Integrating the Wii controller with enJine: 3D interfaces extending the frontiers of a didactic game engine

João Bernardes      Ricardo Nakamura      Daniel Calife      Daniel Tokunaga      Romero Tori
Universidade de São Paulo, Dept. de Engenharia de Computação e Sistemas Digitais, Brazil

Abstract

The goal of the work described here is to integrate a 3D device, the Wii Controller, and enJine, a didactic engine, motivated by the growing use of 3D interfaces. The paper discusses how this increases enJine’s didactical and technological potential and details the adopted solution as a layered architecture. Two interaction styles were tested. Test results show a variety of data about the controller, that this solution works as desired and that using it to modify a game to use the WiiMote as its input device is a simple task.

Keywords: Tangible User Interface, Gesture-based Interface, Game Engine, Didactic Tool

1. Introduction

This paper describes a work whose main goal is to integrate a 3D device, the Wii Controller (also called Wii Remote or WiiMote), and enJine, a didactic 3D game engine. The advantages of using a game engine, and therefore games and game programming, as a didactic and motivational tool for students have been discussed by several authors. A similar discussion has been presented before including the use of enJine [Tori et al. 2006] and focused in teaching computer science or engineering. But what should be the advantages of adding a 3D input device in such a setup?

Games for non-portable platforms have, for several years now, been in their large majority 3D. Even for portable platforms a tendency towards 3D games exists, as shown, for instance, by the 3D engine for cellphones called M3GE [Gomes and Pamplona 2005]. This tendency is still slow only due to hardware cost. Using 3D game programming when teaching is not only useful to work with spatial concepts, necessary to the target students and some courses, such as Computer Graphics, but also helps in student motivation, since they tend to perceive these games as the "best" ones and their creation as a worthy challenge. Conventional input devices, however, such as joysticks, mouse and keyboard, were designed primarily for 2D applications and their use in 3D requires more sophisticated techniques and training. While this is usually not an issue for those already trained (such as avid gamers or users of 3D modeling or CAD applications), it is a barrier for users new to gaming. Non-conventional, 3D input devices and user-interface techniques address this issue by providing a more natural (and arguably fun) way of interfacing with 3D applications, and their use is already a strong tendency in electronic games, as demonstrated by the success of Nintendo's WiiMote [Nintendo 2007].

Wii is the last generation console from Nintendo, competing with Microsoft's XBox 360 and Sony's Playstation 3 (PS3). Wii has admittedly less processing and graphics power than its competitors and, despite that, has met unexpected success in the market. In the US alone it sold 600 thousand units up to 8 days after its launch, and 4 million units in 6 weeks [Dobson 2006]. The console was soon missing in the market and being sold by values overpriced by up to 40%, approximately. These numbers are far superior to those attained by Wii's competitors. The XBox 360, for instance, sold approximately 3 million units worldwide up to 90 days after launch [Carless 2005]. The natural and fun 3D interface embodied in the WiiMote is considered the main reason for this performance despite its otherwise inferior technical specification (Wii's lower price is also a factor). Several users have even bought both a Wii and a XBox 360, a pair
 nicknamed "Wii60", to have both the 3D interaction and high-end graphics to play with. It is interesting to notice that Interlab had already been researching and publishing about new 3D interfaces for games in enJine for a while before the excitement over WiiMote. Soon after its appearance, Sony also announced motion-sensing capabilities for PS3's controller had been in research for over 2 years and would be present at launch. All this indicates that 3D and more natural interfaces are indeed a strong tendency in games (and hopefully in other applications as well).

This tendency is one of the reasons to integrate the WiiMote and enJine (as well as for other enJine extensions discussed in section 3). As a didactic tool for several computer science or engineering and even game programming courses, enJine must always strive to be up-to-date with the evolution of interface and game technologies. This not only influences student motivation, but also offers a broader range of options for each course. User Interface courses at both undergraduate and graduate levels, for instance, are often limited to 2D devices and techniques, but with the use of enJine and the WiiMote they could cover some 3D techniques as well, such as tangible interfaces. At a lower level of abstraction, even signal processing (to treat the raw data obtained from the WiiMote) could be explored didactically.

