a set of workshops [1], in which children were to perform one of the three specific motions described above.

[1]Details of these workshops will be explained in detail in Chapter 4 on page 49

The device has several different peripherals which provide feedback to the users, such as rumble, lights via Light-Emitting Diode (LED), and sound. Moreover, the Wiimote provides 8 bits of acceleration data, in a device centered Cartesian coordinate system (X, Y, and Z axis). The Wiimote communicates via Bluetooth.

In order to understand how this feedback is related to body motion, it is important to explain how the host application collects information from the Wiimote and turns this data into a recognized motion. The Wiimote streams data packets containing acceleration data at 0-100 packets per second. This information is collected by the host and after a selectable period of time (500, 250, or 125 milliseconds), the host application forms a chunk containing all the recently received packets. This chunk is analyzed, by calculating both average and peak values of acceleration for each of the axes. Following this analysis, the host application outputs an estimate of whether the body motion was one of these three motions that the child was to perform: "Slow&Soft", "Big&Large", or "Robotic".

The speaker of the Wiimote provides short and low sound volume effects. The speaker could be configured either as a 4-bit Adaptive Differential Pulse Code Modulation (ADPCM), or 8-bit Pulse Code Modulation (PCM) audio device. Once the device is configured, the sound effects are streamed by a host application, a square wave (click sound) by sending 20 bytes at a time. The application directly couples the sound feedbacks to the application's estimation of the user's body motion.

We have defined a sound, emitted every second, as our basic audio feedback. If the motion was "Slow&Soft" the user will hear only this sound. If the motion was "Big&Large", then clicking sounds were added to the basic sound. In the third case, "Robotic" body motion results in the Wiimote playing 3 consecutive sound effects in a short period of time.

Michael Kantor and David Redmiles have noted that minimizing the total delay between detecting and evaluating the motion and audio feedback enhances the awareness of the user [9]. Therefore with the above audio feedback, the user knows which kind of motion was recognized, just by listening to the sound emitted by the Wiimote.

In addition, the Wiimote utilizes a small motor to make the device rumble. We designed the vibration feedback to indicate whether the user's current body motion was the expected motion. Consequently, a lack of vibration alerts the user that they are not performing the correct motion. The vibrations are short enough to avoid overlap between consecutive device responses, but long enough to properly alert the user.

Similar to audio feedback, vibration feedback is computed over the motion chunks formed by the host. Hence, the user gets either, a vibration or none, depending on the user's body motion during the last sensing interval.

The following chapters will describe the design, implementation, and evaluation of a system which provides feedback in real-time based upon recognition of the user's movements. The system will be evaluated in terms of the design requirements.The initial system architecture is described along with, how it was modified, extended, re-designed, and implemented. Each of the the different modules and layers in the design and implementation, are described in detail in terms of how motion recognition can be performed successfully.

The overall performance of the system is evaluated in terms of the delay between motion and the time of generating feedback; as well as the probability of correctly recognizing the motion. The thesis ends with a statement of some conclusions and recommendations for new techniques to improve the results and alternative ways to compare motions.

Chapter 2

Background

This chapter presents all the mathematical concepts needed in order to understand the related projects and the algorithm implemented in this thesis project. Once these methods are presented several related projects are described. Some of these studies are closely related with my work, while some other projects described common methods that could be applied in future work following this thesis project. Finally, the architecture used to fulfill the design requirements of the research study is described in detail.

2.1 Mathematical approaches to processing the acceleration data

This section presents all the mathematical models that are subsequently used to process the acceleration data, necessary in order to understand section 2.2, and to understand the algorithm that was implemented to recognize motion.

2.1.1 Hidden Markov Model

2.1.1.1 Markov Model

A Markov Model is random process that fulfills the Markovian property. This property says that the current state of the model is independent of all previous observations (X_n) except the most recent [1]. Therefore, the next state (i.e., the future state) will depend only on the current state (i.e., the present state), not on the past states. Equation 2.1 presents the Markovian property as a conditional probability function.

$$p(X_1, ..., X_n) = \prod_{n=1}^{N} (X_n \mid X_1, ..., X_{n-1}) \tag{2.1}$$

If we add the constraint of a discrete−time process, we obtain a particular case of the Markov Model known as a Markov Chain. It is possible to characterize a Markov Chain as a set of either finite or non−finite states (random variables), where the changes among states are known as transitions. The set of states is known as state−space and the probability of those transitions are called transition probabilities. Figure 2.1.2 illustrates a fine Markov chain with three states and their transitions.

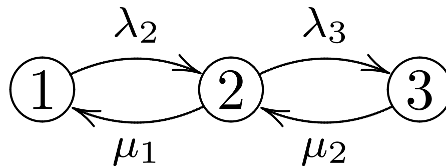

Figure 2.1. Markov chain

2.1.1.2 Hidden Markov Model

A Hidden Markov Model (HMM) is an specific case of the Markov Model that introduces the concept of latent variables. These latent variables are used to represent a specific instance of the state space. Figure 2.2 illustrates the relationship between latent variables and system states.

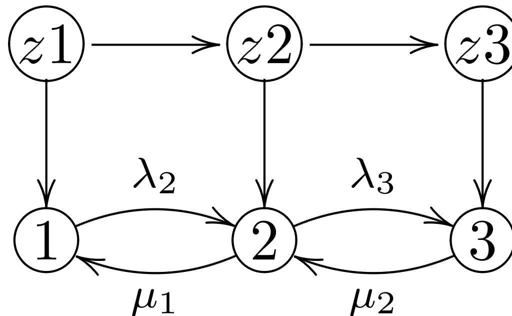

Figure 2.2. HMM latent variables

Latent variables are discrete and multinomial variables, Z_n, and describe which component is responsible for generating the corresponding observation X_n [1]. These latent variables depend on the previous state of the latent variables, through the same relationship as in a Markov Model, i.e., conditional probability. Figure 2.3 shows the possible states of a latent variable.

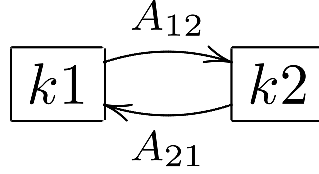

Figure 2.3. Latent variables state transition diagram

In order to characterize a HMM we need to introduce the concept of emission probability (ϕ), which is the set of parameters that governs the conditional probabilities of the observed variables. In orther words, emission probabilities are the K possible states of the binary vector Z_n [1]. Equation 2.2 shows the conditional probability of a HMM.

$$p(Z_n \mid Z_{n-1,A}) = \prod_{k=1}^{K} \prod_{j=1}^{K} A_{j,k}^{Z_{n-1},jZ_{nk}} \tag{2.2}$$

2.1.2 Kalman filter

A Kalman filter is a set of equations that recursively updates and estimates the state of the system. This process takes into account past measurements and present state, in order to predict future system states. Moreover, it stores the errors and deviation between the measurements and the estimated states in order to predict future states. This is done by a set of matrix calculations. The matrix represents the state of the system. The following matrices are used in the different steps of realizing a Kalman filter:

Measurement vector (Z) contains the measurements of the system.
For instance, positions, accelerastion, velocity, ...

State vector (X) states of all measured components, and the first derivative of those components.

Covariance matrix (P) contains the errors produced in state vector estimations.

Measurement error (R) stores the random errors produced by the measurement equipment. Usually, the values in this vector are hard coded based on specific values of the used sensors that are used.

A Kalman filter is divided in two different phases: *Prediction* and *Correction* [25]. During the *Prediction* phase two predictions are made:

System state is the reponisble for estimating the future state of the system. It basically propagates the state vector (X) by predicting the future state with regard to a time propagator (Φ). See equation 2.3.

$$X_{predicted} = \Phi(\triangle t)X(t) \tag{2.3}$$

Error covariance propagates a covariance matrix (Z), which contains the error uncertainties, regarding the previous covariance matrix (P^{+}_{k-1}) and the same time propagator used in system state prediction (Φ). Equation 2.4 presents the covariance matrix propagation.

$$P^{-}_{k} = \Phi P^{+}_{k-1}\Phi^{T} \tag{2.4}$$

Before computing the *Correction* steps the Kalman filter computes the innovation (N), which is the difference between the new measurement and the filter's prediction. In order to compute innovation an auxiliary measurement matrix (H) is needed, due to the difference in dimensions between state vector (X) and measurements (Z). See equation 2.5.

$$N = Z - HX_{predicted} \tag{2.5}$$

Once the innovation is computed, it is possible to perform the three different steps which belongs to the *Correction* phase:

Kalman gain (K) indicates how much of the innovation should be applied to the estimation [6]. See equation 2.6.

$$K = P^{-}H^{T}(HP^{-} + R)^{-1} \tag{2.6}$$

State vector updating converts the current estimation into the measurements of the system. This step is performed after a new measurement is received. See equation 2.7

$$X^{+} = X_{predicted} + KN \tag{2.7}$$

Error estimate updating is triggered after a new measurement arrives. The update process is based on identity matrix (I), auxiliary measurement matrix (H), Kalman gain (K), and the current covariance error matrix. See equation 2.8.

$$P^{+} = (I - KH)P^{-} \tag{2.8}$$

Figure 2.4 illustrates the different steps of both phases *Prediction* and *Correction*.

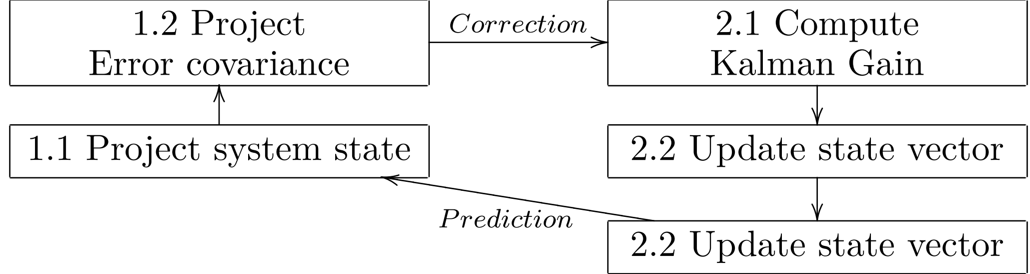

Figure 2.4. Kalman filtering process

2.1.3 k-Means Clustering

K-Means is a nonprobabilistic algorithm which partitions a multidimensional space. The algorithm performs a classification of the data measured by comparing the observations (D-dimensinal Euclidean variables), with the K centers of the space [1] . A cluster or partition is a set of data points (X_n) which are associated by computing the distance among them, which is smaller that the distance to other cluster centers.

Every cluster center is represented by a vector with n dimensions (μ_k). The classification is performed by calculating the distance of each data point (observation) from a cluster center, and selecting the closest cluster center. Therefore, once the algorithm has computed all the distances (d) to all cluster centers, it associates the data point with the closest cluster center vector (μ_k). Distance is presented in 2.9 equation.

$$d(X_n, \mu_k) = \sqrt{\sum (X_n - \mu_k)^2} \tag{2.9}$$

Equation 2.10 shows how the binary indicator (r_{nk}) of the n data point is set to 1 if this center is the closest cluster center [1].

$$r_{nk} = \begin{cases} 1 & \text{if } k = argmin\{d(X_n, \mu_k)\} \\ 0 & \text{otherwise} \end{cases} \tag{2.10}$$

The algorithm is divided into two steps [15]:

1. Assignment: in this step the algorithm computes all the distances and assigns data points (X_n) to the closest cluster centers (μ_k). This process is described by equations 2.9 and 2.10.

2. Update: every cluster center is adjusted by computing a new mean. The mean parameters are computed based upon all data points that were allocated to a specific cluster. This is show in equation 2.11

$$m^k = \frac{\sum_n r_{nk} x_n}{\sum_n r_{nk}} \tag{2.11}$$

These two steps repeated until the algorithm converges, i.e., when no more data points could be assigned. Figure 2.5 shows a k-Mean process sample within a two dimension Euclidian data points. In the first system state (a) the data points are located into the space and two random center clusters are defined (black, and red). In b, it is possible to observe that all the data points are assigned to closest data cluster. The next step (c) shows how the cluster centers are updated by computing all the means (of the data points) that belong to each cluster center. Finally, in (d) is shown how the algorithm has arrived to its convergence, since no more assignments could be performed.

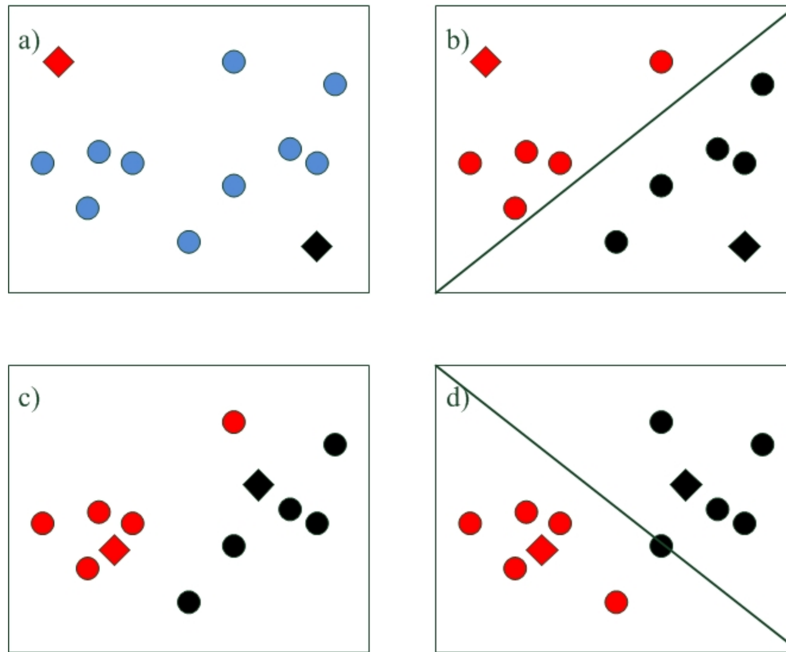

Figure 2.5. k-Means process sample

2.2 Related work

This section presents several projects that are related to my work. These projects present different solutions based on the methods explained in detail in section 2.1. I will briefly explain the goal of these projects and their results.

2.2.1 HMM based

Many projects have utilized a Hidden Markov Model (HMM). This method has become popular, especially in speech and hand-written character recognition. Wiimote Gesture Recognition [16] is a project which presents a solution based on a system that performs data acquisition, filtering, and exploitation through HMM. The goal of this project is motion recognition, while providing reasonable performance. Wiimote is used as an interaction device to capture acceleration of the user's hand. They achieve an 85% accuracy with 3 or 4 HMM states.

Another interesting project is an accelerometer-based gesture control for a design environment [10]. This project consists of two related studies. In the first study Mantyjarvi, et al. investigate the gestures that users make for controlling a design environment. The second study concerns the usefullness of gestures as an interaction modality as compared to other interaction modalities. The project studied which types of gestures are natural and useful for performing any task (i.e., controlling a garage door). These authors present a motion recognition method based on HMM which incorporates a training system (also based on HMM). Their results depend on the number of training vectors, ranging from 1 vector (with 81.2% accuracy) to 12 vectors (with 98.9% accuracy).

Another motion recognition project is called "Analysis of 3D Hand Trajectory Gestures Using Stroke-Based Composite Hidden Markov Models" [11]. This project presents a glove-based solution to recognize a hand's 3D gesture's trajectory. It also incorporates a Polhemus magnetic position tracker, which generates a sequence of sampled 3D positions. They introduce a new concept of gesture, where a gesture does not represent a HMM state. Instead, they define a gesture as a set of different strokes. This project compares traditional HMM gesture (where every HMM state represents a gesture) and strokes as a basis for input HMM algorithm. They achieve 96.88% of accuracy.

