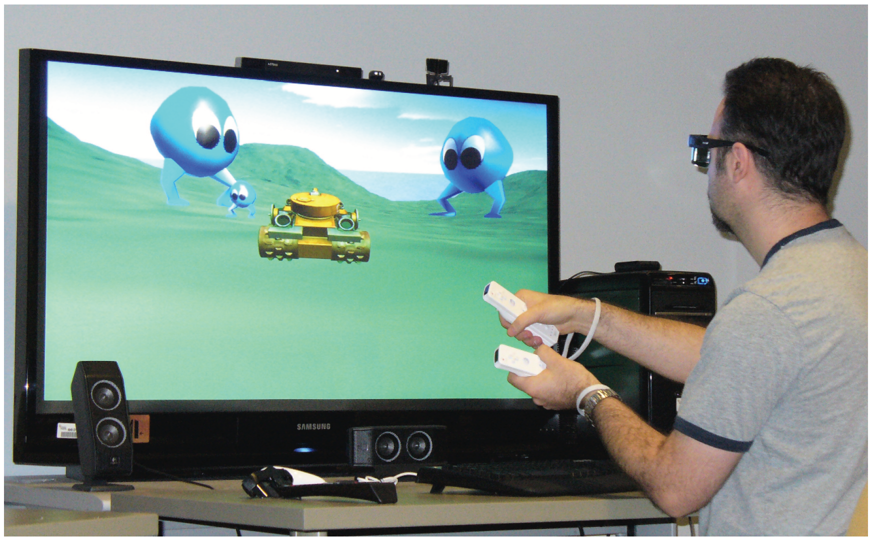
An Introduction to 3D Spatial Interaction with Video Game Motion Controllers

A SIGGRAPH 2010 Course
Tuesday, July, 27, 2010
2:00PM - 5:15 PM

Joseph J. LaViola Jr.
School of EECS
University of Central Florida
jjl@eecs.ucf.edu

Richard L. Marks
Sony Computer Entertainment US R&D
Sony Computer Entertainment America
Richard_Marks@Playstation.sony.com



Contents

1 Course Introduction

3D spatial interfaces [Bowman et al. 2004] give users the ability to spatially interact with 3D virtual worlds because they provide natural mappings from human movement to interface controls. These interfaces, common in virtual and augmented reality applications, give users, rich, immersive, and interactive experiences that can mimic the real world or provide magical, larger than life interaction metaphors [Katzourin et al. 2006].

3D interfaces utilize motion sensing, physical input, and spatial interaction techniques to effectively control highly dynamic virtual content. With the advent of the Nintendo Wii, EyeToy, and a host of soon-to-be-released peripherals such as the PlayStation Move motion controller, Microsoft's Natal, and Sixense's TrueMotion, game developers, researchers and hobbyists are faced with the challenge of coming up with compelling interface techniques and gameplay mechanics that make use of this technology [Wingrave et al. 2010]. Researchers in the fields of virtual and augmented reality as well as 3D user interfaces have been working on 3D interaction for nearly two decades. The techniques and interaction styles and metaphors developed over this time in these communities are directly applicable to games that make use of motion controller hardware.

1.1 Course Topics

In this course, presenters will demystify the workings of modern day video game motion controllers and provide a thorough overview of the techniques, strategies, and algorithms used in creating 3D interfaces for tasks such as 2D and 3D navigation, object selection and manipulation, gesture-based application control, and character control. We will discuss the strengths and limitations of various motion controller sensing technologies, found in today's and soon-to-be-released peripherals including accelerometers, gyroscopes, and 2D and 3D depth cameras. We also will present techniques for compensating for their deficiencies including gesture recognition and non-isomorphic control-to-display mappings. Attendees will receive valuable information on how to apply existing 3D user interface techniques using commercial motion controllers as well as develop their own techniques.

1.2 Course Schedule

1. Welcome, Introduction,& Roadmap – 20 minutes - LaViola
   (a) Motivate the importance of the topic
   (b) Introduce presenters
   (c) Game UI - Past, Present and Future
2. Common Tasks in 3D User Interfaces – 30 minutes - LaViola
   (a) Selection and manipulation techniques
   (b) Travel techniques
   (c) System control techniques
3. 3D Interfaces with 2D and 3D Cameras – 45 minutes - Marks
   (a) EyeToy
      i. Motion detection
      ii. Color tracking
   (b) PlayStation Eye
      i. Marker tracking
      ii. Face tracking
   (c) 3D cameras
      i. Technology overview
      ii. Foreground/background separation
      iii. Augmented Reality
      iv. Real-time motion capture
4. Working with the Nintendo Wiimote – 40 minutes - LaViola
   (a) Device characteristics
   (b) Device mathematics
   (c) Home-brew Wiimote software
   (d) Heuristic motion analysis
5. 3D Spatial Interaction with the PlayStation Move – 45 minutes - Marks
   (a) PlayStation Move characteristics
   (b) PlayStation Move technology
   (c) Spatial input vs. buttons

    (d) 1-to-1 data mapping vs. other mappings

6. 3D Gesture Recognition Techniques – 30 minutes - LaViola
    (a) 3D Gestures
    (b) Linear and AdaBoost classifiers
    (c) A 25 gesture open source dataset

1.3 Speaker Biographies

Joseph J. LaViola Jr.
Assistant Professor, University of Central Florida

University of Central Florida
School of EECS
4000 Central Florida Blvd.
Orlando, FL 32816

(407)882-2285 (voice)
(401)823-5419 (fax)
jjl@eecs.ucf.edu (email)

Joseph J. LaViola Jr. is an assistant professor in the School of Electrical Engineering and Computer Science and directs the Interactive Systems and User Experience Lab at the University of Central Florida. He is also an adjunct assistant research professor in the Computer Science Department at Brown University. His primary research interests include pen-based interactive computing, 3D spatial interfaces for video games, predictive motion tracking, multimodal interaction in virtual environments, and user interface evaluation. His work has appeared in journals such as ACM TOCHI, IEEE PAMI, Presence, and IEEE Computer Graphics & Applications, and he has presented research at conferences including ACM SIGGRAPH, ACM CHI, the ACM Symposium on Interactive 3D Graphics, IEEE Virtual Reality, and Eurographics Virtual Environments. He has also co-authored "3D User Interfaces: Theory and Practice," the first comprehensive book on 3D user interfaces. In 2009, he won an NSF Career Award to conduct research on mathematical sketching. Joseph received a Sc.M. in Computer Science in 2000, a Sc.M. in Applied Mathematics in 2001, and a Ph.D. in Computer Science in 2005 from Brown University.