Another motivator for this integration is the fact that enJine's simplicity (one of the main requirements for its goal as a didactic tool) and flexibility, especially in its abstract input layer, reduce the complexity of such work. Finally, enJine has secondary goal. While its main objective is to be an effective teaching tool, its simplicity and flexibility have lead to a new goal: to serve as a test bed for new technologies in games. Lately it has been used primarily to explore Augmented Reality (AR) - a form of interface that combines real images, seen directly or captured in video, and virtual elements in a scene interactively and in real time with 3D registration between real and virtual elements [Azuma 1997] - and other new interfaces in games.

While several other 3D devices, such as 3D mice (magnetic or gyroscopic), could have been chosen as an input device for enJine, the choice of the WiiMote was not merely driven by "fashion". The main factors for this choice were the low cost and especially the availability of this controller, which, unlike other 3D devices, is widely accessible to the general public. Before a more detailed discussion of WiiMote, it is important to notice now that, aside from its use as a regular wireless game controller, it provides tracker-like data (what it actually measures is accelerations in 3 axes; it can also be used as a pointing device thanks to an infrared system), which can be used to implement both tangible and simple gesture-based interfaces.

Tangible User Interfaces (TUI) are defined as Computer-Human Interfaces that use interactions with

the physical world as metaphors for information manipulation in the virtual world [Ishii and Ulmer 1997]. In these interfaces, a physical and a virtual object are associated so that changes in one (usually manipulation of the physical object) should affect the other. This association of the WiiMote with virtual objects in games is quite clear in several cases, such as when using the controller to represent pointing a gun, swinging a bat, a tennis racket or a sword. There are even kits of simple plastic devices that can be securely attached to the Wii to make it look like these objects. Often, however, there is not such a direct correspondence between the manipulation of the WiiMote and the in-game object and the gestures made by the user are what actually serve as a game input.

Considering that, for gestures, the WiiMote is nothing but a hand tracking device (both hands can be tracked if using the Nunchuk, an accessory that can be attached to the WiiMote), it only allows the use of pointing gestures or gestures composed of different hand trajectories or relative positions between hands, as in the case of the system proposed for Cabral et al. for use in Virtual CAVE Environments [Cabral et al. 2005] (actually, Cabral's system uses 2D trajectories, while Wii can track in 3D, but the limitation to trajectories or relative positions is the same). Differences in hand pose cannot be accounted in such systems (recognition of gestures combining trajectories and pose usually require either complex computer vision algorithms or the use of datagloves considered awkward by many users), limiting the space of possible gestures. With the WiiMote, associating trajectory-based gestures and button inputs may reduce this limitation, but not overcome it.

Regardless of this limitation, the WiiMote, as a 3D device either for tangible or gesture-based interfaces, can be useful to enJine, to explore 3D interfaces during certain courses, to keep enJine up-to-date with game technologies, for student motivation or simply to explore the limits of these new interfaces within games or Augmented Reality.

The next section discusses related work, before section 3 discusses enJine (specially its input layer) and 4, the WiiMote. Section 5 explains in detail the solution adopted to integrate them (illustrated in Figure 1). Finally, section 6 presents the results of tests and prototypes based on this solution and the final section shows conclusions derived from this work.

2. Related Work

Nintendo's Wii and its controller have been available to the public for a relatively short time, therefore there is not much published work to be found about its use. Before presenting this work, however, this section discusses some underlying questions, such as tangible and gesture-based interfaces (particularly the problem of gesture recognition with the WiiMote) and games that use them. Architectures with flexible input layers,
like enJine, that could probably make use of the WiiMote just as simply, are also described.

Tangible Interfaces were defined previously as those where there is a correspondence between a real and a virtual object. This allows the user to manipulate the virtual objects in 3D through touch and manipulation of the real object, in a very natural manner. Compare, for instance, the task of rotating a real object with one's own hands to get the same movement in a three-dimensional digital model and of to using a mouse to select the virtual object and then using some complex interface to rotate it in three axes. Tangible interfaces also allow and even stimulate collaboration between users by allowing movement and interaction with fewer constraints, with both hands and even the body. The "ownership" of a virtual object, for instance, can be instantly determined by seeing which user holds its physical correspondent.

In TUI applications, it is important to keep registration between physical and virtual objects. There are many ways to accomplish this. The WiiMote uses acceleration sensors. Another common solution is the use of computer vision and fiducial markers (distinctive visual markers on objects) as is done by the popular ARToolKit [Billinghurst et al. 2000].

