Activate Your GAIM: A Toolkit for Input in Active Games

Matthew Brehmer, T.C. Nicholas Graham, Tadeusz Stach
School of Computing
Queen's University
brehmer@cs.ubc.ca, { graham, tstach}@cs.queensu.ca

ABSTRACT
Active games are video games that involve physical activity. Interaction in active games is captured via a variety of input devices such as accelerometers, cameras, pressure sensors and exercise equipment. It is difficult for programmers to deal with this profusion of devices, leading most active games to be tied to a particular hardware platform. In this paper, we introduce the GAIM toolkit. GAIM simplifies input handling in active games through a high-level API that abstracts the details of individual devices. This allows developers to write code independently of the input devices used, allows the toolkit to dynamically adapt to the devices a player has available, and allows people with different hardware to play together. We illustrate the approach through two active games developed using the GAIM toolkit.

Categories and Subject Descriptors
H.5.2 [User Interface]: Input devices and strategies, Interaction styles;

Keywords
Active video game, exergame, game development toolkit

1. INTRODUCTION
Active games, video games that involve physical activity, have become tremendously popular in recent years. Examples include Wii Tennis, where players swing an accelerometer to control a tennis racquet [18]; Dance Dance Revolution, where players perform dance steps to music [8], and Frozen Treasure Hunter, where players pedal a bicycle while carrying out quests in a virtual world [26]. The success of the Nintendo Wii, which has sold over 67 million units to-date [6], indicates the popularity of active games. Interest has been further increased by the announcement of new motion sensing technologies, such as Microsoft's Project Natal and Sony's Move.

Despite the importance of this new form of interaction, active games are difficult to program, and are poorly supported by toolkits. Most active games are designed for a specific hardware platform: Wii games are based on input from accelerometers and IR tracking; EyeToy games are designed around camera input [12], and PCGamerBike games are tied to pedal and steering input. This is analogous to the early days of graphical user interfaces, where programmers needed to write custom code to support each mouse or trackball that might be connected to the computer.

In this paper, we present the General Active Input Model (or GAIM) toolkit. GAIM simplifies the programming of active games by abstracting the details of active input devices. GAIM provides programmers with a set of *abstract input*s that capture the game-level intent of the input rather than the low-level inputs provided by the device. For example, an exercise bicycle or a heart rate monitor might provide a *power* input representing the player's degree of exertion; a Wii Remote might provide *gesture* and *point* inputs capturing the player's movements; and a depth camera or Wii Balance Board might provide *stance* inputs specifying the player's position.

The use of these abstract inputs carries three advantages over hand-coding. First, programmers are freed from the details of low-level input hardware, no longer having to deal with hidden Markov models for processing accelerometer data [23] or custom API's for exercise equipment. Second, games can run over a wide range of hardware without special coding. This allows games to automatically adapt to the hardware that the player actually has available, without special coding or recompilation for different devices. This approach also allows players with different hardware to play together, reducing the barriers to multiplayer exercise.

This paper is organized as follows. To motivate the need for device-independent input handling, we review the diversity of hardware used in active video games. We then summarize the GAIM classification of input for active games that is used as the basis of our toolkit. We then introduce the software framework itself, and show how it can be used to implement active games over a variety of hardware devices. Our examples show that GAIM-based active games can require less input-handling code than non-active versions of the same game.

2. INPUT HARDWARE
Recent years have seen the emergence of an excitingly diverse set of devices supporting active games. Here we summarize common approaches used to capture physical movement in active games.

Accelerometers measure changes in speed and rotation. For example, Nintendo's Wii Remote contains a 3-axis accelerometer that is used to detect gestures such as swings of a tennis racquet or golf club (figure 1A). Networks of accelerometers can be used to perform full pose detection [23]. Accelerometers are subject to cumulative error, however, and therefore are poorly suited for detecting absolute position. This problem can be to some degree addressed by augmenting the accelerometer with a *gyroscope*, as with Nintendo's Wii MotionPlus attachment for the Wii Remote.

Developing games using accelerometers requires the use of complex classification and recognition algorithms, typically based on hidden Markov models [7, 23]. Individual gestures must be trained.

Cameras permit the capture and analysis of images or video. Some active games use vision to determine a player's position and movement. For instance, vision-based input is used in Sony's EyeToy and PlayStation Move, Microsoft's Project Natal (figure 1C), and some academic games [11, 25]. Unlike accelerometer-

Figure 1: Example active input hardware. (A) Wii Remote, (B) Wii Balance Board, (C) Microsoft Project Natal camera, (D) Tunturi E6R ergometer, (E) PCGamerBike Mini

based approaches, vision often seeks to track the actual positions of objects, allowing for example the real-time tracking of a baseball bat, a sword or a light sabre.