Richard L. Marks
Senior Researcher
Sony Computer Entertainment US R&D

Sony Computer Entertainment America
US R&D Division
919 E. Hillsdale Blvd.
Foster City, CA 94404

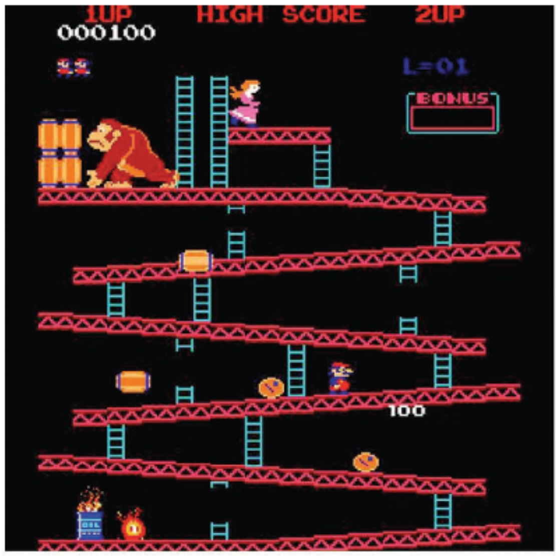
(650)655-8000 (voice)
(650)655-8060 (fax)
richard_marks@playstation.sony.com (email)

Richard Marks is currently a Senior Researcher in Sony's US R&D group, investigating new interactive user experiences. Inspired in 1999 by the unveiling of PlayStation 2, he joined PlayStation R&D to investigate the use of live video input for gaming. He developed the technology for the EyeToy camera and worked closely with the Sony London studio to make it a successful product selling over 10 million units.

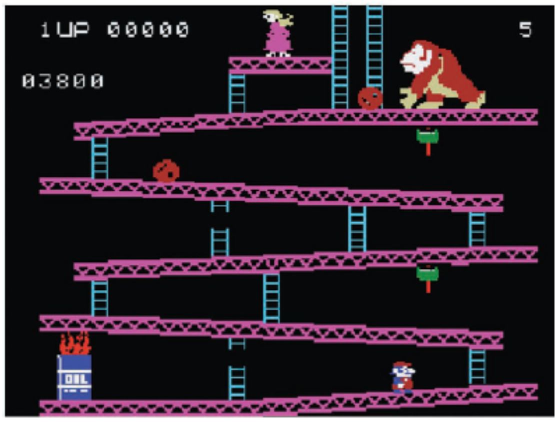
Richard later helped create the PlayStation Eye camera and a computer vision SDK for the PlayStation 3. Concurrently, he explored the use of 3D cameras and the new experiences they could enable. Most recently, he has been working on Sony's new motion controller, which combines both visual and inertial sensing to enable 3D interaction. Richard received a B.S. in Avionics from MIT in 1990, then received his Ph.D. in 1996 from Stanford University in the field of underwater robotics.

1.4 Game UI - Past, Present, and Future

The video game industry began in the mid-'70s. During these early days, if you wanted to play video games, you had two choices: go to an arcade or buy a game console such as the Magnavox Odyssey, which debuted in 1972. (PCs became popular in the 1980s but were expensive compared to game consoles.) Arcades were much more popular than game consoles because you could pack more graphics, sound, and game play into a large, decorated, wooden cabinet than you could into a relatively small console. This trend continued through the '80s as arcade game graphics and sound became more sophisticated. However, game consoles weren't far behind, as systems such as ColecoVision, the Sega Master System, and the Super Nintendo Entertainment System emerged. An example of how game consoles were catching up to the arcade is Donkey Kong on the ColecoVision, which was very close to the arcade version (see Figure 1).



(a)



(b)

Figure 1: *Donkey Kong: (a) the arcade and (b) ColecoVision game console versions. Game console graphics quickly approached the quality of arcade games.*

In the '90s, things changed. Consoles became faster, with better graphics and sound than arcade games. If you bought one of these consoles (or your parents bought one for you), you didn't have to put a quarter into a machine every time you wanted to play the coolest game. In addition, you could play for as long as you wanted and didn't have to travel anywhere to do so. So, the standard video arcade was able to move into the home. It became clear that arcade games couldn't compete with game consoles and PCs, in terms of graphics, sound, and length of play per gaming session. To compete with game consoles, the only thing video arcades could do was innovate at the user interface. This innovation came in the form of a variety of input devices and strategies that got players more actively involved.

During this period, arcade games introduced several particularly innovative user interface designs. For example, in BeachHead (see Figure 2a), the user wore a helmet-like device, resulting in a 360-degree field of regard. In games such as Football Power (see Figure 2b), players actually controlled a soccer ball with their feet. In Aliens Extermination (not shown), players used realistic gun props to interact in a first-person-shooter-style game. In Manx TT (see Figure 2c), users rode a physical motorcycle to control a virtual motorcycle. In Dance Dance Revolution (see Figure 2d), users interacted on a small dance floor. Adopting such interfaces for game consoles wasn't cost effective, so the video arcade was able to live a little while longer.