For tangible interfaces to become intuitive and seamless, with little need of training for their manipulation, it is necessary to choose physical objects that are already well-known to the targeted users (that is probably one of the reasons why the Wii controller so closely resembles a TV remote) and also make use of metaphors that appear natural to these users, thus eliminating the need to learn new ways of virtual interaction, as users only have to rely on abilities naturally developed during the course of their lives.

Gesture-based interfaces have been researched for even longer than TUI, but only in the past couple of decades enough computing power has been more accessible to bring them to reality, and it is an area where intense research is done to this day. It is curious that after all this time, it is entertainment that is popularizing this sort of interface.

Gesture-based systems require two main tasks to be accomplished: analysis and recognition. Analysis is the task of gathering data about the gesture and is usually accomplished either through active position, bending or acceleration sensors in the user's hands (datagloves are a common example) or through computer vision techniques, an usually more complex process, but which frees the user from any awkward devices. Computer vision-based analysis is especially adequate for AR applications, since a video camera is already often used in these applications when combining real and virtual images. Gesture recognition is a particular problem in the field of pattern recognition. Given the data acquired during the analysis phase, the system usually must classify the gesture as one element of a known set. The problem is complicated due to the variations that exist in every gesture, even when performed by the same user. Statistical and Artificial Intelligence (AI) techniques are often used for recognition.

Reviewing the extensive body of work in this area is beyond the scope of this paper, especially when there are good reviews available about gesture-based interfaces in general [Huang and Pavlovic 1995; Marcel, 2002] or for vision-based systems [Pavlovic et al. 1997; Wu and Huang 1999]. Of particular interest, however, is how these tasks are performed by Nintendo's Wii. A company specializing in AI for entertainment, AiLive, was approached by Nintendo and developed a tool called LiveMove [AiLive 2007] that uses context learning, a machine learning technique, to interpret Wii motion data as gestures. Game developers build a "motion recognizer" for each gesture they wish to classify in a given game by providing training samples of the gesture and can, therefore, use gesture recognition in a flexible and efficient way without needing expertise in the complex field of motion interpretation. AiLive claims the main advantage of this tool is its performance. It is able to recognize up to 40 different gestures on 8 Wii remotes or nunchuks simultaneously at 60 frames per second using less than 5% of the Wii's modest CPU power and typically less than 700kb of memory. This proprietary solution can be either supplied by Nintendo to authorized Wii developers as part of the development toolkit or acquired directly from AiLive and supposedly works with other motion-sensing controllers, as well as with MS Windows XP.

For open-source and didactic work, as in the case of enJine, using black-box proprietary tools such as LiveMove is much less advantageous than for the game industry, however, and developing alternatives to LiveMove is not only a necessity, but also an interesting future work. Works that discuss how to obtain accurate position measures from WiiMote's acceleration data, for instance using Kalman Filters [Rasco 2007], are useful in that direction.

Long before Nintendo's Wii and WiiMote, researchers saw the potential of 3D, natural interfaces in games, particularly in Augmented Reality games and related several projects that use tangible interfaces or gestures. Here, only a few examples of these projects are presented, to illustrate their different approaches. The CAT/STARS platform [Magerkurth et al. 2004], False Prophets [Mandryk et al. 2002] and Monkey Bridge [Barakonyi et al. 2005] use tangible interfaces in board-like electronic games, respectively using RFID tags, infrared diodes and fiducial markers. AR sports games, such as the Golf Simulator [Govil et al. 2000] often use sport equipment itself, such as a golf club, as tangible interfaces. AcquaGauntlet [Oshima et al. 1999] uses sensor-based gesture recognition as input while AR PushPush [Kim et al. 2005] uses computer vision to recognize simple, static gestures.
It is also important to review other existing solutions to adapting novel input devices to existing software systems. Two different approaches to this problem can be identified: the creation of an abstraction layer to decouple input devices from stimuli processing and the emulation of well-known input devices through an adaptation layer.

The CIDA architecture [Farias et al. 2006] is an example of the input abstraction layer approach. CIDA defines a set of input device classes based in physical characteristics of those devices. An application developer using CIDA interfaces with one of those abstract classes, instead of directly handling input from the physical devices. This way, any input device with a suitable plug-in for that device class can be used in the application. The ICON system [Dragicevic and Fekete 2004] also employs a device class model for input devices, although it is more focused on the representation of user interactions through multiple devices. The Haptik library [Pascale and Prattichizzo 2005] defines an abstraction layer based for a specific class of input devices. Much like CIDA, devices in Haptik are represented by software components that communicate with the host application through a standard interface that exposes the device attributes. The advantage of this approach is that the application becomes inherently ready for new input devices. It also encourages programming the user interface based on intended user actions, instead of specific physical inputs. However, using an abstract input layer requires learning a new programming library and its conceptual model, which may increase development time and program complexity.