2.2.2 Kalman based

Shiratori and Hodgins [20] uses a Wiimote as the main device to control a physically simulated character. They present three different interfaces - each using accelerometer data from Wiimote. These interfaces require users to imitate some

motions such as walking, running, and jumping. The goal of the project is to investigate the use of a Wiimote to control characters and to explore the latency and its effect on the degradation of the user's control in character control. They use a Kalman filter to reduce noise and to extract some features such as a motion frequency, phase difference between Wiimotes, amplitude, and direction inclination.

Yang Wai-Chong [23] uses a Wiimote to capture 3D motion for use with a low-cost approach for interacting with a Head-Mounted Display. The project resulted in a system where the user could perform manipulations of a virtual object. Moreover, the users are allowed to perform tasks, such as navigating through a virtual environment. They system utilizes the IR lights mounted on a Nintendo Wii Zapper gun in order to measure the device's position. In order to estimate the global position and the global orientation, a Polhemus Patriot 6-DOF magnetic tracker was used, which has the gun reported positions as inputs. This magnetic tracker provides position and orientation as outputs. A Kalman filter is used in order to improve and smooth the readings from an overhead optical sensor. The project concludes that the system is suitable if slight inacuracies will not affect the user's task performance in a virtual environment.

Finally, Torres et al. [22] have investigated the use of an Inertial Measurement Unit (IMU) in order to track movement. The IMU is a device composed (in this case) of a 3-axis sensors indicating acceleration, angular velocity, and magnetic field. They describe a set of software algorithms to interpret the data from IMU measurements. Although, this project is not closely related to motion recognition, it uses Kalman filtering to combine the different measurements outputs from IMU to predict the orientation of the device. The Kalman filter enables the use of gyroscopes with short-term precision,with accelerometers and magnetometers which have long-term stability. The output from each of these devices are computed together by a Kalman filter, which estimates an orientation matrix. Finally, the orientation matrix will be used to predict the device's position.

2.3 System Architecture

This section presents the different parts of the system. Figure 2.6 shows how the host application and Human Interface Device (HID) device (Wiimote) are connected. The details of these components will be given in subsequent sections.

2.3.1 Nintendo Wii remote device (Wiimote)

The Wiimote is the main input device we used during the workshops. It has several different peripherals which provide feedback to the users. It is a wireless device based on Bluetooth (using a Broadcom BCM2042 [3] chip). In order to communicate with

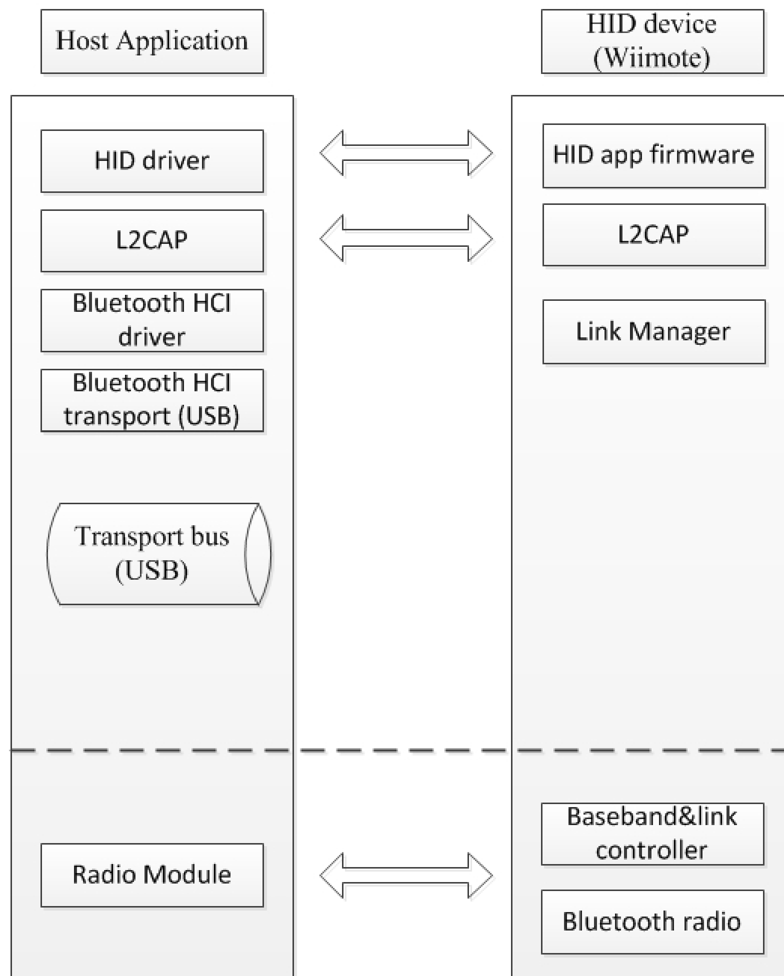

Figure 2.6. Human Interface Device stack

the hosts, the host interface connects to the host's Universal Serial Bus (USB) and communicates using the HID [18] protocol.

2.3.2 Human Interface Device

The main purpose of the HID protocol is to provide a communication channel between devices and applications which require low-latency input-output operations. It provides control of the device's initialization and allows self-describing devices [18]. HID defines how data should be transmitted, while avoiding manufacturer specific protocols. Use of HID is crucial to achieve interoperability, security, and performance.

HID operates over a lower communication layer,in this case USB. The Logical

Figure 2.7. Nintendo Wii remote device (Wiimote)

Link Control and Adaptation Protocol (L2CAP) is used to carry data across the Bluetooth wireless link. HID acts as an interface between these lower layers, and the host application layer. HID basically defines the host and device interface requirements required to implement the desired communication.

2.3.3 Software Architecture

The software architecture is a traditional multi-tier design, based on the client-server paradigm. It is composed of three main layers: low level (C API), interface (Java Native Interface (JNI)), and high level (Java). The purpose of this design was to make the most of the underlying C performance, but at the same time provide a robust and flexible high level application interface. The architecture is shown in figure 2.8.

2.3.3.1 C API

The lowest layer used a non-commercial [1] API, called "wiiuse" develop by Michael Laforest [12]. The most important features this API provides are input and output mechanisms to manage the data, control the wiimote and communication status, and the functions to deal with wiimote peripherals (in this API the motor, LEDs, and the Infra-Red (IR) camera could be controlled).

However, some features needed to be added, such as functions to deal with the speaker, because these were not implemented in the original wiiuse. Therefore, I extended this API by adding some libraries related to the speaker. I also extended several data structures, such as the data structure which represents the wiimote's

[1] GNU GPLv3 and GNU LGPLv3 licenses

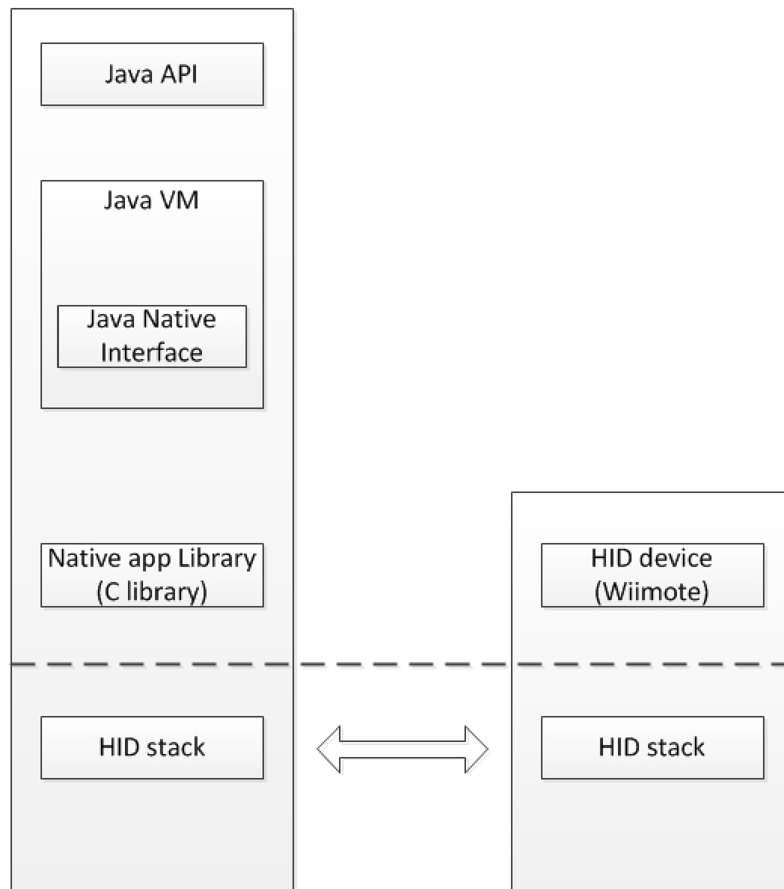

Figure 2.8. Software Architecture

status. The resulting extensions are document in chapter 4 and the source code is included in Appendix A.

2.3.3.2 Java Native Interface

Java Native Interface (JNI) [13] provides Java applications with the ability to call native libraries written in other programming languages (such as C, and C++). JNI allows us to use C libraries (such as the extended wiiuse API described above), for operations such as event polling, and to retrieve the incoming data from the Wii remote.

This layer is based on "wiiuseJ" [5], developed by Guilhem Duche, extended to enable the use of the new versions of "wiiuse". These extensions are documented in chapter 4 and the source code is included in Appendix B.

2.3.3.3 Java API

Initially, we used the "wiiuseJ" Java API, a non-commercial API which is available from Google code [5]. This API was also developed by Guilhem Duche. It provides several interesting features, including a monitoring system, a complete event structure, and event listener protocol.

2.4 WiiuseJ

This section describes both the software architecture and the interaction model behind the WiiuseJ API [5], a Java interface to interact with Wiimote. The interface implements a layered architecture based on the Model-View-Controller (MVC) pattern [19] and event handling paradigm, formed by a set of listeners which react to events generated by the Wiimote device. Details of the Model View Controller pattern are described in section 2.4.2.

The API consists of roughly 60 classes. These classes can be classified into three sets according to the MVC pattern: listeners and managers (controller), events (model), and displayers (view). Interacting with Wiimote devices is done through the implementation of interfaces which provide full control over the events coming from the Wiimote. Therefore, once the listener implements this interface the programmer can create a reactive software application. This listener is especially useful since it provides a large number of different types of events. Moreover the architecture enables an easy way to handle these events. For instance, it is really easy to gather motion information, just by coding a couple of classes, one of which implements *WiimoteListener* interface.

In addition, a Graphical User Interface (GUI) test was provided. This GUI test offering a clear design and whit lot of options for testing purposes. This Graphical User Interface (GUI) test can provide information about the real-time acceleration, orientation or g-force values, and shapes (movement paths). The Graphical User Interface (GUI) can also be used to send commands to the Wiimote device, such as cause it to rumble or turn on/off the LEDs.

Before presenting my analysis of the MVC pattern, I will present a model schema, which offers a means to understand the basic system structure. Following this, the architecture is divided into components, each of which will be analyzed by means of Unified Modeling Language (UML) class diagrams [7]. Finally, the last sections will show some of the relevant processes that are invoked when an event occurs. This process will be presented using UML sequence diagrams.

2.4.1 Big Picture - Model Diagram

This section presents a brief introduction to the system structure through a model diagram, which shows the layered architecture and its components.

I have divided the architecture into three different layers, which I believe to offer an accurate view of the API. As said previously the main architecture pattern is an Model-View-Controller (MVC). However, since any application has to react to incoming events, we must introduce an event handler model. The result is a MVC model adapted to the required roles (events and listeners) in an event handler system.

Figure 2.9 represents the model, showing only the most relevant classes (in order to give a clear snapshot of the overall architecture).

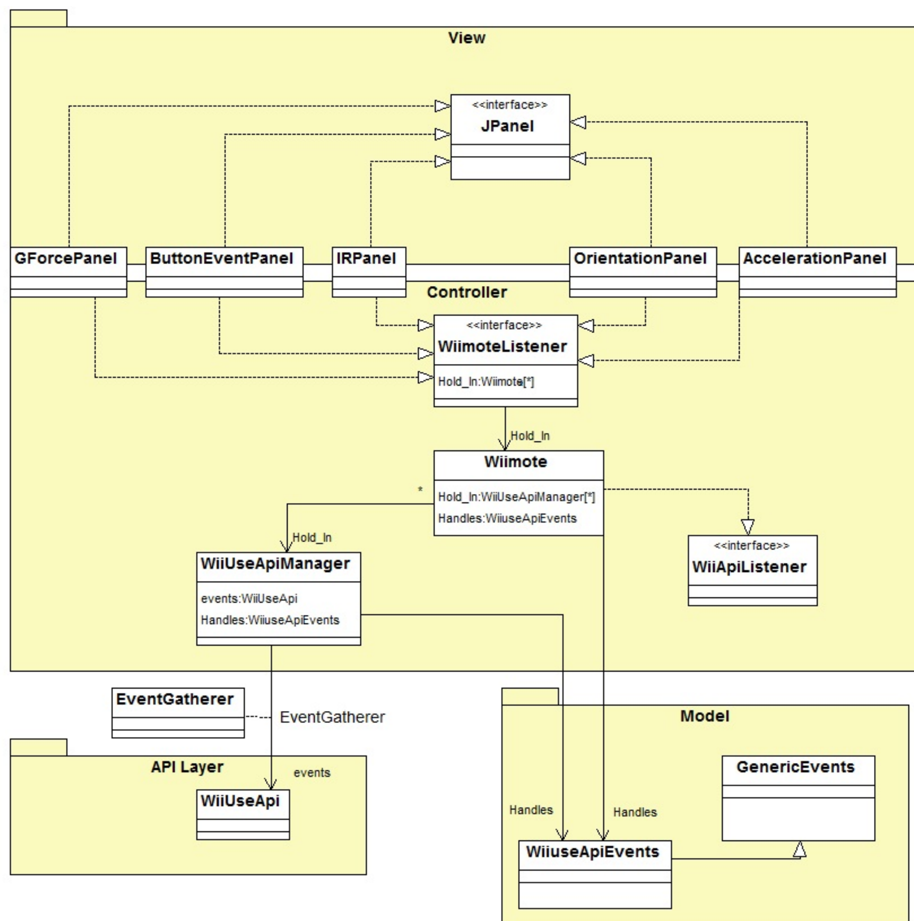

Figure 2.9. System architecture shown as a Model View Controller Diagram

2.4.2 Model-View-Controller Architecture

2.4.2.1 Model

The well-known role of the model [14] is the representation of the stored data and its related components. Therefore, the model represents the state of the system and it provides an interface to interact with this state information, which typically allowing reading, writing, and modifying of the state data.

The proposed model is based on events, which form the data to subsequently be modified and handled by the controller. The events are structured according to their purpose, and handled by specific listeners depending on the listener layer, a concept that will be explained in detail in 2.4.3 section. The following class diagram reflects the implemented event structure:

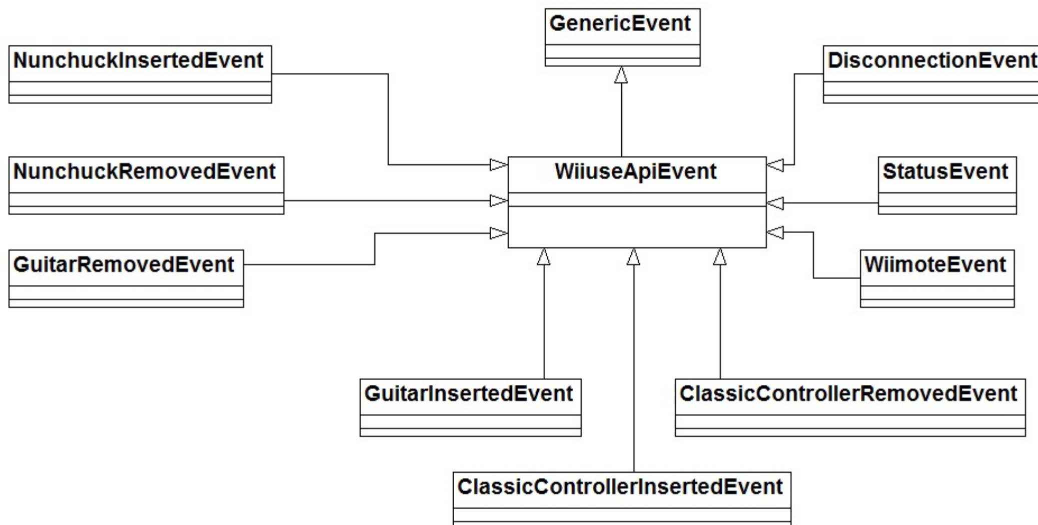

Figure 2.10. Events Class Diagram

On top of the class hierarchy we find *GenericEvent*, which it is an abstract class that defines *setters* and *getters* related to the *Wiimote* class. The second level consists of the *WiiuseApiEvent*, which is another abstract class that introduces the concept of event type through an object variable called *eventType*.