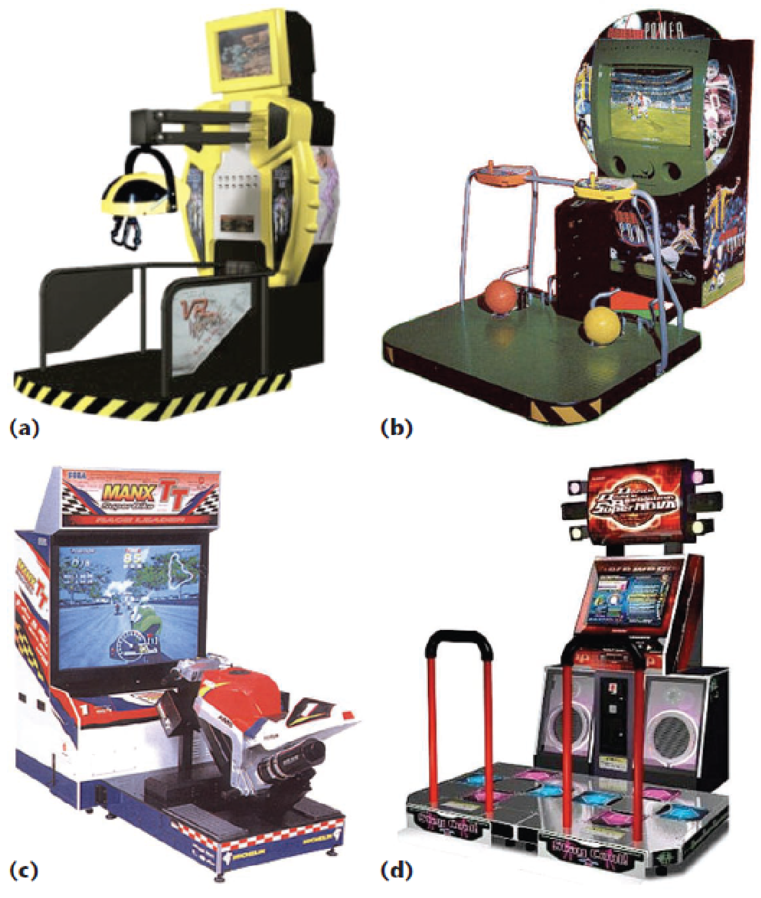


Figure 2: *Arcade games that used more realistic input strategies: (a) BeachHead, (b) Football Power, (c) Manx TT, and (d) Dance Dance Revolution. Such innovative interfaces helped arcade games compete with PCs and game consoles.*

During the '90s, VR technology appeared in video games. VR-based games helped keep arcades alive because they included more advanced interfaces employing such technologies as stereoscopic vision, head and body tracking, and 3D spatial interaction. These games appeared mainly in arcades and entertainment centers.

One of the first VR games was Dactyl Nightmare, which W Industries/Virtuality developed in the early ’90s as part of a suite of VR games. Players entered a pod-like structure, put on a head-mounted display, and used a tracked joystick to interact with the virtual world. The pod protected users from walking around in the physical world. The games themselves were somewhat primitive, but the immersive experience hadn’t been seen before in a video arcade game. One VR entertainment center was BattleTech, based on the BattleTech universe. The first one opened in Chicago in 1990. It aimed to provide an immersive experience where several users could play simultaneously not only at a single location but also networked across all BattleTech centers.

Although these types of video games provided more interesting interactive experiences than consoles, the cost of upkeep, lack of throughput, cost of play per gaming session, and continued improvement of game consoles led to the demise of most arcades. Arcades still exist, but they’re nowhere near as popular as they were in the late ’70s and ’80s. Today they’re often coupled with family entertainment centers, bars, bowling alleys, and other small venues. However, the video arcade game’s evolution shows that as time passed, these games had to provide users with higher levels of interaction-not just using a control stick and a set of buttons-to keep them interested and coming back for more.

1.4.1 Input Control and Game Complexity

If we examine the evolution of game consoles from the ’70s to today, we can see profound improvements in computer graphics, sound, artificial intelligence, and storytelling. However, a closer look shows that these games have become more complex, in terms of not only story, graphics, sounds, and so on but also gameplay. As video games became more complex by giving users a greater variety of things to do and controls to master, the input devices for controlling them stayed relatively constant. If we look at PC games, the argument is clear: mouse and keyboard have been the predominant control devices since PC gaming started in the early ’80s.

One could argue that console game input devices have improved and gotten more useful since the Atari 2600’s simple directional joystick and button. Game controllers certainly have gotten more complex, but not necessarily better. As new generations of game consoles emerged, the controllers simply added more buttons and joysticks to previous versions.

For example, the 1983 Nintendo Famicom used a game pad with direction buttons (to replace the directional joystick) and four buttons; the Sega Master System had a similar design. In 1994, when Nintendo introduced the first 3D game console (Nintendo 64), the controller had a directional joystick, a directional pad, and 10 buttons. Finally, the Sony PlayStation introduced the DualShock controller, which has two analog joysticks, a directional pad, and 10 buttons.

As controller complexity increased, so did the complexity of a game’s control scheme. These schemes allowed for more expression, at the cost of making the games difficult to learn and master. So, the interfaces became tailored to hard-core gamers, often alienating the casual player. For example, each new version of John Madden Football has gotten more realistic and improved how you control the football players. However, the game now has so many choices and controls to master that it is difficult to remember them all, effectively limiting what can be done during gameplay. It’s clear that game interfaces must become easier to use while maintaining the high levels of expression and control of modern console and PC games.

1.4.2 3D UI in Games Today

Glancing at video game history shows three trends that lead us to why video games are moving toward using modern video game controllers. First, once game consoles had better graphics and sound, had more interesting stories, and let users play much longer, arcade games had to give players something that they couldn't get on consoles: innovative interfaces that provided more natural means of expression. So, to compete in a market where consumers can choose from several gaming platforms, better graphics was no longer a key to staying on top. More natural gameplay, as we've seen with the Nintendo Wii's success, keeps people wanting more.