The GlovePIE software [GlovePIE 2007] follows the emulation approach. Essentially, that software is capable of reading input data from several different input devices and generating events to the operating system as if they had come from a keyboard, mouse or joystick. A gesture captured through a virtual reality glove, for instance, could be used to generate a mouse button press event. The conversion between incoming input signals and outgoing events is controlled by a custom script language. An advantage to this approach is that it is possible to use new input devices in software that was written to support only conventional devices such as keyboard and mouse. Furthermore, there is no need to learn a new programming library to handle different input devices. However, if a scripting language is used, it must be learned, and may be limited in the input signal conversions that can be expressed. Another disadvantage to this approach is the introduction of additional processing, due to the signal conversion and event generation.

Finally, there are a couple published works that relate the use of WiiMote. Curiously, both use it in musical training applications. A Master's Dissertation analyzes its use for interactive percussion instruction [Belcher 2007] and describes interesting details of the WiiMote's workings and WiiRemoteFramework, part of the WiiLi project [WiiLi 2007]. The Pinocchio system [Bruegge et al. 2007], on the other hand, allows the use of the WiiMote as an input device to conduct a virtual orchestra.

3. EnJine

EnJine is an open-source game engine based on the Java 3D API, intended to serve as a didactic tool and described in detail in other works [Nakamura et al. 2006]. EnJine is aimed at teaching game design and computer science, especially computer graphics [Tori et al. 2006] and software engineering subjects, to undergraduate or graduate students. It is also used as a test bed for research in game development, virtual and augmented reality and other related fields. The software and additional documentation are available at http://enjine.incubadora.fapesp.br/portal. Some of its chief features are:

- 3D rendering based on Java 3D (which, in turn, currently uses OpenGL);
- Multi-stage collision detection (currently only implementing subspace and bounding volume filters) and ray-casting collision detection;
- An abstract input layer to simplify the use of non-conventional input devices;
- Core classes constituting a game's basic architecture: the game, its stages, game objects (which are very flexible) etc.;
- A framework package to facilitate the creation of certain games (currently only single-player games);
- Separation between graphical and logical update cycles.

![Figure 2. The input abstraction architecture in EnJine](image)

In this paper, particular attention needs to be given to EnJine's input layer. It implements a simple architecture for input abstraction, presented in Figure 2. Different input devices are accessed as concrete implementations of the InputDevice class, which in turn exposes its capabilities as a collection of InputSensor objects. Each InputSensor has a numeric identifier, a description and a floating-point value that can be used for both discrete-state sensors, such as buttons, as well as multi-valued sensors such as...
position trackers. Each InputDevice implementation is responsible for updating the state of its sensors.

The InputManager class maintains a registry of InputDevices and can be queried by the user application. It also maintains a collection of “bindings” that relate each InputSensor to an InputAction. InputActions are objects that represent user commands with a specific intent within the application. For instance, there might be “jump” and “run” InputActions. Periodically, the InputManager updates the state of the InputActions according to the state of the InputSensors. The state of the InputActions may be accessed by other game components to execute desired user commands.

This simple yet flexible input layer was created at first merely to simplify the configuration of game input options in conventional devices and to accommodate these devices but has proven particularly adequate to interfacing with non-conventional devices, such as cameras tracking fiducial markers, color-marked objects or gestures. Currently, 3 projects extend enJine in this direction: integrating enJine and jARToolKit, Robot Arena and integrating enJine and HandVu.

The first project using non-conventional interfaces and enJine was to integrate it with jARToolKit (a port of ARToolKit to Java) and creating several game prototypes, including a somewhat complex dance game named “We AR Dancin’”, to test the possibilities of using fiducial markers and computer vision as input devices for AR games [Tsuda et al. 2007]. These tests revealed several possibilities, limitations and particularities of this sort of interaction, such as the importance of visual feedback to the user. While during these tests the fiducial markers were mostly used for registration purposes in simple gesture-based interfaces, the resulting system can just as easily be used with markers on tangible interfaces. Aside from enabling enJine to aid when teaching 3D interfaces and AR concepts (such as its definition and the concept of registration), at a lower level of abstraction this work allows the exploration of image processing concepts (for instance, to remove video background).