The rest of the event classes contain information for the respective listeners. I will focus only on *WiimoteEvent*, as it is the most important event in my work, since it is the super class of events related to motion capture. This event has the class structure shown in figure 2.11.

The *WiimoteEvent* serves as container for the events that are closely related

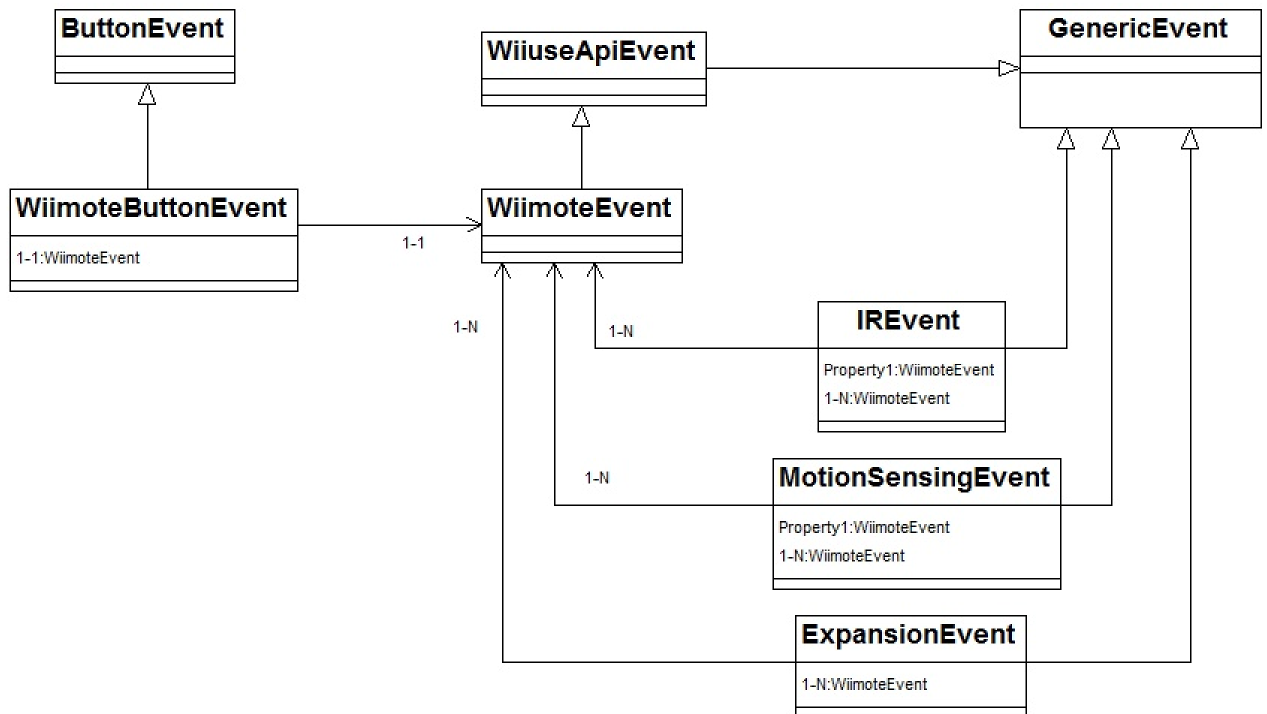

Figure 2.11. Wiimote Event Class Diagram

with the user interaction: Buttons, IR Camera, Motion, and Expansion. Details of these are given below.

- Buttons handle event produced by the Wiimote. The implemented buttons are: A, B, Up, Down, Left, Right, Minus, Plus, and Home. Moreover, the event also differentiates between three pressed actions: Pressed. Held. and Just Pressed.
- IR Camera provides information related with the coordinates (X and Y) and also with the screen.
- Motion provides information about these important features related with motion features: (three axis) Acceleration, G- force, and Orientation.
- Expansion can be used to implement of new events.

2.4.2.2 View

The main goal of the View component [14] is to present the data (model) to the user. The view component provides the interface between the model and the user, allowing the user to interact with the model. Depending on the purpose of the

application and the user's goals, the view should provide different views of the Model. For instance, for human interaction GUIs are widely used.

The View component in WiiuseJ API is based on the class structure shown in figure 2.12.

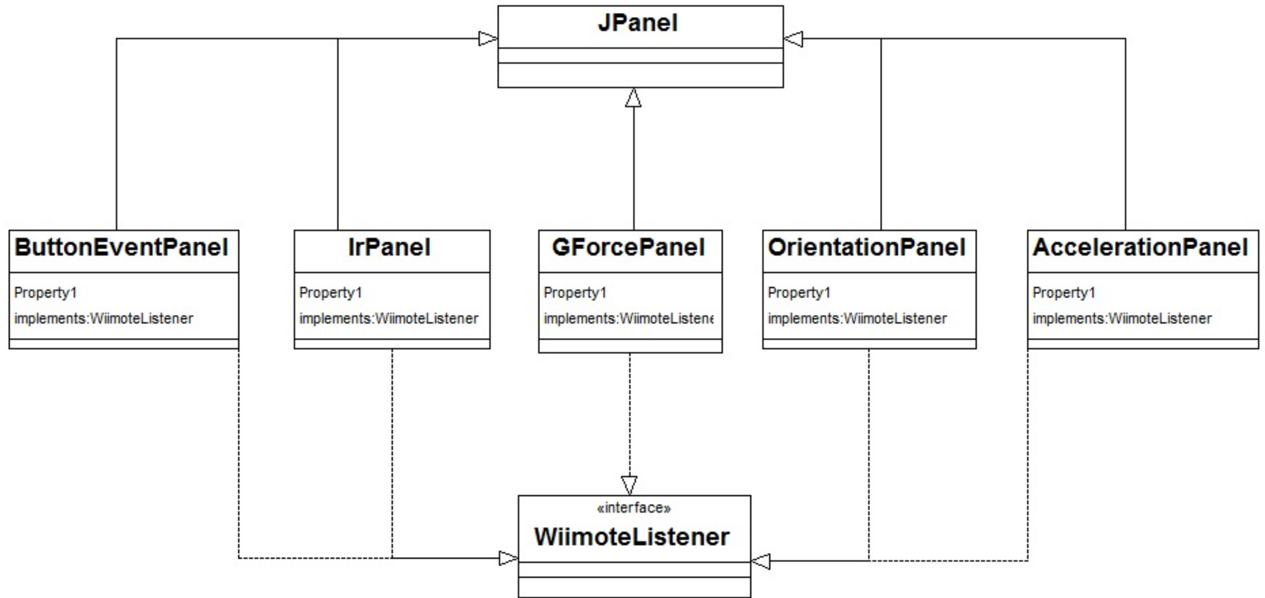

Figure 2.12. GUI objects Class Diagram

These are the basic classes provided to implement a GUI. Each class represents a screen whose main purpose is to test different aspects of the Wiimote. In this case the name of each screen clearly indicates the purpose of the screen. WiiuseJ also includes a complete GUI developed with these classes and the addition of some other classes. This complete GUI offers the interface shown in figure 2.13. While figure 2.14 shows how the classes are structured and the relationship among them.

The main class in the View component is *WiiuseJGuiTest*. It is in turn responsible for each for each of the following tasks:

- Act as a container (*JFrame*) of the different screens (*JPanels*). It performs the initialization and update of the contained *JPanel* classes.
- Bind GUI with Wiimote events, by containing a reference to *Wiimote* object and implementing *WiimoteListener* interface. Afterwards, the GUI must be registered in the *Wiimote's* event listeners list to enable the event-handling connection.

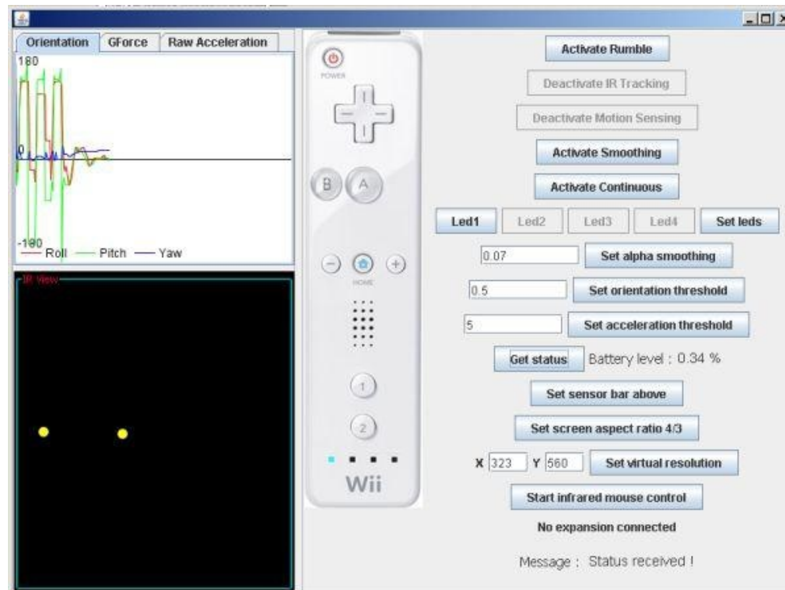

Figure 2.13. Graphical User Interface Testing

- Act as a listener for the incoming interface screen events (from the human testers), these events are used either send commands to the *Wiimote* object or processed locally (screen changes).

Classes from the basic schema (*IRPanel*, *ButtonEventPanel*, *GForcePanel*, *OrientationPanel*, and *AccelerationPanel*) implement the *WiimoteListener* interface and extend *JPanel*. So, these classes are components of the GUI and at the same time act as a listeners. Hence, they handle incoming events and perform actions. Therefore, in order to receive events, the reference of the class needs to be saved in the "transmitter" class (typically on a listener list) and the "receiver" class needs to extend to proper interface. This ensures that the event-handling classification is not broken by any class. In this case they are on the bottom of the listerner's model. I will provide more details about event handling model in section 2.4.3.

2.4.2.3 Controller

This component [14] interacts with both model and view, in order to fulfill the actions from the view which has effect on the model. Therefore, the controller is the responsible for maintaining the model in terms of writes, reads, and updates. It also can manipulate the access permission, to change the allowed actions for a specific user.

The API has the class structure for the controller shown in figure 2.16. At the of the hirarchy we find the *WiiApiManager* class, which deals with the lowest

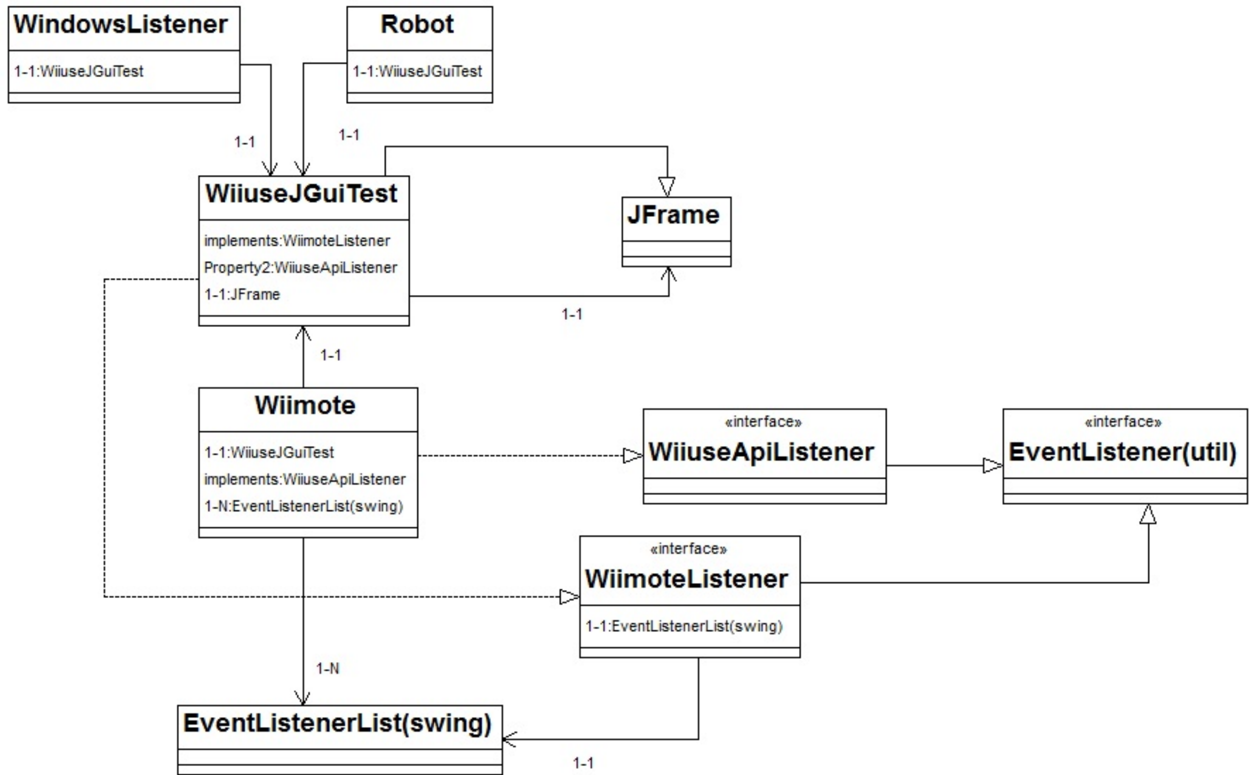

Figure 2.14. GUI objects and Listerners Class Diagram

API level, contained in the *WiiUseApi* class. It also extends from the *Thread* Java class, such that the *run()* method has a loop where catching all the events coming from the *WiiUseApi* class. By implementing a thread in the listener we connect the controller and view components. Partitioning the functionality into different components means that an error in one component does not cause problems in the other component. Moreover, this is a good method to create a reactive GUI, which allows the user to interact with it when the component is properly running. Moreover, a reactive GUI allows manual reinitialization of the controller if an error occurs. In this case, a disconnection is forced and the user needs to press the reconnect button. By pressing this button a reconection of the *Wiimote* is performed by *WiiUseApiManager* and the communication recovered.