Capturing human input with computer vision may require complex classifiers [17, 22]. Challenges include accurately identifying objects in varying lighting conditions, tracking occluded objects, and filtering complex backgrounds.

Accelerometer and vision-based input have the shortcoming that they provide no resistance to the player's movement. For example, in a tennis game, the player does not feel the weight of the tennis racquet, or feel the force of the ball impacting the racquet. *Ergometers* are exercise equipment that provide resistance, and allow real-time measurement of the physiological effects of physical activity. Ergometers are common in active games designed to promote physical activity ("exergames"). Examples include the Tunturi E6R, a recumbent exercise bicycle (figure 1D); the Fisher-Price Smart Cycle, a children's bicycle; the PCGamerBike Mini, a floor-mounted pedaling device compatible with many commercial games (figure 1E), and the Gamercize stepper products.

Ergometers are often used to determine the rate of travel of avatars in games, such as controlling a truck in Heart-Burn [21], a robot in Frozen Treasure Hunter [26], or the speed of a boat in Swan Boat [1].

Sometimes games require players to contact locations in the physical world. *Pressure sensors* and *touch sensors* capture contact with a surface, and may report the degree of force applied to the contact. These sensors are typically built into pads or mats. For instance, Dance Dance Revolution uses touch sensors to track players' dance steps; Remote Impact uses pressure sensors to measure the force and position of players' punches and kicks [14], and the Wii Balance Board contains pressure sensors to determine a player's center of gravity (see figure 1B).

This wide range of input hardware makes it difficult to program active games. Some hardware requires complex, low-level programming using hidden Markov or image processing algorithms. When attempting to detect similar inputs (e.g., the player's position or gestures), very different algorithms may be required depending on the input device. Even when the devices are similar, completely different API's may be required (e.g., for exercise bicycles such as the Tunturi E6R and the PCGamerBike.) By abstracting input from hardware in our GAIM toolkit, we allow active games to be developed independently of specific input devices. This is analogous to how the developer of a PC video game does not need to know whether the player is using a mouse, a touchpad or a trackball.

GAIM is based on a recent classification of active input [20]. To motivate this choice, we first explore existing input classifications and toolkits, and then present the abstract input model underlying our toolkit.

3. CLASSIFYING INPUT IN ACTIVE GAMES

There have been several approaches to classifying and formalizing input for classical desktop interfaces. Card et al. have classified the design space of traditional input devices [3]. Duke et al. have created a formalism for describing input interactions [5]. Toolkits such as Garnet [15] and the Universal Structured Model [4] allows for the development of interactive desktop applications independently of input devices.

The Reality-Based Interaction framework provides an understanding for interfaces beyond traditional desktop interaction (e.g., virtual, mixed, and augmented reality) [9]. While the framework's high-level descriptions of interactions help understand the range of interactions possible in modern interfaces, a finer-grained approach is required for the development of an active games toolkit.

Toolkits have been developed for handling physical and tangible interactions. The Exemplar tool was created for the development of sensor-based controls [7]. Although the tool links sensor input to application logic, the level of abstraction is not general enough to encapsulate active game interactions. The Papier-Mâché toolkit abstracts input for tangible objects in a ubiquitous environment [10], while the iStuff toolkit allows for the development of interactive ubiquitous systems [2]. Although both Papier-Mâché and iStuff treat input in an abstract manner, they are heavily focused on ubiquitous computing, and do not cover many of the input types common in active games.

3.1 Abstract Input Classification

Our own classification [20] identifies six styles of interaction commonly found in active games: gesture, stance, point, power, continuous control, and tap. These styles were identified through the examination of 107 academic and commercial active games. As we will see in section 4, the GAIM toolkit's API is based on these input styles, rather than on concrete input devices. We now detail these six abstract interaction styles.

A *gesture* is the movement of part or all of the body in a defined pattern. Gestures in active games typically represent commands and not real-time controls. For example, in Wii Bowling, when a player completes a throwing gesture using a Wii Remote, it is interpreted as a specific command (avatar releases ball). Gestures