Second, more complicated video games and video game controllers gave users more expressive power but alienated casual gamers. In many cases, these games use somewhat abstract control schemes, when we consider the mappings between control mechanisms that are easy to perform naturally and spatially (running, jumping, punching, kicking, and so on) and a series of button presses. To bring back the casual gamer and improve overall gameplay, 3D spatial interaction is the natural next step.

Third, the technology to make 3D spatial interaction mainstream and not just a gimmick has arrived. Console games had previously incorporated advanced game interfaces (in the late '80s and early '90s) with devices such as the Nintento UForce, Mattel PowerGlove, and Sega 3D glasses. However, poor technology and lack of support from game developers caused their early demise. Today, faster and cheaper sensors, faster processors that can perform complex tracking and recognition, and the need to reduce game control complexity have finally made 3D spatial interaction feasible.

Current video games employ 3D spatial interaction in three ways. First, as the EyeToy showed, simple vision-based tracking can let players use their bodies to control game characters. Second, realistic 3D spatial interfaces based on active physical props-specifically, guitars and drum sets-give gamers the ability to interact as if they were in a real rock band. Guitar Hero and Rock Band are interesting examples of this type of realistic control scheme because people are willing to buy these devices for use with just one game. Ten years ago, no one would have believed that people would spend $90 to $200 on a single game with a specific controller. (This supports the possibility of more arcade-style interfaces from the '90s making their way into the home.) Finally, and probably the most important, is Nintendo's approach with its Wii and Wiimote. The Wiimote is one of the most significant technological innovations in 3D spatial interaction for gaming. It not only acts as a gamepad but also makes games accessible to the casual gamer because it can sense 3D motion. It showed that consumers were willing to accept this type of interface device and, as a result, we see that 3D motion sensing interfaces are going to be a part of every gaming platform.

1.4.3 Video Games and Modern Controllers

Given the plethora of new 3D motion sensing controllers, it is important to have a solid understanding of their strengths and weaknesses and how they can be effectively used to create video game interfaces. Work on this understanding has already begun. For example, in 2006, SwordPlay was developed (see Figure 3), a video game prototype in which users fight enemies using a sword and shield, a bow and arrow, and a set of spells they can sketch with the sword. This work aimed to leverage existing 3DUI techniques in the context of a video game. The game was played in a four-sided stereoscopic display (Cave Automatic Virtual Environment) and with 6-DOF trackers. Unfortunately, most people probably don't own a four-sided CAVE or an expensive 6-DOF tracking system. The project was successful in that it showed what might be possible in the future. Well, the future is now and the remaining sections of these course notes

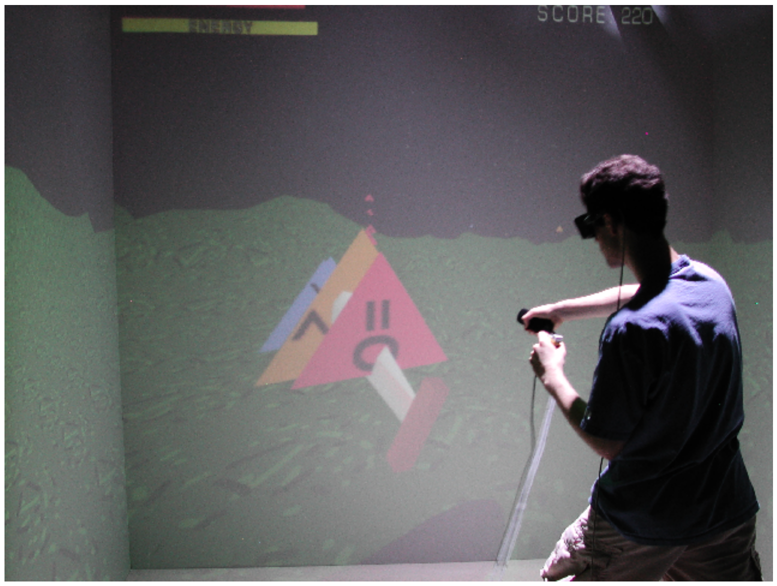


Figure 3: *A screen shot of SwordPlay, a video game prototype used to explore how 3D UIs can fit in a gaming environment.*

will provide useful information on how to work with the latest video game motion controller technologies.

2 Common Tasks in 3D User Interfaces

Before examining modern day video game motion controllers, we first will take a brief tour of the techniques used for common tasks in 3D spatial interaction. What is 3D spatial interaction anyway? As starting point, we can say that a 3D user interface (3D spatial interaction) is a UI that involves human computer interaction where the user's tasks are carried out in a 3D spatial context with 3D input devices or 2D input devices with direct mappings to 3D. In other words, 3D UIs involve input devices and interaction techniques for effectively controlling highly dynamic 3D computer-generated content, and there's no exception when it comes to video games. Figure 4 shows example 3D UIs.

A 3D video game world can use one of three basic approaches to interaction. The first maps 2D input and button devices, such as the keyboard and mouse, joysticks, and game controllers to game elements in the 3D world. This is the traditional approach; it's how people have been interacting with 3D (and 2D) video games since their inception many years ago. The second approach simulates the real world using replicas of existing devices or physical props. Common examples include steering wheels, light guns, and musical instruments (for example, the guitar in Guitar Hero). These devices don't necessarily provide 3D interaction in the game but do provide 3D input devices that enable more realistic gameplay. The third approach is true spatial 3D tracking of the user's motion and gestures, where users interact in and control elements of the 3D gaming world with their bodies. This control can come through vision-based devices such as the PlayStation Eye and motion-sensing devices such as Nintendo Wii Remotes (Wiimotes) and the PlayStation Move motion controller.