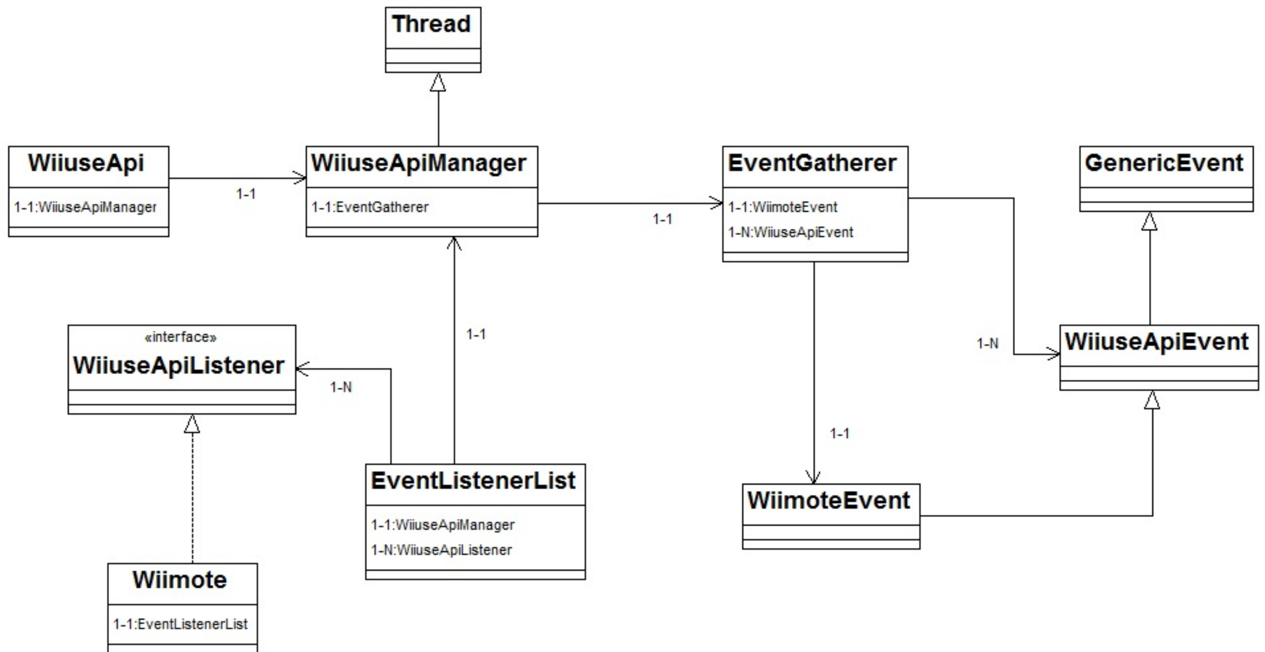

Figure 2.15. WiimoteApiManager Class Diagram

The most important tasks of *WiiApiManager* are presented here:

1. Deal with the low level API translating the high level requests into primitive calls. For instance, the method related to the Wiimote's connection, *connectWiimotes()*, can be decomposed into the following *WiiUseApi* primitive method calls:
   - *init()*,
   - *find()*, and
   - *connect()*.
2. Capture the events, through the *EventHandler* class, and transmit them to the attached listeners (these listeners implement the *WiiApiListener* interface).
3. Manage and control the connected devices (*Wiimotes*). This interactions utilize methods such as: *activateRumble()*, *setLeds()*, or *getStatus()*.

The sequence diagram shown in figure 2.16 illustrates the connection process which is performed when the main application wants to discover and register the available Wiimotes. In this process the *WiiApiManager* has a relevant role, since it is responsible for interacting with the primitive methods (from the *WiiuseApi*

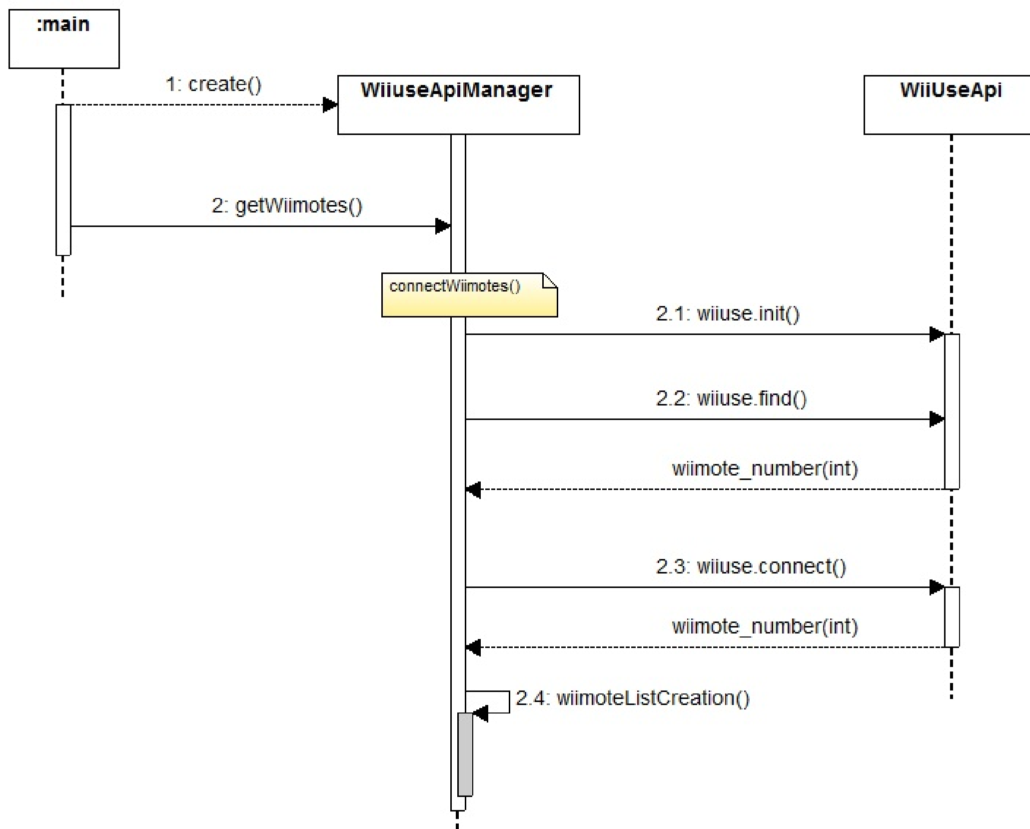

Figure 2.16. Wiimote Connection Sequence Diagram

class), and to create the Wiimote data structure.

When the *WiiuseApiManager* creates a list of *Wiimote* objects, those must be created by *WiiuseApiManager*, instead from the *WiiuseApi*. Therefore, the *WiiuseApi* class just returns the number of connected devices, not a set of objects representing each of them. Once the *Wiimote* objects are created they are associated with and unique identifier.

2.4.3 Event Handler System

This section presents the event system through dynamic schemas, as sequence diagrams. In section 2.4.2 the static architecture is presented by means of class diagrams, which clearly reflects the application structure. However, they do not provide a good view of how the processes are performed by the different components of the model. This process view will be described in the following subsections.

2.4.3.1 The Listener Model

As presented above, events are static data structures (represented by classes) which form the model. There are different types of events, depending on the purpose of the event. At the top of the the event hierarchy we find the *GenericEvent* class, which it is extended by the other event classes.

The result is a dynamic system that is responsible for performing event management. This structure is based on listeners, which implement two basic interfaces: *WiiuseApiListener* and *WiimoteListener*. Both of these interfaces are extensions of the Java *EventListener* API.

The listener hirarchy is shown in the figure 2.17. The basic performance of the controller is based on a listener hirarchy which utilizes a set of low level listeners, typically organized as a list of listeners (*EventListernerList*). Once an event occurs this listener notifies all of the other listeners which are stored on the list the event. The notification is performed by calling a specific method defined in each listener interface. For instance, *WiiUseApiManager* utilizes *notifyWiiUseApiListeners()* method to perform this notifications. *Wiimote* performs these notifications through a set of methods, which the name of those depends on the listener's name (*notify+ListenerName()*).

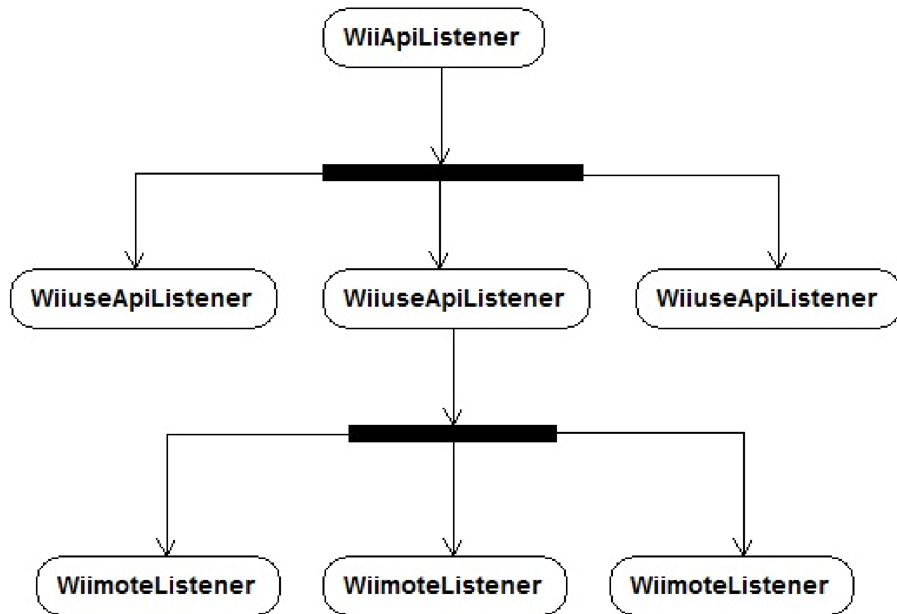

Figure 2.17. Listener Hierarchy Diagram

2.4.3.2 Event Handling Processes

In order to illustrate the event handling processes we will consider the following examples. The first of these will be the WiiApiManager event gathering and passing to WiiuseApiListener listeners. This process is shown in the figure 2.18.

The process is performed in a loop located in the *run()* method of the thread. *WiiApiManager* uses an object called *EventGahtherer* which is a container of the incoming events. After retreive the gatherer *WiiApiManager* notifies that events are to the available *WiiuseApiListeners* by calling the *onUseApievent()* method. The process ends when the incoming event type is *DISCONNECTION_EVENT*, which closes the connection (closeConnection()).

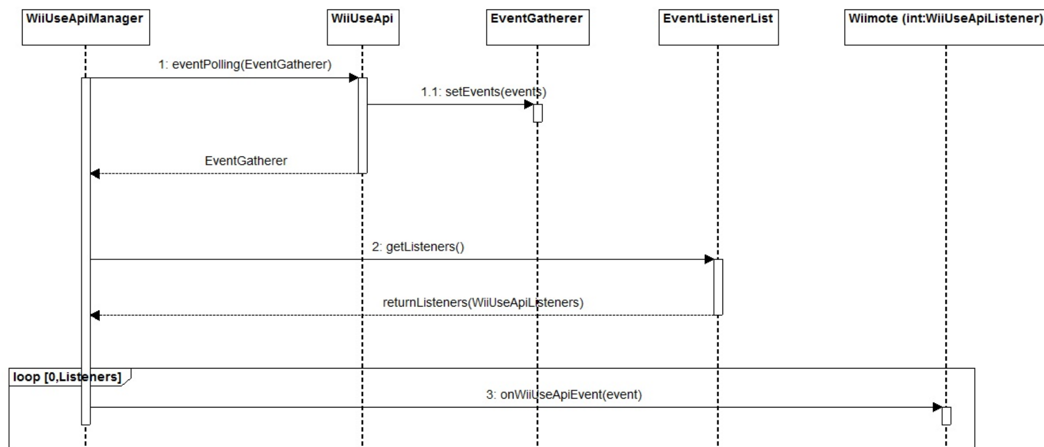

Figure 2.18. Event Handling Sequence Diagram

The above process is followed by the WiiuseApiListener passing generic events to WiimoteListeners, which is shown in the figure 2.19.

This is the continuation of the first process. Here, the *WiiApiManager* has gathered all the incoming events and notifies the *WiiuseApiListeners* of these events. This notification triggers another notification, in this case from the *WiiuseApiListeners* to *WiimoteListeners*, which are stored in *Wiimote* object.

The diagram shown in figure 2.18 shows only a single event case, *GENERIC_EVENT* type, which is the most interesting event in terms of user interaction and motion capture. This event, as mentioned before, may contain: buttons, IR, motion, and expansion events.

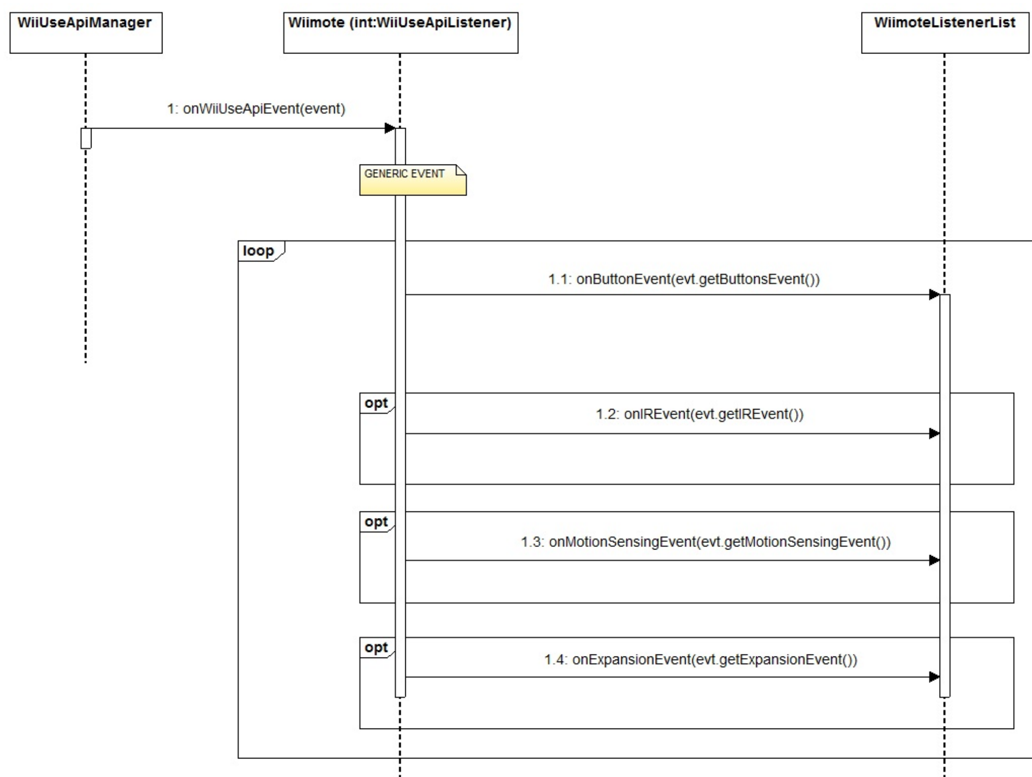

Figure 2.19. Event Distribution Sequence Diagram

Chapter 3

Method

3.1 Project Context

As mention in Introduction, this masters thesis is framed in a research project [21] within Mobile Life Center in collaboration with the Wii Science project [8]. The aim of this collaboration was the study of bodily interation and kinesthetic learning with a group of children.

Involvement of the body to interact with technology is considered as an alternative to traditional mouse and keyboard interaction. Bodily interaction provides the users new forms of experiences that could not achieved with traditional interactions (i.e., hand-eye interactions). Therefore, the project provided a context to children which supports the interaction in terms of consumption, preservation, and creation of energy. As metioned before, a Wii remote (Wiimote) was used as device to provide sensing of a user's motion and to provide feedback.

3.1.1 Design Process - Workshops

The design process was divided into several workshops. These workshops provided sufficient information and experience to define a final activity called "Weather Gods and Fruit Kids Game", which allowed us to achieve the goals of the research project described above. I will briefly explain the different workshops, which are crucial to a better understanding of the game that was designed and that drove the evolution of my own part of this project.

First workshop

This workshops served as a preliminary study. Two members of the team collaborated with teachers to design a game to demonstrate how kinesthetic learning could be applied

to traditional subjects (i.e., physics). The design method consisted in a brainstorming session. The conclusion was that the concept of energy (consumption, charging, and preservation) was a suitable topic to focus on.

Second workshop

The aim of this second activity was to design a simulated environment in order to teach kids about energy consumption. The children wore simulated sensors (on their hands and legs) and need to perform a set of activities: throwing a ball, moving to mimic a set of pictures (kite, snake, and ball), and moving according to specific motions (slow motion, robotic motion, and bid and round). The children were able to see the energy consumption for each activity through a computer's screen.