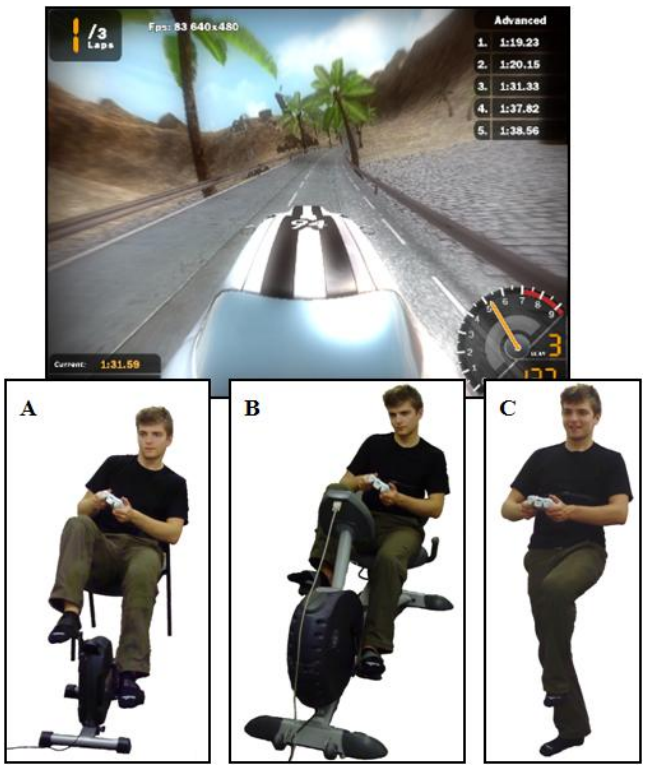

Figure 2: Racing Game. (A) PCGamerBike Mini input, (B) Tunturi E6R input, (C) Polar heart rate monitor input

Figure 3: Spacewar Game. (Bottom) Wii Remote and Balance Board input

may also include additional information, such as force and direction.

Stance represents the position of a person's body and limbs at a specific time. A stance describes the player's pose as opposed to an action the player may be performing. In contrast with gestures, which capture movements over time, a stance captures a player's position at a given instant. Stance input is the core of the Posemania dancing game [23], where players must position their body correctly in time in order to score points. Stance in Posemania is detected by a set of accelerometers attached to the player's body. Stance is also used in Namco Bandai Games' We Ski, where players lean left and right to determine their direction when skiing down a hill.

Pointing is used by players to reference in-game entities. Pointing captures a deictic reference, not a player's pose. Players typically point a finger or hand-held device at a region of the display. For instance, in Sega's House of the Dead series, a player points a light gun at the screen to aim her weapon.

Power is a measure of the intensity of the player's activity. Power input is captured continuously over a period of time. For example, in Swan Boat players use a treadmill to power an on-screen boat [1]; the player's running speed determines the speed of the boat. Conversely, the Heart Burn game uses heart rate as a measure of power to control the speed of a player's virtual truck on a race track [21].

Continuous control maps the player's body movement to the position of an on-screen entity. For instance, in Sega Superstars: Sonic, the EyeToy camera tracks a player's hand to determine the movement of an on-screen character. Similarly, in the Body-Driven Bomberman game, a player's position in physical space is mapped directly to the virtual position of her avatar [11]. Continuous control differs from gesturing in that it provides real-time mapping to the virtual object being controlled, whereas gestures invoke a command once they are complete.

A *tap* occurs when a player touches an object or location. For example, in Dance Dance Revolution, players dance in time to music by tapping locations on a floor mat with their feet. In Remote Impact, players spar with each other by punching a touch-sensitive mat [14].

The six input types presented above generalize the common interactions in active games. As described in section 4, these input types form the API of our GAIM toolkit.

3.2 Example Games

We use two example games to illustrate how this input classification naturally abstracts active input from the underlying input devices. We converted existing games to active gameplay using the GAIM toolkit; our experience in doing so is described in section 5.

3.2.1 Racing Game

This 3D racing game (based on Microsoft's XNA Racing Game) allows a player to race a car around a track. In our active version of the game, the car's speed is controlled by a *power* input, and its direction is controlled with the left analog stick of an Xbox 360 controller. Players can provide power to the game using a choice of stationary bicycles or by jogging on the spot (figure 2).

Power is derived either from the player's speed (pedal cadence and tension) or the player's heart rate (relative to their target heart rate.) In this example, inputs from four different devices (two

bicycles and two models of heart rate monitor) are abstracted to the single *power* input type.

3.2.2 Spacewar

Spacewar (based on the XNA Creator's Club Spacewar game) is a multiplayer 2D game inspired by the original Spacewar game of 1962. In the game, a player maneuvers her ship around obstacles while trying to shoot her opponent's ship. In our active version of Spacewar, a player's *stance* is used to steer her ship, and a "hammering" *gesture* is used to fire weapons (figure 3). Leaning left rotates the ship to the left, and leaning right rotates the ship to the right. Leaning forward accelerates, while leaning backwards slows the ship down. A player vigorously shakes her hand up and down to fire the ship's weapons. Stance is captured using a Nintendo Wii Balance Board, and gestures are captured with a Wii Remote. This example illustrates how a game's interaction may be composed from multiple active input types.