Robot ARena [Califé et al. 2007] is an infrastructure that can be used as a platform for development of innovative indoor Augmented Reality games based on tangible interfaces or a wireless controlled robot and Spatial Augmented Reality (SAR), which brings the visualization of the augmented content directly into the physical space [Bimber and Raskar 2005]. The main goal of this project was to develop and test that infrastructure, exploring new ways of combining several techniques of low-cost tracking (such as by color, infrared or ultrasound), robot control and SAR. All these subjects can be explored didactically by this extension of enJine.

Finally, a work currently in progress aims to integrate enJine and HandVu [Kolsch and Turk 2005], a real time gesture recognition library able to handle both hand tracking (for trajectory-based gestures) and to recognize 6 different hand poses. The main didactic purpose of this work is to study the potential and limitations of gesture-based interfaces, especially when it is possible to recognize not only movement but also different hand poses.

4. WiiMote Specifications

Nintendo aimed to match the familiarity of a remote control and motion-sensing technology to make gaming accessible to a broader audience and the result was the WiiMote, considered the most important reason for Wii's success. The remote also includes a speaker, a rumble feature and 4 blue LED's for feedback as well as an expansion port for additional input devices, such as the Nunchuk, which are used in conjunction with the controller. The Nunchuk contains the same motion-sensing technology enabled in the Wii Remote but also includes an analog stick to assist in character movement. Using Bluetooth technology, WiiMote sends user actions to the console from as far as 10 meters away. As a pointing device, it can send a signal from as far as 5 meters away. Up to 4 Wii Remotes (plus 4 accessories such as the Nunchuk) can be connected at once and the controller and accessories can be used ambidextrously [Nintendo 2007].

According to the WiiLi Project [WiiLi 2007], an unofficial but rather complete source of information about the WiiMote, it follows the Bluetooth Human Interface Device (HID) standard and sends reports to the host with a maximum frequency of 100Hz. The controller does not require any of the authentication or encryption features of the Bluetooth standard. In order to interface with it, one must first put the controller into discoverable mode by either pressing specific buttons. There are 12 buttons on the WiiMote. Four of them are arranged into a directional pad, and the rest are spread over the controller. When a button is pressed or released, a packet is sent to the host via a specific HID input report containing the current state of all the buttons. This state is also included in all other input reports. The motion of the remote is sensed by a 3-axis linear accelerometer capable of measuring accelerations over a range of ±3g with 10% sensitivity. The sensor uses a right-handed coordinate system with the positive X-axis to the left, the positive Z-axis pointing upward, when the remote is held horizontally, as show in Figure 1. Inputs on each axis are digitized to 8 bit unsigned integers. WiiMote's motion sensors are not accurate enough to use the remote to control an on-screen cursor. To correct this, Nintendo augmented the controller with an infrared image sensor on the front, designed to locate two IR beacons within the controller's field of view. The beacons are housed within the “Sensor Bar”. WiiLi also documents the complete list of report types and the corresponding data transmitted in each report. Currently, the sound
data format that must be sent to the speakers is WiiMote's least known aspect to the general public.

5. Implementation

To integrate enJine and the WiiMote, the first task was to find a way to connect and communicate with the wireless controller. The first attempt was thwarted by the use of Bluetooth. This approach, however, met with several difficulties. First, while J2ME faces a less problematic situation due to the JSR 82 Bluetooth specification [Thompson et al. 2006], for PC applications using the JSE there is no implemented official Bluetooth API for Java yet. Several proprietary (and not free) APIs exist [Java Bluetooth.com 2007], but using such solutions is unthinkable for extending enJine with its open-source and free nature. An excellent library for Java, Avetana [Avetana 2007], is only free for a 14 day trial or in Linux. Trying some of the free Bluetooth communication libraries, both for Java and for C/C++, led to a few disappointments. Bluesock [Bluesock 2007], for instance, works fine with RFCOMM devices, but not with USB Bluetooth dongles using other protocols. Even Windows-specific solutions were considered, using the Bluetooth management and sockets support for Windows. However, while studying this solution, a simpler (but still platform specific) alternative was found.