Third workshop

In this third workshops sensors were used (i.e., a Wiimote) in order to measure the actual energy consumption. The goal of this workshop was make children think about the results of each activity in terms of energy consumption. Three activities were designed for this purpose: (i) runing and walking, (ii) operate a lamp and a fan, and (iii) move to charge a battery. The Wiimote provided both the audio and vibration feedback.

Through these workshops were that children explored the consumption and storage of energy in a controlled and structured manner. Moreover, we observed that the presence of the computer's screen, for the visualization of the results, restricted the interactions between the children. Based upon the second and third workshop we decided to avoid visualitzation as the user's feedback for The Game.

3.1.2 The Game

"Weather Gods and Fruit Kids Game" consists on two teams (fruit kids and weather gods) of two players each, which have to compete each other. The activity scenario was a gym with several obstacles. All the participants had Wiimotes attached to their arms and legs in order to capture body motion. Figure 3.1 illustrates how the devices were attached to the players body.

Fruit kids have the objective of collecting pieces of fruit, without touching the ground, and bring this pieces of fruit to their nest. Once at the nest, they refill their energy storage to a full level. Weather gods are placed on a stage, where they have

Figure 3.1. Attached wiimotes

good visibility of the game space. Their goal is to obstruct the fruit kids by stealing all their energy. In order to do this, the weahter gods charge their Wiimote and, once charged, cast spells by performing the specific motions (i.e., “Slow motion”, “Big and Fast”, and “Robotic” motions presented in chapter 1). The spell casting was accompanied with thunder (speakers) and lighting (spotlight).

In this activity we used vibration and audio as the main feedback to the user in order to support users understanding their status in the game. However, this feedback differed depending on the team that the user was a member of. This

difference in feedback was as follows:

Fruit kids

The feedback was used to inform them of their current energy level. The vibration consisted of pulses with different frequency proportional to their energy level. With a high frequency corresponding to a high energy level.

Weather gods

In this case the vibration informs the players whether the performed motion was the expected one. Audio feedback (via the Wiimote speaker) was used to inform the user of the performed motion (as described in chapter 1).

3.1.3 Conclusions

After analyzing the recorded videos and interviews we found that weather gods performed motions with both arms and legs. So, their whole bodies were involved in this experience, rather than just the specific location where the devices were attached.

Another important aspect of the experience was the role of the device. For instance, the device and the context allowed the users to perform in an unconstrained way all motions, so, neither the device nor context dictated their movements. This increased the child's ability to interact in the game and lead to the strange dances that children came up with. In addition, we observed that the children performed motions more freely than during the first two workshops, the children's movements were more controlled.

The important role that this freedom plays is reflected in each child's behaviour. For instance, some weather gods performed weird and tricky movements. As a consequence, the users could produce their own experience by engaging theirselves in this embodied experience. Another example of this freedom was that some weather gods choose different ways to cast an spell (i.e., arm fully extended out forward from the shoulder), regardless of the fact that the action simply required pressing a button.

3.2 The architecture from a design point of view

This section describes in detail the architecture [4] that has been proposed in order to fulfill the requirements that arose from the design process presented in section 3.1.

3.2.1 Overview - Model Diagram

Similar to WiiuseJ, presented in section 2.4.3.2, our platform is based on the Model-View-Controller (MVC) pattern. However, many changes and extensions were introduced in order to fulfill the design requirements introduced in chapter 1 and section 3.1. One of the most important changes is the introduction of multi-threaded behaviour, in order to improve the performance and to enable multiple device interaction. Figure 3.2 shows the UML class diagram, which represents the main system classes of the proposed system architecture. In following sections more detailed information and figures were provided to explain the proposed architecture.

3.2.2 Multithreaded Application

One of the most important requirements was the ability of the platform to deal with multiple devices in real-time. However, WiiuseJ did not implement a multi-threaded structure, hence, it could not deal with synchronized devices nor control the user feedback devices in real-time. Unfortunately, these were important requirements, since our workshops were designed to study body motion within a group of users that interact with each other and to provide different device responses (i.e., rumble, sound, and lights).

The solution was to develop a multi-threaded Java API, where all the atomic actions (i.e., sending a sound to the speaker, turning on rumble) were controlled by different threads. This is a distributed system design, where a thread (*MessageDelivery*) acts as a C API event listener, delivering all the incoming events to different listeners (Wiimote listeners). Every device is represented by a Wiimote object, which has a listener, making the devices completely independent of each other. By representing every device as an indenpendent process within the same system, we could implement interaction among them, while at the same time, providing a sense of concurrent feedback to the user.

The framework we used to achieve our goals was the Executor interface ([2], chapter 6). It provides a platform where the tasks or units of work, can run asynchronously. The main benefit of this framework is improved thread resource management, as well as increased responsiveness and throughput. Hence, it is possible to decouple the task submission from task execution. This is crucial for our purposes, since we need to deliver the sensor with low delay. For instance, the

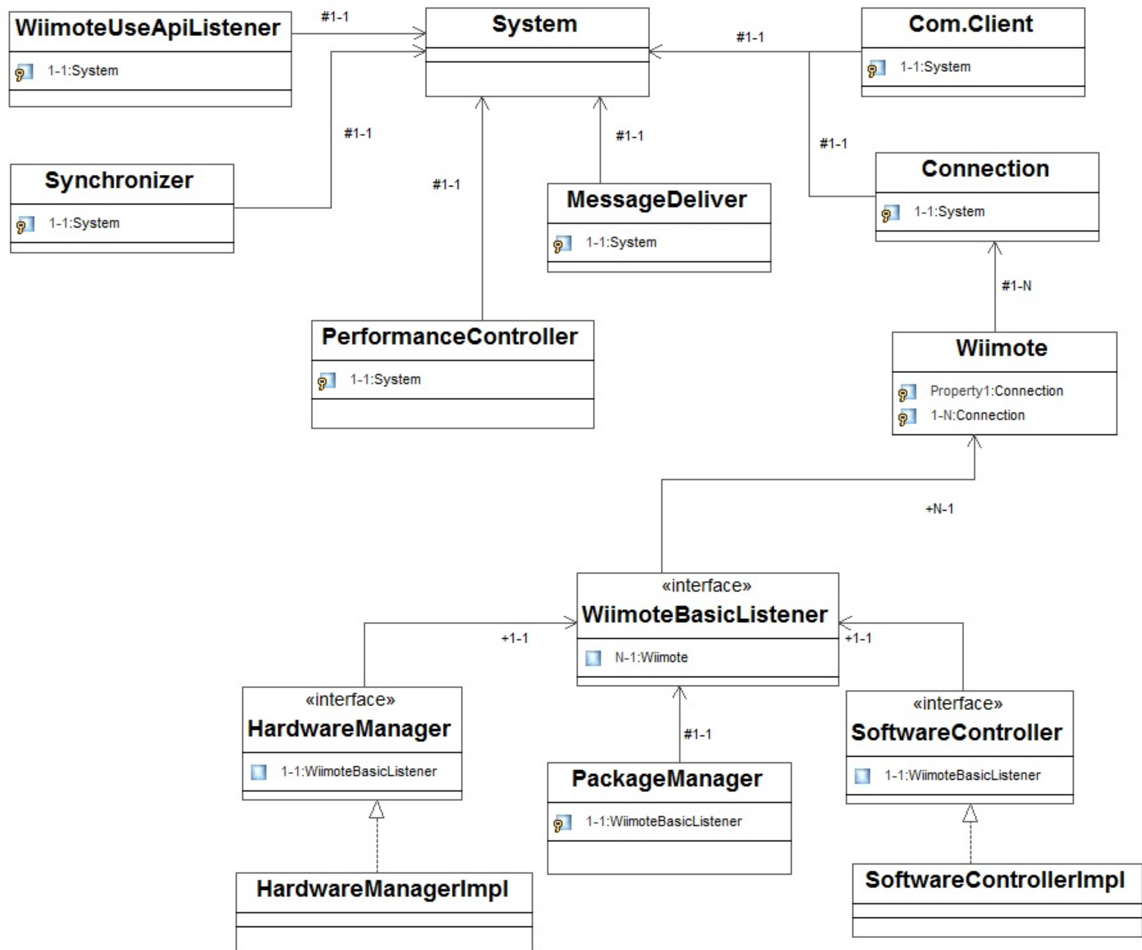

Figure 3.2. Architecture - overview

atomic actions (i.e. turn on a LED) are asynchronously executed by tasks ([2], chapter 5) (Runnable or Future), which instantly release the listeners. Therefore, the listeners could manage more incoming events from the Wiimote, increasing the application's throughput.

Finally, we defined the communication channel among the architecture components. Since our problem perfectly fit a producer-consumer pattern, we used a queue system based on blocking and non-blocking queues ([2], chapter 5), which simplifies the thread communication. For instance, the *GameController* component turns on the rumble by sending a command to *RumbleController*. This is achieved through a blocking queue.

3.2.3 Model-View-Controller Architecture

3.2.3.1 Model

The model hierarchy presented in figure 2.10, represeting the event system, was not modified, but rather some of the classes were adapted by the addition of some fields to some the rellevant classes.

Although most of the classes of the original API (WiiuseJ) were maintained, some changes needed to be introduced in order to extend the functionalities of the system. As introduced in chapter 2 section 2.4, one of the most relevant class is *GenericEvent*, which is at the top of event class hierarchy. This class was modified by the addition of some fields such as *refNumber*, which is used to keep track of the packets. This field enables the synchronization of the packets and makes possible system testing tasks. It is number which is assigned when the packets arrives to the system.

Another important class is *WiimoteEvent*, which was modified by the addition of a time stamp field. This field is also used for both testing and to enable the construction of data *chunks*, which are composed of acceleration packets sent by the Wiimote device. Controller component classes such as *Postman* and *PacketManager* are responsible for performing these tasks. This process will be explained in detail in subsection 3.2.3.3.

Some classes were added in order to support new functionalities. For instance, the *AccelerationPacket* class, represents every element in a data *chunks* structure. This class is the basic unit of work of the system, as these data chunks will be used for motion recognition and our subsequent testing. This class stores some important information such as accelerometer statitics for each of the three axes (x, y, and z), the mean and the variance of the accelerations samples, and a time stamp.

Another important addition is the class *Centroid*, which supports the k-Means algorithm. This class is a representation of the algorithm's clusters, thus, it stores the data points which belongs to each cluster - these are used to compute the cluster center at each algorithm iteration.

As explained in section 3.1, the concept of a spell needed to be introduced, since WiiuseJ is a generic API, and we needed some specialized functions to support the workshop context. This lead to the addition of a *Spell* class, which contains basic information such as *energy* and *type*, as required in the final workshop.

Finally, some configuration classes were introduced to easily change the system configuration. This is the case of *SpeakerConfiguration* class, which stores all the supported frequencies in both modes PCM and ADPCM.

3.2.3.2 View

Although WiiuseJ provides a GUI which could be used for data visualization, such as showing the instant acceleration, this GUI was not really useful for our purposes. The main reason is the multi-thread architecture behavior, where a lot of processes are executed in parallel performing multiple tasks at the same time. This parallelism makes data visualization for debugging and testing purposes such as for statistics gathering quite difficult.

Moreover, the conclusions presented in section 3.1 discouraged the development of a GUI for children. Therefore, the decision was to develop a simple visualization component based on Log4J [17], which basically provides a log framework that enables data visualization. This framework is presented in detail in subsection 4.1.

3.2.3.3 Controller

Controller suffered multiple modifications and extensions. The need of dealing with multiple devices, presented in section 3.2.2 force us to design a new Controller component. This new design is based on *executor* framework and complemented with *tasks*, and communication *queues*. In addition, some design aspects of WiiUseJ has been changed in order to increase the code reutilization and the information hidding. All changes and extensions will be explained in this section.

Class Hierarchy One of the most important changes in class hierarchy is the role of *WiiUseApiManager*. This class, in later API versions, had several tasks. These tasks are:

JNI controller
: In order to have access to C functions, JNI methods are called. The closest class connected to JNI is *WiiUseApi*, which acts as a controller. The methods are synchronized to enable secure access to shared functions. *WiiUseApiManager* acts as a intermediate controller, providing information about the system to *WiiUseApi*.

Connection controller
: Before enabling the event handler system, the connection task creates the *Wiimote* objects and stores them in a list.

Message gathering
: The message gathering task is the responsible for gathering the events generated in C API (i.e., based upon the events coming from Wiimotes devices). Moreover, it performs some basic computations before forwarding the events further in the system.

Message delivery
Once the messages were received the were forwarded to the eventhandler system through *WiiUseApiListeners* obejects.

The conclusion was that *WiiUseApiManager* performs many tasks that were not closely related. This supposes a poor design in terms of modularity, since the problem it is reduced to a *big* class instead divided into several sub-problems. Additionally, it complicates the inheritance, by having to include with a lot of unrelated methods and system state. Finally, the development process becomes tedious, since the amount of information obcures how the methods change the state of the system.

As a result the *WiiUseApiManager* was decomposed into four explicit tasks, with each task represented by a different class. The proposed design provides specialized classes and interfaces, which increases the reuse of code, improves information hidding, and makes it easier to understand the whole system. These four tasks are:

JNI controller
This task is performed by *WiiUseApiManager*. As explained above, it is a controller which filters the access to JNI methods.

Connection controller
The *ApiConnection* interface and *ConnectionManager* were created to perform this task. The *ConnectionManager* is the responsible to for initializing the connection with devices, by creating the *Wiimote* objects, sending them connection feedback (vibration), and creating the appropriate listeners for every connected device.

Message gathering
The *MessageDelivery* class is responsible for dealing with incoming events. These events are handled by a continuous event gathering function which is provided by JNI and the C API.

Message delivery
Once the *MessageDelivery* task has gathered the incoming events it sends them to *DeliveryAssistants*, which are threads (implementing the Runnable interface) that perform some message modification (such as setting a time stamp). Once this task is performed they deliver the event to the *WiiUseApiListener*, which is an interface implemented by *Wiimote* objects.

PackageManager *PackageManager* plays an important role in this architecture. As explained in Introduction, *chunks* of data are formed from the accelerations events received from Wiimote. *PackageManager* is the responsible to form this data *chunks*. Once the listerner receives the acceleration events, these are sent to

this controller, and then, based on the packet time stamp, are grouped in these data *chunks*. The data structure which groups these motion events is formed by a vector that contains the acceleration values (x, y, and z axis), the mean and the variance of these values, and the distance to the closest cluster centers, in case that k-Means algorithm was enabled.

It is important to remark that there is a trade-off between the *chunk* period and the feedback delay. For instance, a hight *chunk* period forms data structure with a lot of information, which provides more information to motion recognition algorithms, but at the same time increases the application delay. This parameter was refined through the design process workshops, and the conclusion was to take 250 milliseconds to have enough information, a range from 0 to 25 acceleration events per *chunk*, but, at the same time provide a real-time feeling to users.

In addition, perform statistic gathering of the acceleration values received and send rellevant information (time stamp) to *PerformanceController*, which keepts track of the architecture delay. In section 3.2.4 an UML sequence diagram is provided in order to illustrate the role of the different controller classes, including *PackageManager* class.

Event Handling Event Handling system, introduced in detail in chapter 2, also had suffered several modifications and extensions. These changes were motivated by the need to implement different system modes, as introduced in section 4.2, that required new classes to deal with the new functions.Hence, instead having generic listeners (*WiimoteListener* interface), especialized listeners are required, since the tasks are not related. Figure 3.3 shows the current listener class hierarchy.