4. THE GAIM TOOLKIT

Based on the classification presented in section 3.1, the GAIM toolkit is a class library that abstracts the details of individual input devices. This provides developers with three key advantages. The toolkit:

- abstracts the details of peripherals used to capture active input, simplifying the programming task;
- allows active games to adapt to the hardware the player actually has, without special coding or recompilation;
- allows people using different devices to play together.

GAIM allows developers to program active games based on the six input types described in section 3.1. The toolkit provides a variety of implementations for each input type, allowing transparent plug-replacement of input devices without requiring modification to the program code. As with the XNA Racing Game example, the *power* input type can be used to determine the speed of a player's avatar. *Power* input might be provided by a variety of possible devices, such as a stationary bicycle, a treadmill, or jogging on the spot. The game's program code does not need to reference the underlying device. Because of this, a single game can be compatible with many input devices (as long as at least one of them provides the required input type), and in multiplayer games, players can interact with the game using different devices.

4.1 Abstract Details of Input Devices

Developers of active games currently must program using the API's of specific hardware input devices. For example, the active Racing Game (figure 2) can be played using a Tunturi E6R Ergometer, a PCGamerBike Mini, and a Polar heart rate monitor. To obtain information on the player's exertion level, these devices require the use of the Tunturi protocol, the FitXF protocol and the SparkFun protocol, respectively. Therefore, special purpose code must be written for each device that might be used with the game. This device dependence hinders the portability of games. For example, a game designed for Wii Remote and Balance Board input does not easily port to the PlayStation Eye.

Our approach instead uses the six abstract input types identified in section 3.1. The toolkit provides interfaces for four of the six input types: *IPower*, *IGesture*, *IPoint* and *IStance*. (Interfaces for the *tap* and *continuous control* input types will be added in the

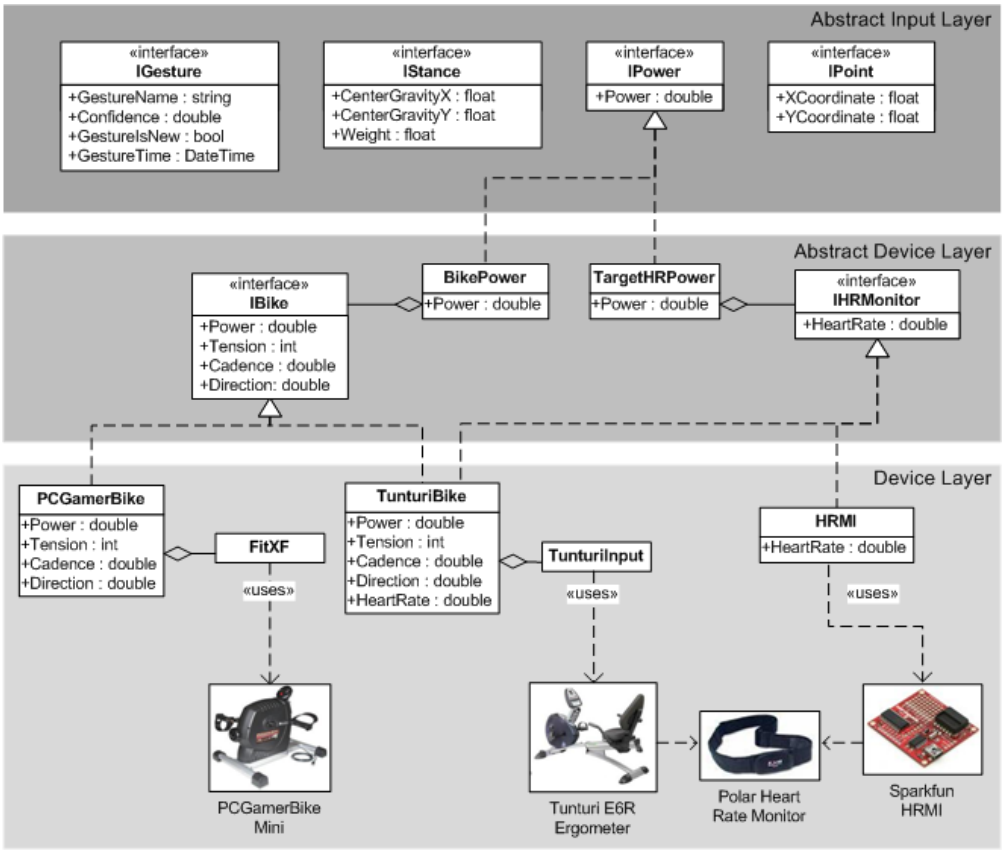

Figure 4: Design of the GAIM toolkit with expanded IPower interface

future.) If a programmer wishes to use a particular input type in a game, she instantiates the appropriate interface. For example, in the Spacewar game of figure 3, the direction of the spaceship is based on the player's stance. The programmer asks the toolkit for an implementation of the *IStance* interface. The toolkit selects the best device to provide stance input, and returns an *IStance* object tied to that device. The game queries this object for the current center of gravity values, and uses them to set the ships direction.