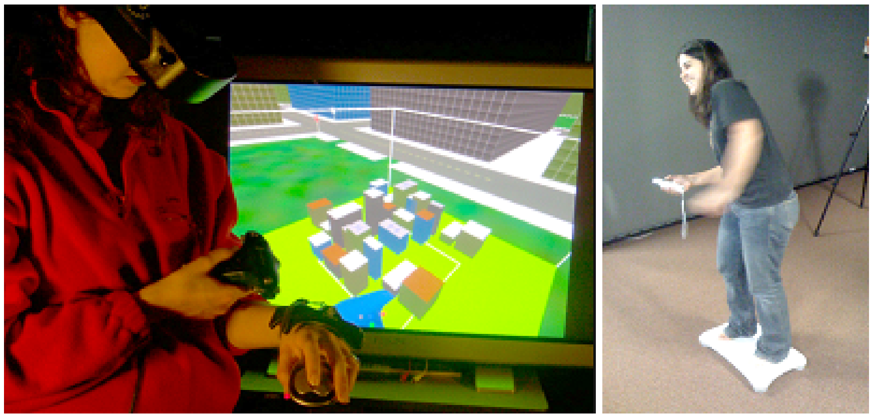


Figure 4: *Two examples of a 3D UI. The image on the left shows a user interacting with a world-in-miniature and the image on the right shows a user navigating using body-based controls.*

There are essentially four basic 3D interaction tasks that are found in most complex 3D applications [Bowman et al. 2004]. Actually, there is a fifth task called symbolic input, the ability to enter alphanumeric characters in a 3D environment, but we will not discuss it here. Obviously, there are other tasks which are specific to an application domain, but these basic building blocks can often be combined to let users perform more complex tasks. These tasks include navigation, selection, manipulation, and system control. **Navigation** is the most common VE task, and is consists of two components. *Travel* is the motor component of navigation, and just refers to physical movement from place to place. *Wayfinding* is the cognitive or decision-making component of navigation, and it asks the questions, ”where am I?”, ”where do I want to go?”, ”how do I get there?”, and so on. **Selection** is simply the specification of an object or a set of objects for some purpose. **Manipulation** refers to the specification of object properties (most often position and orientation, but also other attributes). Selection and manipulation are often used together, but selection may be a stand-alone task. For example, the user may select an object in order to apply a command such as ”delete” to that object. System control is the task of changing the system state or the mode of interaction. This is usually done with some type of command to the system (either explicit or implicit). Examples in 2D systems include menus and command-line interfaces. It is often the case that a system control technique is composed of the other three tasks (e.g. a menu command involves selection), but it’s also useful to consider it separately since special techniques have been developed for it and it is quite common.

There are two contrasting themes that are common when thinking about 3D spatial interfaces; the real and the magical. The real theme or style tries to bring real world interaction into the 3D environment. Thus, the goal is to try to mimic physical world interactions in the virtual world. Examples include direct manipulation interfaces, such as swinging a golf club or baseball bat or using the hand to pick up virtual objects. The magical theme or style goes beyond the real world into the realm of fantasy and science fiction. Magical techniques are only limited by the imagination and examples include spell casting, flying, and moving virtual objects with levitation.

Two technical approaches used in the creation of both real and magical 3D spatial interaction techniques are referred to as isomorphism and non-isomorphism. Isomorphism refers to a one to one mapping between the motion controller and the corresponding object in the virtual word. For example, if the motion controller moves 1.5 feet along the x axis, a virtual object moves the same distance in the virtual world. On the other hand, non-isomorphism refers to ability to scale the input so that the control-to-display ratio is not equal to one. For example, if the motion controller is rotated 30 degrees about the y axis, the virtual object may rotate 60 degrees about the y axis. Non-isomorphism is a very powerful approach to 3D spatial interaction because it lends itself to magical interfaces and can potentially give the user more control in the virtual world.

In this section, we will review several common techniques used to perform these basic tasks. Note that it is beyond the scope of this lecture to go into great detail on these techniques or on the concepts of 3D UIs in general. The reader should examine ”3D User Interfaces: Theory and Practice” for a rigorous treatment of the subject [Bowman et al. 2004]. Also note for this section, we assume (for the techniques where it makes a difference) that ’y’ is the vertical axis in the world coordinate system.

2.1 Navigation

The motor component of navigation is known as travel (e.g., viewpoint movement). There are several issues to consider when dealing with travel in 3D UIs. One such issue is the control of velocity and/or acceleration. There are many methods for doing this, including gesture, speech controls, sliders, etc. Another issue is that of world rotation. In systems that are only partially spatially surrounding (e.g. a 4-walled CAVE, or a single screen), the user must be able to rotate the world or his view of the world in order to navigate. In fully surrounding systems (e.g. with an HMD or 6-sided CAVE) this is not necessary since the visuals completely surround the user. Next, one must consider whether motion should be constrained in any way, for example by maintaining a constant height or by following the terrain. Finally, at the lowest-level, the conditions of input must be considered - that is, when and how does motion begin and end (click to start/stop, press to start, release to stop, stop automatically at target location, etc.)? Four of the more common 3D travel techniques are gaze-directed steering, pointing, map-based travel, and ”grabbing the air”.

2.1.1 Gaze-Directed Steering

Gaze-directed steering is probably the most common 3D travel technique, although the term ”gaze” is really misleading. Usually no eye tracking is being performed, so the direction of gaze is inferred from the head tracker orientation. This is a simple technique, both to implement and to use, but it is somewhat limited in that you cannot look around while moving [Mine 1995]. Potential examples of gaze-directed steering in video games would be controlling vehicles or traveling around the world in real-time strategy games.

To implement gaze-directed steering, typically a callback function is set up that executes before each frame is rendered. Within this callback, first obtain the head tracker information (usually in the form of a 4x4 matrix). This matrix gives you a transformation between the base tracker coordinate system and the head tracker coordinate system. By also considering the transformation between the world coordinate system and the base tracker coordinates (if any), you can get the total composite transformation. Now, consider the vector $(0, 0, -1)$ in head tracker space (the negative z-axis, which usually points out the front of the tracker). This vector, expressed in world coordinates, is the direction you want to move. Normalize this vector, multiply it by the speed, and then translate the viewpoint by this amount in world coordinates. Note: current ”velocity” is in units/frame. If you want true velocity (units/second), you must keep track of the time between frames and then translate the viewpoint by an amount proportional to that time.