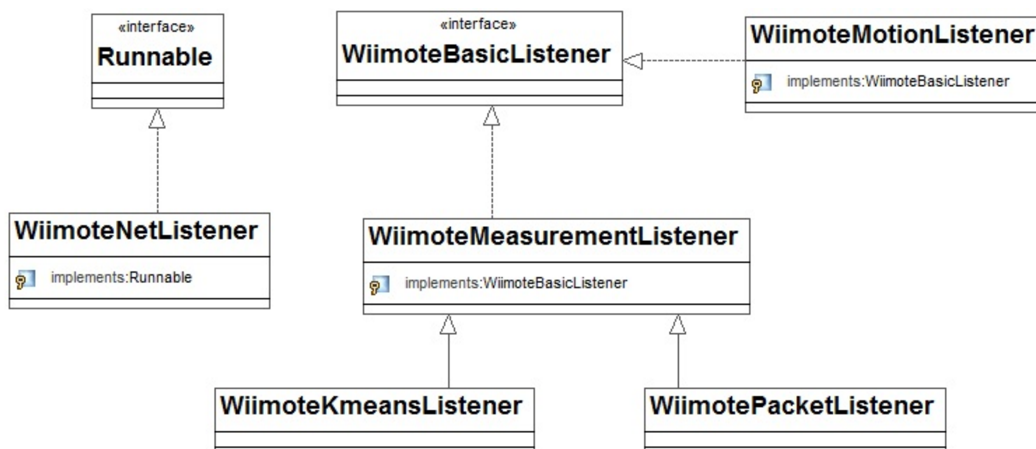

Figure 3.3. Event Handling classes

The following listener classes were introduced for enabling the new functions, most of them are related to testing and gathering system statistics . As introduced above, these listeners are created by *ConnectionManager* depending on the running mode.

WiimoteMotionListener

This is a simple version of *WiimoteListener*, which enables the motion sensing task. This listener is used to collect acceleration statistics without enabling *HardwareManager* and *SoftwareManager* controllers. Hence, it is simply connected to the *PacketManager*, which as explained above is responsible for gathering statistics. Figure 3.2 illustrates this scenario.

WiiuseMeasurementListener

As shown in figure 3.3, *WiiuseMeasurementListener* it is a super-class which provides some common methods and system state to *WiimoteKmeansListener* and *WiimotePacketListener*. It basically defines a set of variables, such as *wiimote* or *measureLength*, that define the system state, shared by these two sub-classes.

WiimoteKmeansListener

This a very specific listener which is used when the k-Means algorithm is enabled. The *WiimoteKmeansListener* has two different phases. The first one is information gathering, where the user needs to perform system training of the designed motions (“Slow”,“Big and Large”, and “Robotic”). In this phase the listener stores the data points obtained and builds the algorithm clusters, one per motion. Once this is done, the motion recognition phase starts, and every acceleration packet is compared (acceleration axis, mean, and packet rate) and classyfied into a cluster.

WiimotePacketListener

As explained in *WiimoteKmeansListener*, the k-Means algorithm performs a comparison between the data points. This comparison, takes several parameters and one of them is the packet rate received in every period of time (data *chunk*). *WiimotePacketListener* is a basic listerner that computes statistics about the number of packets received when a specific motion is performed by the user. This information is used for defining cluster initial values if the training process it is not enabled, in this case random values are set, then we measure system properties and the algorithm’s convergence time.

WiimoteNetListener

> This listener is created when network performance is to be tested. As shown in figure 3.3 *WiimoteNetListener* is implements a Runnable interface instead of *WiimoteBasicListener*. This difference is because the nature of this class is completely different from the other listeners, so, it could not be classified with the others. This listener's main task is sending report packets to Wiimote devices and measuring the network delay. Moreover, it collects statistics that enable use to estimate the network delay. Section 4.3 presents a UML sequence diagram that illustrates this process.

In terms of interfaces, *WiimoteBasicListener* was created. This interface defines the most frequently used methods in the *WiimoteListener* interface and is a generalization of the *EventListener* interface, provided by Java Standard Edition API. As mentioned before, most of the methods defined were not used in the proposed architecture, so, there was no real need for such a large and complex interface; hence our definition of a basic listener. Figure 3.4 shows the Event Handling system interfaces.

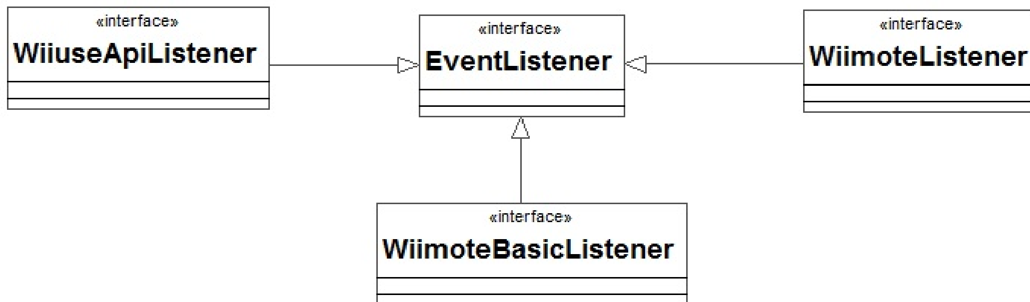

Figure 3.4. Event Handling interfaces

Hardware Controller In order to provide a well structured and multithreaded architecture design, the methods that deal with Wiimote device feedback (LED, rumble, and speaker), were grouped into a Hardware Controller hierarchy. The basic role of these components is to provide a parallel access to *WiiUseApiManager*, which is responsible for enabling access to the JNI API. Hence, for every response type there is a responsible class that controls access. *HardwareManagerImpl* is responsible for creating the other controllers (*LED*, *Rumble*, and *Sound* classes) and providing them the needed parameters to perform their task properly.

Moreover, the *HardwareManager* interface was created to increase information hiding and to increase code reuse. Figure 3.5 illustrates the Hardware Controller class hierarchy.

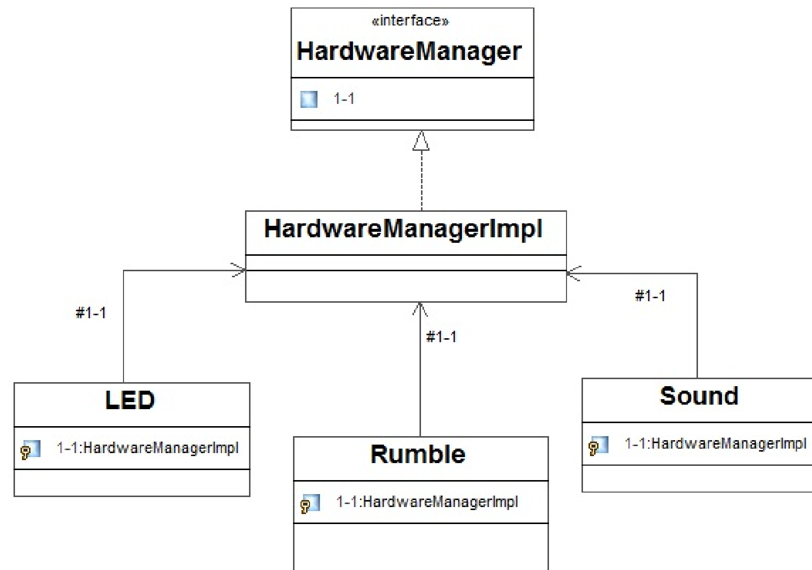

Figure 3.5. Hardware Controller hierarchy

Software Controller Similar to the Hardware Controller hierarchy, some classes were grouped into a layer which is responsible for dealing with the requirements which arose from the "Wheather Gods and Fruit Kids" workshop, as explained in detail in section 3.1. Figure 3.6 illustrates the Software Controller class hierarchy.

SpellGame

The *SpellGame* class controls the different states of the game. It is a state machine where every state represents the expected motion from users. The expected motions change in time, and the users need to identify which motion is expected at any time by two main feedbacks, audio and vibration, as explained in section 3.1. When the device is full charged the state of the system does not change until the users have casted a spell.

RealTimeFeedback

The *RealTimeFeedback* is responsible for execution of the motion recognition algorithm based on computation of mean and peak detection. *RealTimeFeedback* receives data *chunks* from the listener (*WiimoteListener*), that previously

have been computed by *PackageManager*, and performs the motion recognition. The resulting recognized motion is sent to the *EnergyTransmission* class.

EnergyTransmission

The *EnergyTransmission* class manages the energy state of the device. Based on the motions performed by the user at any time, and the motion expected motion for this period of time, this class updates the energy level. If the energy level, representing how much charge the device has, was explained in section 3.1, determines the possibility of a user casting a spell in case of “Weather God”. Once the spell is casted the energy level, of the “Weather Gods”, is set to 0, and no energy is accumulated in the device. The energy levels are based on thresholds that could be configurated to adapt them to different scenarios.

Finally, when a spell is casted, the information that a spell has been cast is sent it to the *CommunicationClient*, which sends the spell cast event to the computer that controls the state of “Fruit Kids”.

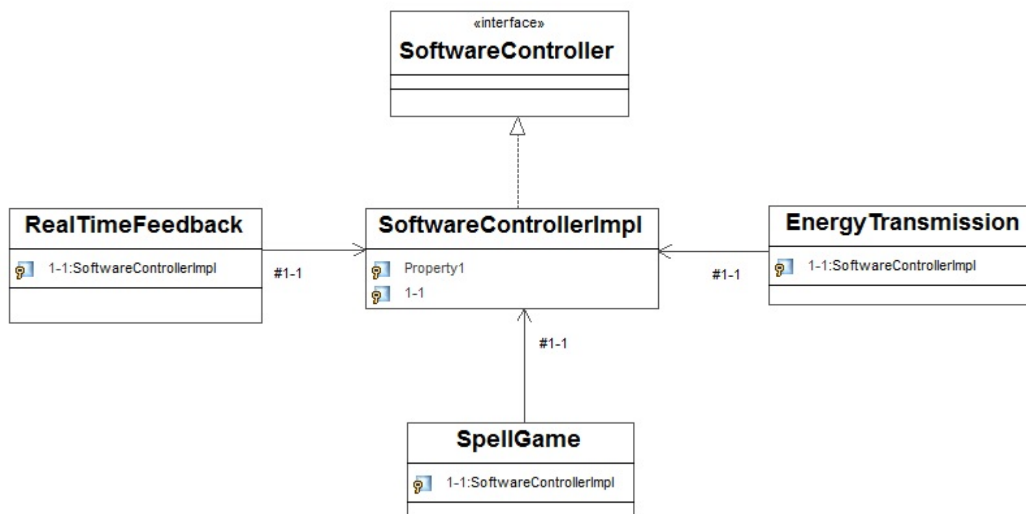

Figure 3.6. Software Controller hierarchy

3.2.4 Whole Process

This section shows some UML sequence diagrams that provides a better understanding about some important processes, which are perfomed within the architecture.

Message Delivery The figure 3.7 illustrates the process of gathering new events and spread them into the architecture, through Event Handling system. One of the most important changes in relation with WiuseJ, explained in chapter 2, is the introduction of *MessageDelivery*, in order to decoupling a set of tasks that were concentrated on *WiiUseApiManager*, as explained above. Moreover, as mentioned before, it is possible to observe that *MessageDelivery* uses the concept of *task* introduce by Executor framework, in order to decouple task creation and task submission. This technique increases the system performance, by creating threads (which implements either *Runnable* or *Callable* interfaces) that perform an specific task. In this case, the *DeliveryAssistant* performs a message modification and deliver the messages to their recipients (*Wiimote* classes).

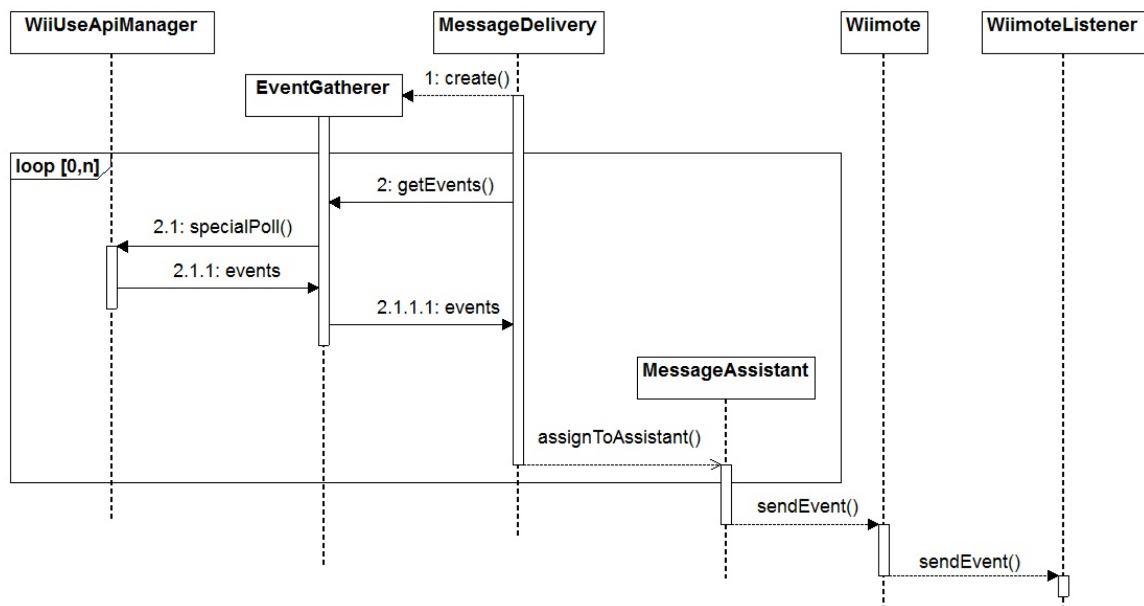

Figure 3.7. Message Delivery system

WiimoteListener Figure 3.8 shows the process where the top of the device listeners, *WiimoteListener*, takes part. In this first step, *WiimoteListener* sends a report to *PerformanceController*, which receives a time stamp. *PerformanceController* needs this information for monitoring the architecture delay, explained in detail

in chapter 4. After sending the time report, *WiimoteListener* sends the received motion packets to *PacketController*, which is responsible to check the time stamp of the motion events and built data chunks, formed by a set of *AccelerationPacket* objects. Afterwards, *PacketController* send back to *WiimoteListener* these data chunks.

Finally, *WiimoteListener* send these data chunks to *RealTimeFeedback*, which performs a motion recognition and sends the result to Software and Hardware controllers. In this case the motion recognition is performed by an algorithm based on heuristics, which computes the acceleration mean in three axis (x, y, and z) and the acceleration peaks. However, the introduction of k-Mean algorithm does not differ so much, since the architecture modules can easily be replaced.

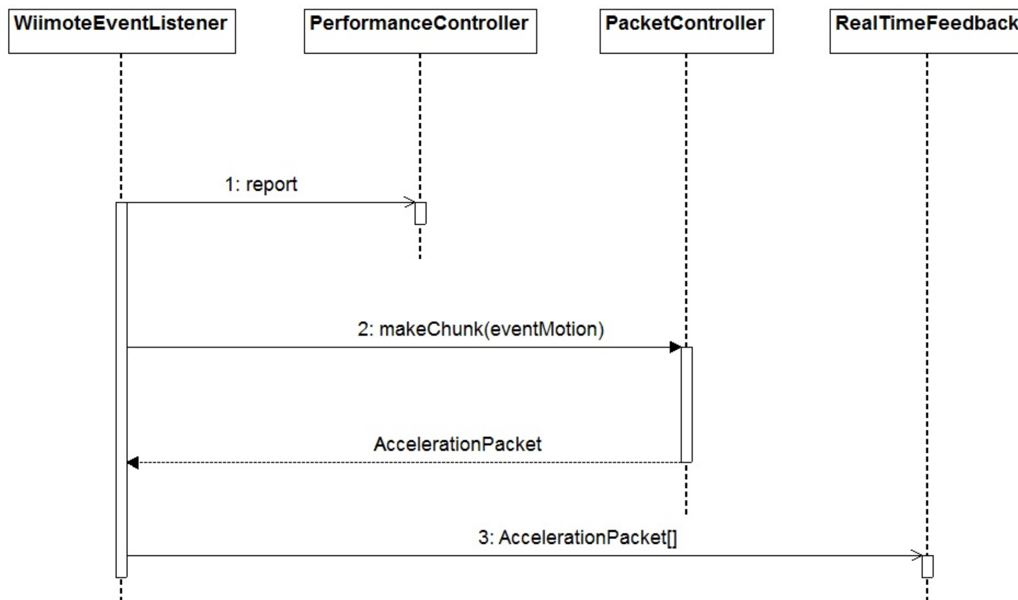

Figure 3.8. Controllers Part 1

Hardware and Software controllers The last step of the process is performed by a set of Hardware and Software controllers. Figure 3.9 illustrates this process. Firstly, *RealTimeFeedback* sends the frequency to the *PermanentSoundController*. This frequency is based on the performed motion rather than the expected motion. As introduced before, this was a requirement to provide the user a feedback of the current performed motion.