Each device is capable of providing one or more input types. For example, the E6R Ergometer provides *power* input, while the Wii Remote provides both *gesture* and *point* inputs.

The toolkit is divided into three layers. The *abstract input layer* is intended for use by application programmers, and provides access to the input types identified in our classification. The *abstract device layer* provides interfaces to broad classes of devices (e.g., bicycles, heart rate monitors, accelerometers), while abstracting their differences. The *device layer* provides access to concrete devices. Classes at this layer interact with application programmer interfaces provided by the device's manufacturer or with independently developed interfaces.

For example, figure 4 shows the classes making up GAIM's *IPower* interface. The interface provides a single property, *Power*, that reports the game player's current power output. The toolkit provides two alternative implementations of power – one based on stationary bicycles (*BikePower*), and the other based on heart rate (*TargetHRPower*). As described in section 3.2.1, heart rate input bases the player's power on how close she is to her target heart rate [20]. These classes rely on interfaces provided by the abstract input layer. The *IBike* interface provides attributes capturing the current power, tension, cadence and direction of the bicycle device. The *IHRMonitor* interface reports the player's current heart rate.

The device layer provides access to the equipment itself. Each concrete device implements one or more abstract inputs. The *PCGamerBike* class implements the *IBike* interface, while the *HRMI* class implements the *IHRMonitor* interface. The Tunturi E6R is a recumbent stationary bicycle supporting both cycling and heart rate monitoring, and therefore the *TunturiBike* class implements both interfaces.

When a game is played, one of the possible implementations of *IPower* will be chosen at runtime, based on preference and availability of hardware.

A challenge in designing these interfaces is that not all devices provide the same functionality. For example, as a full-featured exercise bicycle, the Tunturi E6R provides full control over tension and cadence, and reports power generated in Watts. As a less expensive gaming peripheral, the PCGamerBike Mini provides only cadence information. (Tension can be set manually, but cannot be read programmatically.) The PCGamerBike Mini cannot report true power values, since the tension value is required to compute it. The *PCGamerBike* class therefore estimates power from the current cadence and an average tension value. Additionally, tension can be set manually by an application program should it have better knowledge of the tension (e.g., via user input.)

4.2 Adapt to Player Hardware

The GAIM toolkit allows active games to adapt to the hardware that is available to the player. Some existing active games already do this in a limited form. For example, EA Sports Active's Tennis game allows players to swing their tennis racquet with a Wii Remote. If a Wii Balance Board is available, players may perform additional actions like lunging for the ball. This is a simple form of runtime adaptation, where extra actions are possible if supplemental hardware is available, but the game remains playable without it.

In games developed with GAIM, players may use radically different hardware as long as it implements the required abstract input. To determine which input devices are available, the toolkit uses a simple textual configuration file. The file lists each available input type and, if necessary, specifies the port over which it can be connected. For example, the following configuration file specifies three power sources – two BikePower sources, and one TargetHRPower source. All three specify the actual device that can be used to obtain the input. Additionally, the configuration file specifies that stance information can be obtained via *WiiStance*, as provided by the Wii Balance Board:

```
BikePower PCGamerBike
BikePower TunturiBike COM1
TargetHRPower HRMI COM3
WiiStance WiiStance
```

Within a game, the programmer uses the *DeviceManager* class to access devices implementing the desired input type. For example, to obtain power input, the programmer simply writes:

```
IPower powerDevice = DeviceManager.getIPower();
```

The current power value can then be referenced as `powerDevice.Power`.

The device manager chooses the first implementation of the *IPower* interface specified in the configuration file. In the above example, the PCGamerBike would be used to provide power input. Therefore, the toolkit is able to choose from a set of available devices, based on their ability to implement the desired abstract input interfaces. From the programmer's perspective, this choice is transparent.

Adding new device types to the toolkit requires programming. The device has to be added into the class hierarchy in the appropriate place(s), depending on the abstract input it is capable of implementing. The device manager needs to be modified to be capable of instantiating the new device. A significant benefit of this approach is that existing games can use new hardware as it becomes available without even requiring recompilation.

4.3 Support Hardware of Multiple Players

A major advantage of the GAIM approach is that it allows distributed play between people with different available hardware. For example, in the active Racing Game of figure 2, one player can use a PCGamerBike Mini while another uses a Polar heart rate monitor. This makes it easier for people to play together, even if they have chosen to purchase different active gaming peripherals.