2.1.2 Pointing

Pointing is also a steering technique (where the user continuously specifies the direction of motion). In this case, the hand's orientation is used to determine direction. This technique is somewhat harder to learn for some users, but is more flexible than gaze-directed steering [Bowman et al. 1997]. Pointing is implemented in exactly the same way as gaze-directed steering, except a hand tracker is used instead of the head tracker. Pointing could be used to decouple line of sight and direction of motion in first and third person shooter games.

2.1.3 Map-based Travel

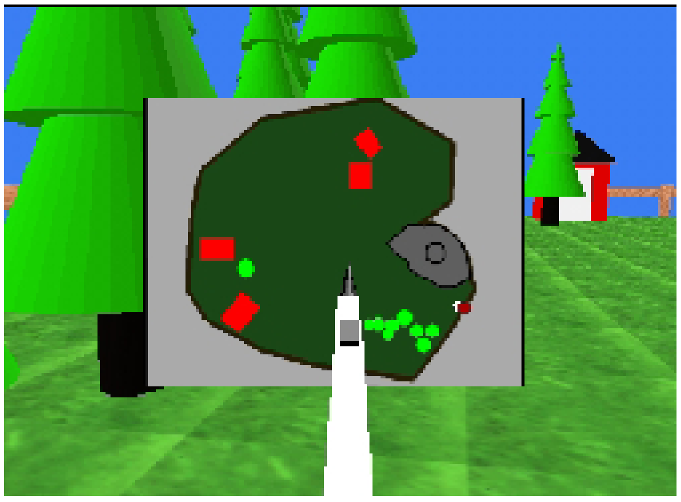


Figure 5: *Dragging a user icon to move to a new location in the world.*

The map-based travel technique is a target-based technique. The user is represented as an icon on a 2D map of the environment. To travel, the user drags this icon to a new position on the map (see Figure 5). When the icon is dropped, the system smoothly animates the user from the current location to the new location indicated by the icon [Bowman et al. 1998]. Map-based travel could be used to augment many of the 2D game maps currently found in many game genres.

To implement this technique, two things must be known about the way the map relates to the world. First, we need to know the scale factor, the ratio between the map and the virtual world. Second, we need to know which point on the map represents the origin of the world coordinate system. We assume here that the map model is originally aligned with the world (i.e. the x direction on the map, in its local coordinate system, represents the x direction in the world coordinate system). When the user presses the button and is intersecting the user icon on the map, then the icon needs to be moved with the stylus each frame. One cannot simply attach the icon to the stylus, because we want the icon to remain on the map even if the stylus does not. To do this, we first find the position of the stylus in the map coordinate system. This may require a transformation between coordinate systems, since the stylus is not a child of the map. The x and z coordinates of the stylus position are the point to which the icon should be moved. We do not cover here what happens if the stylus is dragged off the map, but the user icon should ”stick” to the side of the map until the stylus is moved back inside the map boundaries, since we don’t want the user to move outside the world.

When the button is released, we need to calculate the desired position of the viewpoint in the world. This position is calculated using a transformation from the map coordinate system to the world coordinate system, which is detailed here. First, find the offset in the map coordinate system from the point corresponding to the world origin. Then, divide by the map scale (if the map is 1/100 the size of the world, this corresponds to multiplying by 100). This gives us the x and z coordinates of the desired viewpoint position. Since the map is 2D, we can’t get a y coordinate from it. Therefore, the technique should have some way of calculating the desired height at the new viewpoint. In the simplest case, this might be constant. In other cases, it might be based on the terrain height at that location or some other factors.

Once we know the desired viewpoint, we have to set up the animation of the viewpoint. The move vector $\mathbf{m}$ represents the amount of translation to do each frame (we are assuming a linear path). To find $\mathbf{m}$, we subtract the desired position from the current position (the total movement required), divide this by the distance between the two points (calculated using the distance formula), and multiplied by the desired velocity, so that $\mathbf{m}$ gives us the amount to move in each dimension each frame. The only remaining calculation is the number of frames this movement will take: distance/velocity frames. Note that again velocity is measured here in units/frame, not units/second, for simplicity.

2.1.4 Grabbing the Air

The grabbing the air technique uses the metaphor of literally grabbing the world around you (usually empty space), and pulling yourself through it using hand gestures [Mapes and Moshell 1995]. This is similar to pulling yourself along a rope, except that the rope exists everywhere, and can take you in any direction. The grabbing the air technique has many potential uses in video games including climbing buildings or mountains, swimming, and flying.

To implement the one-handed version of this technique (the two-handed version can get complex if rotation and world scaling is also supported), when the initial button press is detected, we simply obtain the position of the hand in the world coordinate system. Then, every frame until the button is released, get a new hand position, subtract it from the old one, and move the objects in the world by this amount. Alternately, you can leave the world fixed, and translate the viewpoint by the opposite vector. Before exiting the callback, be sure to update the old hand position for use on the next frame. Note it is tempting to implement this technique simply by attaching the world to the hand, but this will have the undesirable effect of also rotating the world when the hand rotates, which can be quite disorienting. You can also do simple constrained motion simply by ignoring one or more of the components of the hand position (e.g. only consider x and z to move at a constant height).

2.2 Selection

3D selection is the process of accessing one or more objects in a 3D virtual world. Note that selection and manipulation are intimately related, and that several of the techniques described here can also be used for manipulation. There are several common issues for the implementation of selection techniques. One of the most basic is how to indicate that the selection event should take place (e.g. you are touching the desired object, now you want to pick it up). This is usually done via a button press, gesture, or voice command, but it might also be done automatically if the system can infer the users intent. One also has to have efficient algorithms for object intersections for many of these techniques. Well discuss a couple of possibilities. The feedback given to the user regarding which object is about to be selected is also very important. Many of the techniques require an avatar (virtual representation) for the users hand. Finally, consider keeping a list of objects that are selectable, so that a selection technique does not have to test every object in the world, increasing efficiency. Four common selection techniques include the virtual hand, ray-casting, occlusion, and arm extension.