Later on, *RealTimeFeedback* sends the recognized motion to *SpellGame*, which controls the state of the game, so, which is the expected motion. If the motions,

expected and performed, match *SpellGame* will activate the device rumbler by sending an activation message to *PermanentRumbleController*. Additionally, will charge the device's energy by sendind a message to *EnergyTransmissionTask*, which may imply the activation of a LED. In this case, *EnergyTransmissionTask* sends a message to *PermanentLEDController*.

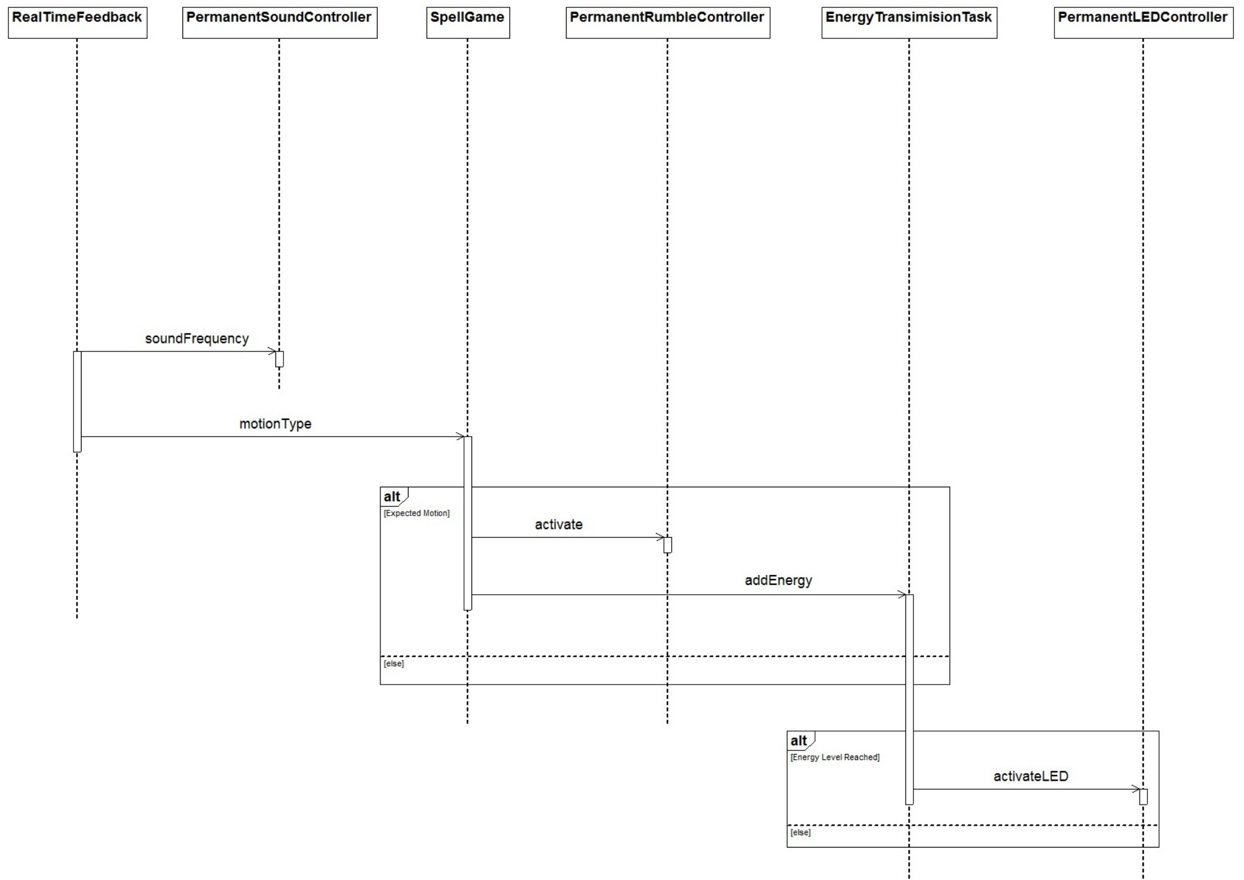

Figure 3.9. Controllers Part 2

3.3 Heuristic algorithm

In order speed up the design process of the platform, and take some statistics during the workshops, we decided to develop an easy algorithm to try to recognize the motions performed by the kids. Moreover, the different workshops allowed us to refine the inputs (parameters) for this algorithm 1.

Algorithm 1 Heuristic algorithm

$motion = NoMotion;$
if $meanX \geq N || meanY \geq N || meanZ \geq N$ **then**
  $motion = Big;$
else
  if $detectPeaks() == true$ **then**
    $motion = Robotic;$
  else
    $motion = Slow;$
  end if
end if
if $motion! = NoMotion$ **then**
  $printReport();$
else
  $logWarning(MotionNotRecognized);$
end if
$returnmotion;$

The algoritm is implemented in a class called *HeuristicAlgorithm* which receives the data *chunks*, which contains the acceleration values gathered into a period of time, from *RealTimeFeedback* component, as explained in section 3.2.3.

First of all, *HeuristicAlgorithm* takes the time which the packet has been received. This information is sent to *PerformanceController* component, which keepts track of the system delay. Afterthat, *HeuristicAlgorithm* computes the mean of the three axes accelerations contained in the recevied *chunk*. Once the three means are computed it calls to the functions which implements the motion recognition algorithm, *giveFeedback()*, which returns the recognized motion.

The following parameters have been used in the heuristic algorithm:

N

This parameter is a numeric value which defines the boundary between two different motions, Slow&Soft and Big&Large. During the workshops it has been refined in order to find the most suitable value according with the target of our research, the kids.

Motion

This variable it can take four different values i) NoMotion, ii) Slow, iii) Big or iiii) Robotic. It represents the recognized motion.

mean

This is the mean, in the three different axes (x, y, and z), of the acceleration values contained into a *chunk*.

Those are the functions used by the algorithm:

detectPeaks()

This function detects how many acceleration peaks contains a *chunk*.

printReport()

Prints some information about the recognized motion.

logWarning()

It prints that the motion has not been recognized in a warning log file.

Chapter 4

Analysis

4.1 Testing

For testing the log4j [17] framework is used. Log4j is a loggin framework which provides multiple advantages, especially in multi-thread environments (such as Java Enterprise Edition (J2EE)). For instance, log4j permits the creation of class loggers, which are instantianted in every object, and it also supports the use of general logs (instantiated by several classes). As a result, every class can have its own log, which stores the customization information in a file, and, in some cases, may print out the messages on the Java console. The customization process, by which the logs are defined and configured are set, is simple and flexible. Details of this process are described in section Introduction of [17].

4.2 System Modes

In order to simplify some testing and statistic gathering, five different run modes have been implemented. Depending on this mode, the controller component may be changed by the use of certain classes. The following run modes can be selected:

Normal

> In this mode a heuristic based motion recognition algorithm is used. This enables both *Software* and *Hardware* controller components through *WiimoteListener* classes, in order to recognize user's motions and provide feedback. This is the only mode that uses the complete architecture and it was used to support the different workshops.

Network Measurement

As explained above, the network measurement mode is used to measure network performance. This mode enables a specific listener called *WiimoteNetListener*, which sends status request packets to Wiimote and waits until the response arrives. In this process the Road Trip Time (RTT) of the network is measuredi in order to measure the delay contribution of the network communication.

Acceleration Measurement

The acceleration measurement mode is used to collect basic statistic about the acceleration values coming from the Wiimote. In this mode, the *WiimoteMotionListener* class is enabled.

Packet Measurement

The packet measurement mode was designed to collection information concerning the packets as a function of motion performed . This mode is closely related with the k-Means algorithm cluster centers, since the number of acceleration packets per second is a component of the four dimensional vector that represents a cluster center.

k-Means

Finally, an specific k-Means mode was implemented to utilize the k-Means algorithm. As explained above, this mode enables the *WiimoteKmeansListener* and this mode is used during two phases: (i) user specific motion information gathering (training), and (ii) motion recognition.

4.3 Performance

Performance is divided into different topics in order to better understand the performance of the system. The following sections will consider performance in terms of three different delays: network delay, architecture delay, and the overall end-to-end delay. Moreover a motion recognition analysis is presented, which will make possible to compare both algorithms heuristic and k-Means.

4.3.1 Network Delay

This section explores how the network affects, in terms of delay, to the proposed architecture. First of all, it is important to define the concept of network delay. In this case, the network delay includes the RTT and other delays introduced by both

devices the Wiimote and the computer. The following delays are taken into account to compute the network delay:

Java API

This delay is formed by the code execution in both ways, when the request is sent by *NetworkListener* and when the reply is received by the same component.

C API

Here, the delay covers the C API code execution which is basically the JNI call and the read and write driver operations.

Network

This is the RTT.

Wiimote

This includes the process time that the device needs to handle both the request and the reply.

As introduced in chapter 2, Nintendo Wii remote device (Wiimote) uses a Human Interface Device (HID) stack in order to send information. However, Nintendo introduced some modifications of the HID standard, which complicates the understanding of the communication protocol. The main information source, in order to design a network delay monitoring strategy, was the *Wiibrew* webpage [26], which by reverse engineering discovered most of the features of the Wiimote. The found features includes a set messages that Wiimote uses in order to communicate with other devices.

In my case, I needed synchronous communication, to measure the network delay without introducing undesirable delays. The decision was to send a *Status Request* message to Wiimote, which forces a *Status Report* reply. WiiBew proposes a high layer representation of the HID messages where each Bluetooth-HID command is in parentheses, and each two digits corresponds to a byte. The following list shows an example of the used messages:

Status Request

$(52)15XX$

The sencond parameter, *15*, is the message id (*Status Request*), and the last one is an action, such as turn off the rumble, that could be performed by the Wiimote.

Status Report

$(a1)20BBBBLF0000VV$

As in *Status Request*, *20* corresponds to the message id, in this case *Status Report*. *BBBB* represents the status of the core buttons. L is the LED status and F is a bitmask of flags (i.e., battery, extension connected ...). Finally VV is the battery level.

4.3.1.1 Network Delay Measurements

In order to measure the network performance the program needs to be executed in *Network Measurement* mode. This mode, as explained in section 4.2, enables *WiimoteNetListener*, which is responsible for sending and receiving messages.

The network measurement test case was designed to send different number of packets per test. Hence, in every test case a defined number of packets are sent, only some of which are received due to packet loss. The values are 500, 1000, and 5000 packets. Ten data collection runs were made for every test case. In the tables below the average of the specific value for these ten runs is report.

Finally, five different timout values (milliseconds) were used. This timout corresponds to the time that C API must listening before cleaning the buffer and re-starting listening process. This is a rellevant parameter because of the packet loss. With certain values, the packet loss increases dramatically. Table 4.1 shows the results from the different tests in terms of network performance (milliseconds).

Table 4.1. Network Performance

	Number of packets		
Timeout	500	1000	5000
10	21.77	23.49	29.91
15	21.66	21.53	21.24
20	21.29	21.23	21.3
25	17.08	21.36	21.1
30	19.2	21.33	21.17

Table 4.2 illustrates the minimum, maximum, and median values of the different delay values used in the tests.

Table 4.2. Network Performance statistics

	Statistic parameters		
Number of packets	Min	Max	Median
500	9.62	58.02	18.62
1000	10.16	81.36	19.96
5000	9.14	70.76	21.16

Table 4.3 shows the percentage of packet loss for each test.

Table 4.3. Packet Lost

	Number of packets		
Timeout	500	1000	5000
10	67.26	79.54	97.51
15	0.0	0.0	12.24
20	39.1	45.46	83.93
25	20.0	13.64	55.01
30	10.0	0.0	21.42

It is possible to observe a high packet loss in the different test. In most of them, the packet loss makes very difficult to have an acceptable motion recognition, since most of the packets do not arrive to the receiver. This is specially crucial in Robotic motion recognition performed by heuristic algorithm, where the lost of packets with acceleration peaks makes impossible the detection of Robotic motion.

In order to figure out the causes of these poor packet loss results, more measurements should be done. One of the important issues that should be taken into account is the Bluetooth stack performance, since this may affect the packet transmission and the motion recognition. In order to measure the different causes that may affect the correctness of the transmission, Ehsan Ullah and Stefan Witte [24] propose an scenario where following parameters are analyzed: i) Bluetooth packet delay, ii) packet loss, and iii) bit error rate. The accurate measurement of theses channel parameters, would make possible to distinguish into the packet loss contribution of Bluetooth stack, and the software architecture.

Furthermore, some results present unexpected values, for instance packet loss with 1000 packets is much lower than in the 500 packets test, with the same timeout (30 milliseconds). In order to avoid the possible initial conditions of the system, and thus, evaluate the platform's steady state, might be convenient to run longer test and more replications.

4.3.2 Architecture Delay

This section presents the delay introduced by the architecture. This is divided into two main delay contributions: the *Delivery System*, explained in depth in chapter 3, and the algorithm (heuristic and k-Means).

4.3.2.1 Algorithm Delay Contribution

The delay of each algorithm is based on how much time they need in order to convergence. For measuring the delay of both algorithms a set of motions has been performed. Each test case consist of 200 data chunks, which each data chunk represents a user motion. There were 5 test cases in order to gather enough information to take a realistic algorithm performance.

When using a heuristic algorithm the user motions (encapsulated in data chunks), are analyzed without prior system information. As a result, the algorithm simply computes mean and peak detection parameters of every received data chunk. Therefore, it is a stateless class.The k-Means algorithm was implemented to introduce some improvements (as was explained in chapter 2). The k-Means algorithm has a information gathering phase (introduced in chapter 3). This phase reduces the convergence time due to two factors: (i) the algorithm just needs to converge to a new value while processing the chunks one at a time and, (ii) the algorithm has built the cluster structure based on three different motions, hence, the convergence time is shorter, since fewer iterations need to be performed.

Table 4.4 shows the delay, in milliseconds, in five tests with both algorithms (heuristic based and k-Means).

Table 4.4. Motion recognition algorithms performance (with all times in milliseconds)

Algorithm	test1	test2	test3	test4	test5	Average
Heuristic	0.23	0.27	0.30	0.28	0.34	0.28
k-Means	3.07	3.87	3.48	2.98	3.39	3.36

4.3.2.2 System Delivery Delay Contribution

The *System Delivery* delay contribution it is an important measurement and indicates the degree of multi-thread performance. A low delay ensures the maximum throughput provided by *Executor* framework, as discussed in chapter 3.

The System Delivery Delay contribution is divided in two different phases. The first one measures the delay of the received packets before the data chunks are formed. So, the *WiimoteEvents* pass through *MessageDelivery*, *Wiimote*, and

WiimoteEventListener. Table 4.5 shows the delay (in milliseconds) as measured by an acceleration packet in the first phase, where 5 test runs of 2000 packets each were performed.

The second phase measurement starts once the data chunk has been built by *PackageController*. It is important to note that the data chunk period, or how long the period was to receive packets (i.e., to collect packets) before being wrapped into the same data structure, directly affects the measured delay. As commented before, there is a trade-off between providing fast feedback and the amount of data necessary to recognize a performed motion. The time to collect data for the data chunk was set to 250 milliseconds. This parameter value was selected based on the workshops, as explained in chapter 3, prior to the start of the game.