A challenge in allowing people with different hardware to play together is that one person's hardware may be better suited to the game than their opponent's, providing an unfair advantage. For example, a player using a Wii MotionPlus enjoys more accurate gesture recognition than one with a standard Wii Remote. Similarly, the Tunturi E6R provides more comfortable and stable seating than the PCGamerBike Mini, but has higher latency in reporting changes of speed. This problem of differing quality is not unique to active games – in a first-person shooter, a player using a laptop touchpad is significantly disadvantaged versus an

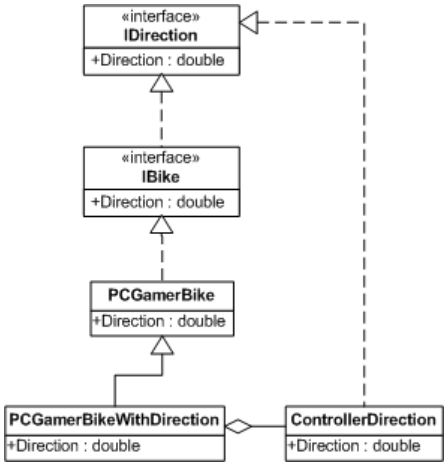

Figure 5: Class diagram of *IDirection* for supporting mixed active *power* input

opponent using a high-end gaming mouse. One could imagine techniques for balancing these inequalities, such as adding latency to the PCGamerBike Mini, or increasing the error rate of the Wii MotionPlus. However, in GAIM, we follow the approach of standard gaming of allowing each device to operate as best it can, and accepting that this may lead to inequalities.

Devices may differ not just in the quality with which they implement interaction but also in basic features. For example, the Tunturi E6R supports forward pedaling only, and has variable resistance under programmatic control. The PCGamerBike Mini supports forward and backward pedaling, but provides no programmatic control of resistance. One way of handling these differences would be to take the lowest common denominator of all possible devices, and support only those functions (e.g., forward pedaling only and no resistance control.) However, this approach burdens players with the shortcomings of all devices, even those not in use. In GAIM, we address this problem using default values, availability properties, and functionality mix-ins, described below.

In the abstract input layer, *default values* are provided for properties that are not available on the device. For example, the Tunturi E6R implementation of *IBike* (see figure 4) always has the value of 0 radians for direction, while the PCGamerBike Mini may have values of 0 or π radians, depending on whether the player is pedaling forwards or backwards. Similarly, the PCGamerBike Mini always reports a tension of 25 Watts, even though the tension is in fact unknown. These default values allow games to operate with lower-capability hardware, albeit with less functionality. Programmers may choose to add special cases to their code depending on whether the hardware supports a particular function. To permit this, the abstract input classes provide *availability properties* to indicate which functions are actually supported. For example, the IBike interface provides a boolean *HasPedalTension* property that specifies whether pedal tension is correctly reported, or estimated via a default value.

Also at the input layer, *mix-in classes* can be deployed to add missing functionality. For example, figure 5 shows how the PCGamerBike's bidirectionality can be extended to general direction. The *IDirection* abstract input reports a direction; PCGamerBike implements this (giving one of two directions), as does the Controller class (giving arbitrary direction, specified by the right stick of an Xbox 360 controller.) A new *PCGamerBikeDirection* class mixes these two functionalities to give a steerable PCGamerBike Mini, where both pedaling and turning direction are taken into account. Specifically, pedaling forward with the stick to the right moves forward and right, while pedaling backwards with the stick to the right moves backward and left. The use of such mix-ins allows traditional devices to be used to augment active input devices, compensating for missing functionality in an active input device.

5. EXPERIENCE

To evaluate the effectiveness of the toolkit, we created the two active games described in section 3.2. Both games were derived from existing games built for keyboard/mouse or game controller play. We chose to modify existing games so that we could compare the code required to create a traditional game versus a device-independent active game.

The active Racing Game was based on Microsoft's XNA Racing Game (available at www.xnaracinggame.com). The game's functionality is unchanged, other than that players now control the speed of their car using a *power* input.

To modify the game, we removed 35 lines of code taking input from the mouse/keyboard or game controller, and inserted 11 lines of code to process power input. Since the game uses the *IPower* interface, no code changes are required to change from one device to another. As described in section 4.2, a simple text file is used to specify which devices are available to the application, allowing the toolkit to determine which class to use to implement *IPower*.

This example illustrates the practicality of basing input on high-level input types such as those described in this paper. Not only does the approach provide device independence, allowing radically different input devices to control the same game, but it (at least in this case) requires less code to process active input than was required to use traditional input devices.

To illustrate support for multiple input devices, we modified the XNA Creators' Club's Spacewar Game (available at creators.xna.com). In our active version of the game, the player's stance is used to steer the ship, and a "hammering" gesture is used to fire weapons. Stance is captured using a Nintendo Wii Balance Board, and gestures are captured with a Wii Remote. Figure 2 shows the modified Spacewar game using the stance and gesture active input techniques.