The solution was to leave Bluetooth communication and management for the operating system (in this case, Windows) to manage, since it can manage all Bluetooth protocols, profiles and stacks transparently. In Figure 1 this is represented by the lowest, gray layer. This solution first requires, therefore, Windows to recognize the WiiMote as a device and connect to it (as mentioned before, for this particular work a USB Bluetooth dongle was connected to handle communication). This is not as straightforward as it may seem, and a detailed tutorial was written after some experimentation and with the aid of WiiLi. This tutorial is available in enJine's homepage.

Once the WiiMote is connected via the operating system, it can be treated as any other device, regardless of the nature of its connection. It is possible, then, to write a simple driver for it, using the Windows Device Driver Kit (or DDK). Figure 1 shows this driver as the blue layer. It is written in C/C++ and implements 3 basic functions: initialize finds the available Bluetooth devices and reserves some memory to be used during the communication while read and write are responsible for the actual transfer of data between the WiiMote and the computer (more details about the usage of these functions are documented along the source code, which can be found in enJine's homepage). Using JNI, this functionality is encapsulated in a DLL and made available to Java applications. Currently, to simplify implementation, this driver has some limitations: first, it is specifically built to communicate only with the WiiMote - the size of the reports used by read and write are hardcoded to the size used by Wii (22 bits). Modifying the code to allow a varying report size would be relatively simple and would allow the driver to be used with any Bluetooth device. A second limitation is that the initialize function only searches for one device (it stops searching after the first is found) and therefore multiple devices cannot be handled. This is also a relatively simple modification that will be addressed in future works. The C/C++ communication driver code was based on the cWiimote library [Forbes 2007].

This driver provides, by design, only low level, general functionality, however. A Java class named WiiMote (represented by the yellow layer in Figure 1) is then added to the architecture to parse, interpret and store this data. To read data from the controller, the user must first call the method setReportType to set the kind of information the controller must send. After that is done, each call to the read method requests this data from the WiiMote, interprets it and stores it as attributes that represent the last controller's state and can be read through access functions. To modify WiiMote's outputs (such as rumble or LED states) a similar procedure is followed: setReportType is called and attributes representing the states to be changed are modified through access functions, then calling updateOutput sends this data to the WiiMote.

This is all that is necessary to connect the WiiMote with enJine (or any Java application). From here on it is a matter of extending the functionality of enJine's input layer by creating InputDevices for the WiiMote. These devices, illustrated in pink in Figure 1, are specific to the each game and may range from the very simple, where each InputSensor reflects raw data representing the controller's state, to sophisticate devices that use pattern recognition to interpret user movements and associates each InputSensor to a given gesture. The game (purple in Figure 1) then associates these sensors to InputActions, just as it would do with a conventional device like a keyboard, and treats these actions normally.

To test the solution's implementation as well as to serve as proofs of concept and ready-made devices that can be used directly or as models by students or developers, two simple devices were developed. One implements a very simple gesture-based interface while the other functions as a TUI. Both devices work simply with acceleration, without integrating it or using complex pattern recognition techniques. This simplicity is actually important, to avoid confusing a new user who tires to understand the basics of how such a device works before trying anything more sophisticate. Both implemented devices also contain an update thread that periodically queries the WiiMote about its state. These threads conflict with each other when accessing the same controller, so it is currently impossible to combine two InputDevices to obtain a
more complex functionality. Each of these devices can be understood as a different style of interaction.

The first device is called WiimImpulseDevice and translates strong, fast movements with the controller as directional commands. Flicking the WiiMote up activates the "up" InputSensor and the same goes for right, left, and down. When returning the control to a rest, default position (i.e., horizontal, face-up and pointing forward) the movement is less brisk and does not "undo" the last motion. A problem arises in this device due to working purely with acceleration data, instead of velocity or position: every time the controller is flicked in a direction, it suffers a strong deceleration in that direction, but also an even stronger acceleration to make it stop. If acceleration data was used directly, this would immediately "undo" the last input intended by the user. To address this matter, WiimImpulseDevice ignores a negative acceleration peak right after a positive peak (i.e., a positive movement) and vice-versa.