The initial point of measurement is the *PackageController* component, where the chunks that contain the motion are formed. The end point of the measurement is difficult to define, since this phase involves many different software and harware controllers. So, I decided to make the measurement in the *SpellGameTask* component, which is the lowest controller that always takes part in the processing, independent of the expected motion. The UML process diagram 3.9, presented in chapter 3, shows a clear picture of the software and hardware controller. In order to measure the delay in the second phase five test runs of 100 data *chunks* were performed.

Table 4.5. Acceleration packet delay in milliseconds

Delay	**test1**	**test2**	**test3**	**test4**	**test5**	**Average**
Acceleration Packet	1.95	2.0	1.06	6.02	3.52	2.91
Data Chunks	8.55	7.49	6.42	5.46	7.31	7.05

The diagram shown in Figure 4.1 shows a simplified view of the data flow of the architecture, which helps to understand the two different measurement phases.

4.3.3 End-to-End Delay

Once the two types of delays, network and architecture, were known, we need to present the overall delay results. A short end-to-end delay is required to guarante a good user experience. One of the most important parameters, in order to ensure reasonable feedback, is the data chunk (collection) time. As explained in previous sections, this is a crucial parameter, since acceleration information is needed to recognize the user motion, but, at the same time, the system needs to be fast enough to ensure application responsiveness.

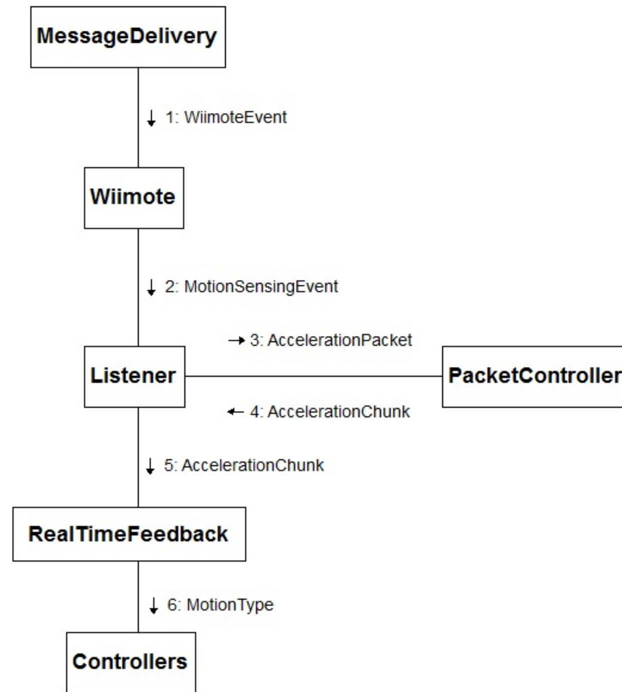

Figure 4.1. Architecture data flow

One of the main purposes of the two first workshops, as described in chapter 3, was to find out the most suitable time value for creating data chunks (from acceleration packets). As mentioned before, the value chosen was 250 milliseconds. We arrived at that conclusion after testing (during the workshops) with different time values (such as 1000, 500, and 125 milliseconds).

Another important point was to check the algorithm's performance. As shown in table 4.4 there are no major differences in terms of time, due to the improvements implemented in K-Means. Network performance also presents a very similar result, since most of the delays with the longest tests (5000 packets), present similar time values, around 21 milliseconds.

Table 4.6 presents the end-to-end delays [1] of the two different algorithms.

Table 4.6. End-to-end delay in milliseconds

		Architecture delay			
		Algorithm	System delivery		
Algorithm	**Network**		Acceleration	Data chunk	**Total**
Heuristic	21.24	0.28	2.91	7.05	281.24
K-means	21.24	3.36	2.91	7.05	284.36

4.3.4 Statistical analysis of experimental results regarding correct motion recognition

This section presents the efficiency of the implemented algorithms, heuristic and k-Mean, in order to recognize the performed motions. The motions have been performed in an unconstrained way, so, neither the device nor the context did not dictated the movements. This constraint was also used during the workshops and the "Weather Gods and Fruit Kids Game", explained in chapter 4.

In this context the correct motion should be performed, which was motion expected by the observer. For instance, during the "Weather Gods and Fruit Kids Game", the motions were randomly choosen by the platfform. In every period of time, a configurable parameter, a motion was considered the correct one. This motion was announced by the observers, and the kids tryed to perform it.

The following experiments have been performed using both algorithms in order to measure the efficiency with the same input. Ten samples of 100 packets were taken for each algorithm. The three designed motions, Slow&Soft, Big&Large, and, Robotic were tested. Moreover, k-Mean mode has been configure with three different values of packet factor (1,2, and, 3), which weights the influence of the packet mean parameter in this algorithm.

Table 4.8 shows the motion recognition efficiency regarding the different algorithms.

As we can see, the efficiency of heuristic algorithm for the two first motions, Slow&Soft and Big&Large, is acceptable, since most of the cases are properly recognized. This is due the the nature of these motions, while Slow&Soft motion has low acceleration averages, Big&Large has high acceleration averages. However, Robotic motion has a low motion recognition efficiency, lower than 50%, even when peak detection is performed.

[1] Notice that Wiimote sensor delay it is not covered

Table 4.7. Motion recognition efficiency in %

		k-Mean		
		factor 1	factor 2	factor 3
Motion	**Heuristic**			
Slow&Soft	83.9	57.8	66.9	82.5
Big&Large	85.6	66.6	76.7	92.7
Robotic	45.8	40.7	32.6	29.2

K-Mean algorithm present a different results. It is possible to observe that the packet factor, a parameter introduced only in this algorithm, it becomes crucial in the Slow&Soft and Big&Large motions. The efficiency increases and, in Slow&Soft case, provides better results than the heuristic algorithm. However, the Robotic motion is dramatically affected by the packet factor parameter, decreasing to roughly 30%.

In order to analyze why Robotic motion presents these poor results in the k-Mean algorithm, I collected some statistics that shows how the samples were classified when performing the Robotic motion.

Table 4.8. Robotic recognition efficiency interferences in %

	k-Mean		
	factor 1	factor 2	factor 3
Motion			
Slow&Soft	24.2	10.7	10.5
Big&Large	34.1	54.7	60.3

These results indicates that k-Mean algorithm has a lot of problems to recognize Robotic motion. One conclusion is the distance of the centroids. When we designed the Robotic motion, based on the statistics we took during the workshops, we located the Robotic centroid between the Slow&Soft and Big&Large movements, in terms of acceleration mean and received packet average. Hence, when a sample is not properly recognized, the algorithm classified it as either Slow&Soft or Big&Large. The packet factor simply increases the lack of Robotic movement recognition by classifying this movement as Big&Large.

A possible solution for improving the Robotic motion recognition, in case of heuristic algorithm, would be instead just detecting peaks, analyze the increment and decrement of the acceleration values in every *chunk*. By recognizing the pattern followed by Robotic motion, it is possible to have a better motion recognition.

4.4 Data process

As described before in depth in chapter 3, the platform builds *chunks* which contain the acceleration values received from the Wiimote. Those *chunks* represent a motion performed by the user in an unit of time. The designed platform analyzes these *chunks* independently.

One of the majors shortcomings is that the implemented algorithms may not recognize properly motions which happen in more than one chunk. This is because they just take care of the current incoming chunk, instead having a reference of chunks received before the current one. Therefore, by analyzing past *chunks* it might be possible to increase the motion recognition accuracy, since some motions may start at some point in the packet, not necesarely at the start of the packet. In this case, a trade of between the amount of packets use for this purpose and the introduced delay (each *chunk* add in average 280 milliseconds) should be analyzed carefully.

Chapter 5

Conclusions and future work

5.1 Conclusions

5.1.1 Goals and insights

The main goal that has been achieved is the platform design. This design supports Nintendo Wii remote device (Wiimote), and enables multiple devices interaction in a real-time environment, while providing a reasonable performance in terms of delay, as demonstrated in chapter 4.

The implemented Java platform provides a framework fulfilling the Human Computer Interaction (HCI) goals proposed in chapter 1: (i) full body movement, (ii) sensing, and (iii) awareness. Moreover, this platfform provides a good way to reuse existing components which may support future features (i.e., new motion recognition algorithms). Moreover, an extension of the C API and Java Native Interface (JNI) were introduced (i.e., to provide sound and LED functionalities).

In terms of motion recognition, a secondary goal of the research project, the achievements were reasonable considering the results presented in chapter 4. These results show that two of the three designed motions, Slow&Soft and Big&Large, are properly recognized most of the times, i.e., up to 90%. However, Robotic motion recognition shows poor results, and some algorithm changes need to be introduced in order to increase the probability of correctly recognizing of this movement.

The major insight that I gained is the the experience of working in a larger collaboration within a research project, where I had to work in team in order to achieve my goals. This team collaboration, and the reasearch framework were essential due to the cross-functional nature of the team. This collaboration required a lot of effort but, at the same time, gave me the chance to learn about other fields. In that sense, the responsability that I had to take, in terms of technical decissions, gave me the possibility to grow as a software engineer.

5.1.2 Suggestions and hints

In terms of motion recognition it is important to notice that there has already been a lot of research and publication this field. Motion recognition became popular a few years ago, and many projects have worked with the same issues. I think it is important to read relevant papers in order to have a good view of the big picture, and then make appropriate decisions based upon analyzing your own goals, the nature of the technology you must deal with and the specific problem that needs to be solved.

The software platfform that has been developed, is the major result of my thesis project. Developing this platform was crucial in order to achieve the goals of the project. It is important to have a good platfform which supports the required basic features while providing sufficiently good performance that it is usable for experiments. Once the basic functionality was working, then it was possible to focus on future improvements as these could be made incrementally - allowing the changes to be both easier and faster.

5.1.3 Modifications

If I were to do this project over again, the only thing that I would change is the device. While the Nintendo Wii remote device (Wiimote) is a very popular device, there are some aspects that need to be took into account before deciding to develop software that will use this device.

The most important aspect is the fact that the Nintendo Wii remote device (Wiimote) is not an open source project, thus, the most of the available documentation was obtained by reverse engineering. This means that there are still some parts of this device that are completely unknown to the software developer community. This makes it difficult to extend the interaction functionalities, because most of the time you do not really have enough documentation since nobody has worked on these specific parts before.

Another reason to choose another device is the limitations of is the device's capabilities. Today many devices (particularly smartphones) offer greater possibilities in terms of hardware devices (i.e., accelerometers, gyroscope, GPS receivers, ...), and also offer greater local processing capabilities. For instance, if you design a platfform that supports smartphones, these devices provides more flexibility, since they can locally run applications, which means that the local processing it is not strictly limited to returning a fixed set of information, as in the case of the Wiimote.

5.2 Future work

5.2.1 Remaining work and next steps

One of the most important issues that needs to be addressed is how to increase the Robotic motion recognition performance. As metioned earlier, this movement was not correctly recognized with either the heuristic and k-Means algorithms. Therefore, another algorithm should be proposed and evaluated.

In terms of the system, the amount of packet loss should be addressed in order to reduce the number of packets that are dropped, since this could greatly improve interaction with the user in a real-time environment. In order to do that a depeer analysis needs to be made to identify the factors that cause packet loss with the current software.

An interesting topic proposed by the projects described in chapter 2, would be to implement HMM and Kalman filter algorithms. The implementation of these algorithms will expand the possibilities of the proposed Java platform, and they may solve some problems which arose in this thesis project (i.e., poor Robotic motion recognition). This next step might be achieved by adding these algorithms to both C and Java APIs. After that, a performance comparison between the implementation of these algorithms in different tiers could be easily done.

Bibliography

[1] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[2] Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea, Brian Goetz, and Tim Peierls. *Java Concurrency in Practice*. Addison-Wesley, May 2006.

[3] BROADCOM. http://www.broadcom.com/products/Bluetooth/Bluetooth-RF-Silicon-and-Software-Solutions/BCM2042.

[4] Platfform code. http://code.google.com/p/wiimotejapi/.

[5] Guillem Duche. http://code.google.com/p/wiiusej/.

[6] Peter P. Eiserloh. An Introduction to Kalman Filters and Applications. Technical report, U.S. Naval Air Warfare Center, China Lake, California, USA, January 2002.

[7] Object Manament Group. http://www.uml.org/.

[8] Lars Erik Holmquist, Wendy Ju, Martin Jonsson, Jakob Tholander, Zeynep Ahmet, Saiful Islam Sumon, Ugochi Acholonu, and Terry Winograd. Wii science: Teaching the laws of nature with physically engaging video game technologies. In *ACM CHI 2010 Conference*, Atlanta GA, USA, 2010. ACM.

[9] Michael Kantor and David Redmiles. Creating an Infrastructure for Ubiquitous Awareness. In *Eight IFIP TC 13 Conference on Human-Computer Interaction (INTERACT 2001)*, pages 431–438, 2001.

[10] Juha Kela, Panu Korpipaa, Jani Mantyjarvi, Sanna Kallio, Giuseppe Savino, Luca Jozzo, and Sergio Di Marca. Accelerometer-based gesture control for a design environment. In *Personal and Ubiquitous Computing*, pages 285–299, 2005.

[11] In Cheol Kim and Sung-Il Chien. Analysis of 3D Hand Trajectory Gestures Using Stroke-Based Composite Hidden Markov Models. In *Applied Intelligence*, pages 131–143, 2001.

[12] Michael Laforet. http://www.wiiuse.net/.

[13] Sheng Liang. *The Java Native Interface. Programmer's Guide and Specification.* Addison-Wesley, June 1999.

[14] Rhishikesh Limaye, Hovig Bayandorian, and Shoaib Kamil. CS294 Pattern Project: Model-View-Controller. http://parlab.eecs.berkeley.edu/wiki/_media/patterns/model-view-controller.pdf.

[15] David J. C. MacKay. *Information Theory, Inference, and Learning Algorithms.* Cambridge University Press, 2005.

[16] Jozef Mlích. Wiimote gesture recognition. In *Proceedings of the 15th Conference and Competition STUDENT EEICT 2009 Volume 4*, pages 344–349. Faculty of Electrical Engineering and Communication BUT, 2009.

[17] Apache org. http://logging.apache.org/log4j/.

[18] Craig Ranta and Steve McGowan. *Human Interface Device (HID)*, 2003.

[19] Hans Rohnert, Peter Sornmerlad, Michael Stal, Frank Buschmann, and Regine Meunier. *Pattern-Oriented Software Architecture:A System of Patterns.* John Wiley, February 2001.

[20] Takaaki Shiratori and Jessica K. Hodgins. Accelerometer-based user interfaces for the control of a physically simulated character. In *ACM SIGGRAPH Asia 2008 papers*, SIGGRAPH Asia '08, pages 123:1–123:9, New York, NY, USA, 2008. ACM.

[21] Jakob Tholander et al. Generalized interaction models. http://www.mobilelifecentre.org/project/show/6.

[22] Javier Torres, Brendan O'Flynn, Philip Angove, Frank Murphy, and Cian O' Mathuna. Motion tracking algorithms for inertial measurement. In *Proceedings of the ICST 2nd international conference on Body area networks*, BodyNets '07, pages 18:1–18:8, ICST, Brussels, Belgium, Belgium, 2007. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[23] Yang Wai-Chong. 3D Spatial Interaction with the Wii Remote for Head-Mounted Display Virtual Reality. In *World Academy of Science, Engineering and Technology*, pages 377–383, 2009.

[24] Ehsan Ullah Warriach and Stefan Witte. Approach for Performance Investigation of different Bluetooth Modules and Communication Modes. In *International Conference on Engineering Technologies*, pages 167–171, 2008.

[25] Greg Welch and Gary Bishop. An Introduction to the Kalman Filter. Technical Report TR 95-041, University of North Carolina at Chapel Hill, Department of Computer Science, 2006.

[26] Wiibrew. Wiibrew. http://wiibrew.org/wiki/Wiimote.

TRITA-ICT-EX-2011:165

www.kth.se