2.2.1 The Virtual Hand

The most common selection technique is the simple virtual hand, which does real-world selection via direct touching of virtual objects. In the absence of haptic feedback, this is done by intersecting the virtual hand (which is at the same location as the physical hand) with a virtual object. The virtual hand has great potential in many different video game genres. Examples include selection of sports equipment, direct selection of guns, ammo, and health packs in first person shooter games, as a hand of ”God” in real-time strategy games, and interfacing with puzzles in action/adventure games.

Implementing this technique is simple, provided you have a good intersection/collision algorithm. Often, intersections are only performed with axis-aligned bounding boxes or bounding spheres rather than with the actual geometry of the objects.

2.2.2 Ray-Casting

Another common selection technique is ray-casting. This technique uses the metaphor of a laser pointer an infinite ray extending from the virtual hand [Mine 1995]. The first object intersected along the ray is eligible for selection. This technique is efficient, based on experimental results, and only requires the user to vary 2 degrees of freedom (pitch and yaw of the wrist) rather than the 3 DOFs required by the simple virtual hand and other location-based techniques. Ideally, ray-casting could be used whenever special powers are required in the game or when the player has the ability to select objects at a distance.

There are many ways to implement ray-casting. A brute-force approach would calculate the parametric equation of the ray, based on the hands position and orientation. First, as in the pointing technique for travel, find the world coordinate system equivalent of the vector $(0, 0, -1)$. This is the direction of the ray. If the hands position is represented by (x_h, y_h, z_h), and the direction vector is (x_d, y_d, z_d), then the parametric equations are given by

$$\begin{aligned} x(t) &= x_h + x_d t && (1) \\ y(t) &= y_h + y_d t && (2) \\ z(t) &= z_h + z_d t. && (3) \end{aligned}$$

Only intersections with $t > 0$ should be considered, since we do not want to count intersections behind the hand. It is important to determine whether the actual geometry has been intersected, so first testing the intersection with the bounding box will result in many cases being trivially rejected.

Another method might be more efficient. In this method, instead of looking at the hand orientation in the world coordinate system, we consider the selectable objects to be in the hands coordinate system, by transforming their vertices or their bounding boxes. This might seem quite inefficient, because there is only one hand, while there are many polygons in the world. However, we assume we have limited the objects by using a selectable objects list. Thus, the intersection test we will describe is much more efficient. Once we have transformed the vertices or bounding boxes, we drop the z coordinate of each vertex. This maps the 3D polygon onto a 2D plane (the xy plane in the hand coordinate system). Since the ray is $(0, 0, -1)$ in this coordinate system, we can see that in this 2D plane, the ray will intersect the polygon if and only if the point $(0, 0)$ is in the polygon. We can easily determine this with an algorithm that counts the number of times the edges of the 2D polygon cross the positive x-axis. If there are an odd number of crossings, the origin is inside, if even, the origin is outside.

2.2.3 Occlusion Techniques

Occlusion techniques (also called image plane techniques) work in the plane of the image; object are selected by covering it with the virtual hand so that it is occluded from your point of view [Pierce et al. 1997]. Geometrically, this means that a ray is emanating from your eye, going through your finger, and then intersecting an object. Occlusion techniques could be used for object selection at a distance but instead of using a laser pointer metaphor that ray casting affords, players could simply ”touch” distant objects to select them.

These techniques can be implemented in the same ways as the ray-casting technique, since it is also using a ray. If you are doing the brute-force ray intersection algorithm, you can simply define the rays direction by subtracting the finger position from the eye position.

However, if you are using the 2nd algorithm, you require an object to define the rays coordinate system. This can be done in two steps. First, create an empty object, and place it at the hand position, aligned with the world coordinate system. Next, determine how to rotate this object/coordinate system so that it is aligned with the ray direction. The angle can be determined using the positions of the eye and hand, and some simple trigonometry. In 3D, two rotations must be done in general to align the new objects coordinate system with the ray.

2.2.4 Arm-Extension

The arm-extension (e.g., Go-Go) technique is based on the simple virtual hand, but it introduces a non-linear mapping between the physical hand and the virtual hand, so that the user’s reach is greatly extended [Poupyrev et al. 1996]. Not only useful for object selection at a distance, Go-Go could also be useful traveling through an environment with Batman's grappling hook or Spiderman's web. The graph in Figure 6 shows the mapping between the physical hand distance from the body on the x-axis and the virtual hand distance from the body on the y-axis. There are two regions. When the physical hand is at a depth less than a threshold D, the one-to-one mapping applies. Outside D, a non-linear mapping is applied, so that the farther the user stretches, the faster the virtual hand moves away.

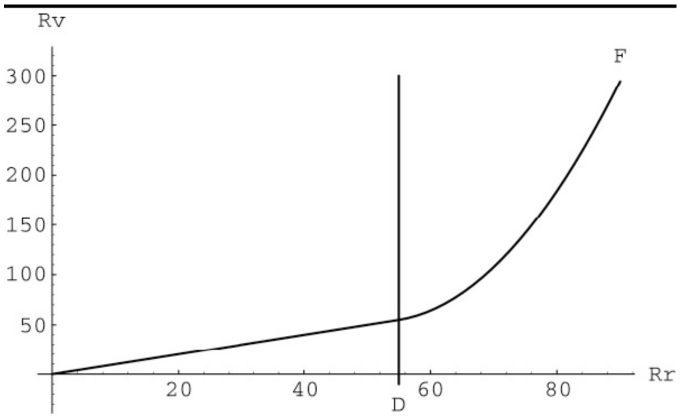


Figure 6: *The nonlinear mapping function used in the Go-Go selection technique.*

To implement Go-Go, we first need the concept of the position of the users body. This is needed because we stretch our hands out from the center of our body, not from our head (which is usually the position that is tracked). We can implement this using an inferred torso position, which is defined as a constant offset in the negative y direction from the head. A tracker could also be placed on the users torso.