The Spacewar game requires 425 lines of code to implement traditional input. The Active Spacewar game required 32 lines to support input based on *stance* and *gesture*. The "hammering" gesture is included with the GAIM toolkit; if a new gesture had been used, time would have been required to train it, but no additional code would have been necessary.

The use of stance and gesture shows that it is possible to combine multiple active input types in a single game. Other combinations are also possible; for example, a game could include power and gesture inputs, or point and tap inputs.

Over all, our experience shows that it is practical to develop active games using the GAIM toolkit. In the case of these two games, the code required for device-independent active input is in fact less than the code required for traditional mouse, keyboard and gamepad input.

6. DISCUSSION

GAIM provides developers with platform independence and ease of development. As with all toolkits, these benefits come at the cost of low-level control of input devices. The GAIM toolkit allows programmers choice over this tradeoff between high-level programming and low-level control. The layered approach allows developers to access input at the level of abstract inputs (where the device is hidden), abstract devices (where the class of device is known, but the actual device is hidden), and concrete devices. For maximum portability, the abstract input layer should be used. However, if a programmer requires direct access to a particular hardware device, it is possible to obtain it.

An open question is the degree to which GAIM will prove to be extensible to new input devices as they come on the market. The input classification on which GAIM is based has been used to describe over 100 active games on a variety of platforms [20], lending confidence to its generality. Upcoming devices (particularly Microsoft Natal and PlayStation Move) appear to provide gesture, continuous control and stance input. As such, they provide similar inputs to existing technologies (EyeToy, Wii Remote, Wii Balance Board), but with significantly improved accuracy. Ultimately, the toolkit's extensibility will be determined by its ability to accommodate new input technologies as they become available.

Further work is required to address the problem that different players may have different experiences depending on the concrete hardware they possess. The mix-in class approach described earlier is a promising way of using standard input devices to augment the capabilities of active input devices. Game designers could also consider offering handicaps to players with less capable hardware. The layered architecture of the toolkit enables both approaches

In designing the GAIM toolkit, we explicitly excluded pervasive games [13], in favour of games that could be played in the living room. This is because, to-date, most active input devices have been attached to consoles or personal computers. The advent of portable devices with global positioning systems and accelerometers (such as the iPhone) has made the development of ubiquitous games practical. Ubiquitous games involve a wide range of sensors for detecting players' position and orientation, based on devices such as GPS, gyroscopes, accelerometers, WiFi triangulation and RFID. It would be therefore be a worthwhile extension of GAIM to include the input devices necessary for ubiquitous gaming.

Another interesting area opened by the GAIM toolkit is the possibility of developing active games for differently abled users. Input mechanisms providing the six input types could be custom-built for players with specific physical limitations. For example, the GameWheels [16] and GameCycle [24] input devices both provide *power* input.

7. CONCLUSION

In this paper, we have introduced a novel toolkit for dealing with input in active games. The primary goal of the toolkit is to help programmers deal with the wide variety of active input devices. The toolkit is based on an abstract input classification [20], which identifies the six input types of power, gesture, point, tap, continuous control and stance. We have designed the toolkit's API around these six input types, abstracting the details of a broad range of devices, such as accelerometers, heart-rate monitors, ergometers and pressure sensors. This approach allows applications to be developed independently of the details of the input devices used, allows applications to automatically adapt to the hardware a player has available, and allows players with different hardware to play together.

We have illustrated that the key to the toolkit's success is its use of an open, layered architecture. Applications can be fully device-independent using the abstract input layer. If knowledge of a particular class of device is required, the abstract device layer can be used. Finally, access to specific devices is possible through the concrete device layer. We have illustrated the use of the toolkit through the conversion of two games from traditional input to active input, and shown that in both cases, less code was required to implement the active version.

Our future plans include extending the toolkit to support more devices, and gaining further experience in its application.

8. ACKNOWLEDGEMENTS

The implementation of the GAIM toolkit owes significantly to several class libraries implementing interfaces to underlying devices. Will Roberts' *TunturiBike* class implements the Tunturi protocol, allowing us to interface with an E6R recumbent bike via a COM port. The *WiimotePoint* and *WiiStance* classes make use of Brian Peek's WiimoteLibrary to support input from Wii controllers (www.wiimotelib.org). The WiiGLE library [19] is used in our *WiiGesture* class, and our gestures sets were trained with the WiiGLE GUI application. We thank the authors of all of these libraries.

This work was partially supported by the NSERC Strategic Project "Technology for Rich Group Interaction in Networked Games".