The second device, called WiitTiltDevice, simply assumes that only slow, gentle movements are applied to the WiiMote, so that gravity is by far the strongest acceleration (actually, the acceleration caused by force applied by the user’s hand that resists gravity - it has the opposite sign) to calculate the controller's pitch and roll based on how gravity is decomposed in the X and Y axes. Yaw is not measured (it would require the use of the infrared sensor). WiitTiltDevice could be used for games requiring subtle and continuous motions for control, such as controlling the flight path of a glider through its inclination. To avoid calling the \( \sin \) method at such a high update rate as WiiMote’s, \( \sin \) is linearized as \( \sin \alpha = \alpha \). This is reasonable for low values of \( \alpha \) but becomes a bad approximation as it increases. At higher angle values, however, the WiiMote accelerometers themselves lose resolution as their direction grows closer to gravity’s and high angles should, therefore, be avoided for this particular device anyway. The approach of using \( \sin \) in the calculations was actually implemented, tested and even found acceptable, although the lag it causes even in a simple application was noticeable (but not severe).

Two simple prototype applications were built with enJine to test these two devices. They merely draw a ColorCube on a black background (that is why such a cube is shown as the prototype in Figure 1) and have its movement controlled by WiimImpulseDevice or WiitTiltDevice. Finally, a previously existing, simple enJine game was modified to use as its input WiimImpulseDevice or WiitTiltDevice (a rather simple modification, simply replacing in the code the names of the old keyboard devices and sensors with the new ones - the actions and the rest of the game remain the same). In this action puzzle, called RollCubes and shown in Figure 3, several ColorCubes exist in a plane and up to two users may control one at a time with a cursor (the cones in Figure 3), rolling it up, down, left or right on its faces. If two or more cubes become adjacent with the same face up (each face has a different color), all of them begin to slowly disappear. The goal is to clear all ColorCubes as quickly as possible.

6. Results

The first piece of data tested in this work was WiiMote’s refresh rate. It is listed nominally as 100Hz and it was necessary not only to be sure of this number, but also to verify if the integration with enJine and all those layers did not lower this rate significantly. It was indeed confirmed that the refresh rate is 100Hz (this was the value used in all further tests and devices) and that the integration with enJine had not lowered it. When attempting to query data at a faster rate, especially immediately after initialization, caused errors (but no crashes or exceptions). It was decided, then, that for all InputDevices using the WiiMote, the thread polling the controller’s state would sleep for 10ms (at least) between each query.

Before implementing the devices and prototypes described in section 5, a simple application that records raw data from the WiiMote was also built, to analyze this data. Some of the most useful analyses came from the controller in a static position, because then only a known acceleration (1g) is applied to it. Figure 4 shows its response to gravity in 10ms intervals over a period of 5s. In the first 2s, the controller is resting on it’s back on a flat surface, thus the positive acceleration in the Z axis, indicating the vertical force that is being applied on the WiiMote by the surface to resist gravity. Then the controller is turned to rest on its left side and the force is applied along the X axis. It’s interesting to notice that while this apparent inversion of the sign for measured accelerations happens only with resisted field-generated forces, such as gravity. For forces directly applied, by the user’s hand for instance, the acceleration is measured with the expected sign.
From this test, the offsets in measured acceleration for each axis and the value of \( g \) can also be determined. For this particular controller, they are approximately 12, 10 and 8 for the X, Y and Z axes, respectively and \( g \) corresponds to 28. In the 8-bit word used to record accelerations by WiiMote, the range of possible values goes from -127 to 128, which is more than enough to accommodate the nominal 3\( g \) it can measure. These values for offsets and gravity vary for each controller and can be obtained through a specific HID report.

Another fact that can be noticed in Figure 4 is how errors in the measures of acceleration are relatively small when the controller is in repose, but can be considerably high (even higher than the 10% nominal sensitivity) during or right after movements. Finally, this test can also show the directions of the positive axes, which were found to correspond to those described by WiiLi and in section 4.

Figure 5 shows the accelerations measured by the WiiMote during 3.5s, while a single, wide arm swing was applied from right to left, with the controller held horizontally and pointing forward. The acceleration in the Z axis (pointing up) corresponds to the resistance to Gravity. In the X axis there is first a positive acceleration pushing the controller from left to right, then an even higher deceleration to make it stop (this behavior was discussed previously and is taken in consideration in WiiImpulseDevice). This was a "weak" swing, thus the low positive acceleration in X. The most intense (and interesting) acceleration, however, appears along the Y axis, which points to the user. So the arm swing can make the controller move in an arc (as in fact happened), this radial acceleration must be applied. It corresponds to the centripetal force responsible for the circular motion. It is interesting to notice that as soon as the circular movement stops, this acceleration reaches zero. Unlike for linear movements, there is no need for a deceleration to match the centripetal force before the controller can reach repose. The small variation in the Z acceleration has the same behavior as the X acceleration (i.e. it corresponds to a linear movement) and indicates that the movement was not perfectly parallel to the X axis: either the controller was slightly tilted around the Y axis during the movement (a strong possibility) or the arm movement had a vertical component, as well as an horizontal one.

These tests led to the InputDevices described in section 5. The implementation of WiiImpulseDevice was quite straightforward after understanding Figure 5. On the other hand, while inspired by Figure 4, implementing WiiTiltDevice was more complicated due to the need to decompose the acceleration along the axes of the moving coordinate system. Both devices worked as designed during testing.

No usability tests were made for the game modified to use WiiImpulseDevice or WiiTiltDevice. Informally, however, it can be noted that using WiiImpulseDevice in a game that requires several quick and repetitive movements is hard on the user's wrist but using WiiTiltDevice led to a pleasant interface, but still slower than simply using the keyboard. Then again, modifying this game to use the WiiMote was not accomplished to improve the game's playability (it is actually the sort of game that works with conventional devices very well), but simply to analyze how much coding work is required to replace a conventional device with a pre-made WiiMote InputDevice and the result was positive: the work involved is very small, as discussed in section 5.

7. Conclusions

This paper described the integration of the WiiMote and enJine, successfully achieving the proposed goal. This integration (as well as other projects such as Robot ARena and the integration with ARToolKit and HandVu described previously) serves the double purpose of allowing the exploration of innovative game technologies for didactic and research purposes, as well as keeping enJine up-to-date with the trend of 3D, more natural interfaces in games. This not only increases student motivation, but also expands the
possibilities to use enJine in teaching, particularly for courses involving AR, user interfaces or games.

The integration solution chosen has multiple layers and, at the lowest level, makes use of the operating system’s handling of Bluetooth communication, since finding a comprehensive and free library for this function was problematic. A more efficient solution was to write a simple device driver to the WiiMote using the Windows DDK and allowing Java to access it using JNI. This driver comprises the second layer of the architecture. The third layer, already in Java, is a class that interprets WiiMote’s raw data, stores its state as attributes and provides a simple interface to send outputs to the controller. With this functionality, using the WiiMote becomes just as simple as using any other device. Two sample InputDevices were created and an existing game was modified to use them.

Testing helped ascertain the unofficial data about the WiiMote, such as its refresh rate and coordinate system, as well as showed that the implemented architecture did not impair this refresh rate significantly. It was found that, while in repose or under low accelerations, errors in acceleration measures by the controller were relatively small, but under higher accelerations they were significant (even surpassing the 10% nominal sensor sensitivity). Testing was also important to understand and visualize WiiMote’s response to user movements to implement WiiImpulseDevice and WiiTiltDevice.

While the project’s goal was satisfactorily reached, the implementation still has a few limitations. Perhaps the most important is the dependence of the Windows operating system. Because Bluetooth communication was isolated in the lowers layers of the proposed architecture, however, once a suitable and free Java API for this task is found (or, in a worst case scenario, developed internally), it should be relatively simple to use it to make the system platform-independent. Until then, however, the Windows system will be prioritized over other solutions because, in the particular case of this work’s authors, it is used much more commonly in classes and by the students.

Other limitation that can also be addressed in future works is to decouple the device driver from the WiiMote, allowing it to serve as a low-level communication interface for any Bluetooth device. Perhaps even more important is to modify the system to work with more than one controller at a time. Adding the possibility of combining more than one InputDevice, to allow the creation of complex interfaces built with simpler modules, is also an interesting future work. Another possibility to study is extending this work to include other motion sensing controllers, such as Sony’s Sixaxis wireless controller.

Perhaps the most interesting possible future works with this system are using it to explore pattern recognition to build more complex gesture-based interfaces and putting the system to practical use, both using the system in classrooms and building games with innovative 3D interfaces (particularly gesture-based and TUIs). To this end, a game is currently being developed combining enJine, the WiiMote and Robot AReNa. While enJine is more of an educational tool in itself, instead of a tool for making educational games, using the WiiMote makes the concepts of accelerations and forces very tangible and building an application to teach those concepts is another interesting future work.

Finally, this work was more concerned with the necessary infrastructure for integrating WiiMote and enJine than with studying the controller’s use, but formally studying usability aspects of games using WiiMote is an interesting and necessary future work.

References