9. REFERENCES

[1] Ahn, M., Kwon, S., Park, B., Cho, K., Choe, S. P., Hwang, I., Jang, H., Park, J., Rhee, Y., and Song, J. Running or gaming. In *Proc. ACE* 2009, 422. 345-348.

[2] Ballagas, R., Ringel, M., Stone, M., and Borchers, J. iStuff: a physical user interface toolkit for ubiquitous computing environments. In *Proc. CHI* 2003, 537-544

[3] Card, S. K., Mackinlay, J. D., and Robertson, G. G. The design space of input devices. In *Proc. CHI* 1990, 117-124.

[4] Dewan, P. Towards a Universal Toolkit Model for Structures. In *Proc. EIS* 2007, 393-412.

[5] Duke, D., Faconti, G., Harrison, M., and Paternó, F. Unifying views of interactors. In *Proc. AVI* 1994, 143-152.

[6] Fletcher, J.C., DS sells 125 million worldwide, Wii up to 67 million, *Joystiq*, Jan. 28, 2010.

[7] Hartmann, B., Abdulla, L., Mittal, M., and Klemmer, S. R. Authoring sensor-based interactions by demonstration with direct manipulation and pattern recognition. In Proc. *CHI* 2007, 145-154.

[8] Hoysniemi, J. International survey on the Dance Dance Revolution game. *Comput. Entertain*. 4(2), 2006, 8.

[9] Jacob, R. J., Girouard, A., Hirshfield, L. M., Horn, M. S., Shaer, O., Solovey, E. T., and Zigelbaum, J. Reality-based interaction: a framework for post-WIMP interfaces. In *Proc. CHI* 2008, 201-210.

[10] Klemmer, S. R., Li, J., Lin, J., and Landay, J. A. Papier-Mache: toolkit support for tangible input. In *Proc. CHI* 2004, 399-406

[11] Laakso, S., and Laakso, M. Design of a body-driven multiplayer game system. In *Comput. Entertain.*, 4(4), 2006, 7.

[12] Larssen, A., Loke, L., Robertson, T., and Edwards, J. Understanding movement as input for interaction—a study of two eyetoy™ games. In *Proc. OzCHI* 2004.

[13] Magerkurth, C., Cheok, A. D., Mandryk, R. L., and Nilsen, T. Pervasive games: bringing computer entertainment back to the real world. *Comput. Entertain*. 3(3), 2005, 4.

[14] Mueller, F.F., Agamanolis, S., Vetere, F., Gibbs, M.R. Remote Impact: shadowboxing over a distance. In *Proc. CHI* 2008, 2291-2296.

[15] Myers, B. A. A new model for handling input. *ACM Trans. Inf. Syst.* 8(3), 1990, 289-320.

[16] O'Connor, T.J., Fitzgerald, S.G., Cooper, R.A., Thorman, T.A., and Boninger, M.L. Does computer gameplay aid in motivation of exercise and increase metabolic activity during wheelchair ergometry? *Medical Engineering and Physics*, 23(4), 267-273, 2001.

[17] Oliver, N., Rosario, B., and Pentland, A. A Bayesian computer vision system for modeling human interaction. In *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(8), 831-843, 2000.

[18] Parker, J. R. Games for physical activity: A preliminary examination of the Nintendo Wii. In *Proc. 6th International Symposium on Computer Science in Sport*, 2007.

[19] Rehm, M., Bee, N., and André, E. 2008. Wave like an Egyptian: accelerometer based gesture recognition for culture specific interactions. In *Proc. British HCI* 2008, 13-22.

[20] Stach, T., Graham, T.C.N., Brehmer, M., and Hollatz, A. Classifying input for active games. In *Proc. ACE* 2009, 422, 379-382.

[21] Stach, T., Graham, T.C.N., Yim, J., and Rhodes, R. Heart rate control of exercise video games. In *Proc. GI* 2009, 125-132.

[22] Moeslund, T.B., and Granum, E. A survey of computer vision-based human motion capture. In *Computer Vision and Image Understanding*, 81(3), 231-268, 2001.

[23] Whitehead, V, Johnston, H., Crampton, N., and Fox, K. Sensor networks as video game input devices. In *Proc. of Future Play* 2007, 38-45.

[24] Widman, L.M., McDonald, C.M,, and Abresch, R.T. Effectiveness of an upper extremity exercise device integrated with computer gaming for aerobic training in adolescents with spinal cord dysfunction. *Journal of Spinal Cord Medicine*, 29(4), 363-370, 2006.

[25] Yim, J., Qiu, E., and Graham, T.C.N. Experience in the design and development of a game based on head-tracking input. In *Proc. Future Play 2008*, 236-239.

[26] Yim, J., and Graham, T.C.N. Using games to increase exercise motivation. In *Proc. Future Play* 2007, 166-173.