Before rendering each frame, we get the physical hand position in the world coordinate system, and then calculate its distance from the torso object using the distance formula. The virtual hand distance can then be obtained by applying the function shown in the graph in Figure 6. $d^{2.3}$ (starting at D) is a useful function in many environments, but the exponent used depends on the size of the environment and the desired accuracy of selection at a distance. Once the distance at which to place the virtual hand is known, we need to determine its position. The most common implementation is to keep the virtual hand on the ray extending from the torso and going through the physical hand. Therefore, if we get a vector between these two points, normalize it, multiply it by the distance, then add this vector to the torso point, we obtain the position of the virtual hand. Finally, we can use the virtual hand technique for object selection.

2.3 Manipulation

As we noted earlier, manipulation is connected with selection, because an object must be selected before it can be manipulated. Thus, one important issue for any manipulation technique is how well it integrates with the chosen selection technique. Many techniques, as we have said, do both: e.g. simple virtual hand, ray-casting, and go-go. Another issue is that when an object is being manipulated, you should take care to disable the selection technique and the feedback you give the user for selection. If this is not done, then serious problems can occur if, for example, the user tries to release the currently selected object but the system also interprets this as trying to select a new object. Finally, thinking about what happens when the object is released is important. Does it remain at its last position, possibly floating in space? Does it snap to a grid? Does it fall via gravity until it contacts something solid? The application requirements will determine this choice. Three common manipulation techniques include HOMER, Scaled-World Grab, and World-in-Miniature.

2.3.1 HOMER

The Hand-Centered Object Manipulation Extending Ray-Casting (HOMER) technique uses ray-casting for selection and then moves the virtual hand to the object for hand-centered manipulation [Bowman and Hodges 1997]. The depth of the object is based on a linear mapping. The initial torso-physical hand distance is mapped onto the initial torso-object distance, so that moving the physical hand twice as far away also moves the object twice as far away. Also, moving the physical hand all the way back to the torso moves the object all the way to the users torso as well.

Like Go-Go, HOMER requires a torso position, because you want to keep the virtual hand on the ray between the users body (torso) and the physical hand. The problem here is that HOMER moves the virtual hand from the physical hand position to the object upon selection, and it is not guaranteed that the torso, physical hand, and object will all line up at this time. Therefore, we calculate where the virtual hand would be if it were on this ray initially, then calculate the offset to the position of the virtual object, and maintain this offset throughout manipulation.

When an object is selected via ray-casting, first detach the virtual hand from the hand tracker. This is due to the fact that if it remained attached but the virtual hand model is moved away from the physical hand location, a rotation of the physical hand will cause a rotation and translation of the virtual hand. Next, move the virtual hand in the world coordinate system to the position of the selected object, and attach the object to the virtual hand in the scene graph (again, without moving the object in the world coordinate system).

To implement the linear depth mapping, we need to know the initial distance between the torso and the physical hand d_h, and between the torso and the selected object d_o. The ratio d_o/d_h will be the scaling factor.

For each frame, we need to set the position and orientation of the virtual hand. The selected object is attached to the virtual hand, so it will follow along. Setting the orientation is relatively easy. Simply copy the transformation matrix for the hand tracker to the virtual hand, so that their orientation matches. To set the position, we need to know the correct depth and the correct direction. The depth is found by applying the linear mapping to the current physical hand depth. The physical hand distance is simply the distance between it and the torso, and we multiply this by the scale factor d_o/d_h to get the virtual hand distance. We then obtain a normalized vector between the physical hand and the torso, multiply this vector by the virtual hand distance, and add the result to the torso position to obtain the virtual hand position.

2.3.2 Scaled-World Grab

The scaled-world grab technique (see Figure 7) is often used with occlusion selection. The idea is that since you are selecting the object in the image plane, you can use the ambiguity of that single image to do some magic. When the selection is made, the user is scaled up (or the world is scaled down) so that the virtual hand is actually touching the object that it is occluding. If the user does not move (and the graphics are not stereo), there is no perceptual difference between the images before and after the scaling [Mine et al. 1997]. However, when the user starts to move the object and/or his head, he realizes that he is now a giant (or that the world is tiny) and he can manipulate the object directly, just like the simple virtual hand.

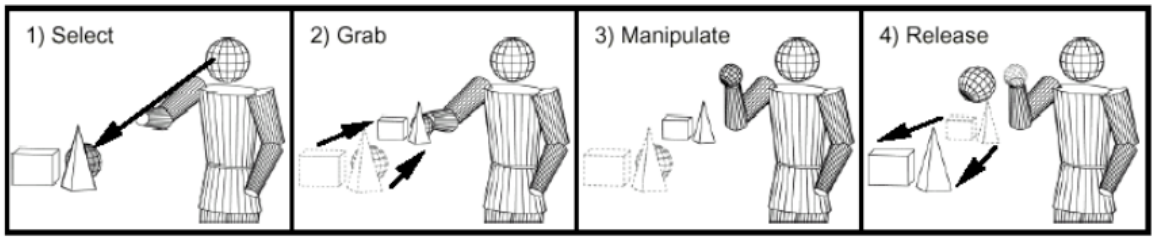


Figure 7: *An illustration of the scaled-world grab technique.*

To implement scaled-world grab, correct actions must be performed at the time of selection and release. Nothing special needs to be done in between, because the object is simply attached to the virtual hand, as in the simple virtual hand technique. At the time of selection, scale the user by the ratio (distance from eye to object / distance from eye to hand). This scaling needs to take place with the eye as the fixed point, so that the eye does not move, and should be uniform in all three dimensions. Finally, attach the virtual object to the virtual hand. At the time of release, the opposite actions are done in reverse. Re-attach the object to the world, and scale the user uniformly by the reciprocal of the scaling factor, again using the eye as a fixed point.

2.3.3 World-in-Miniature

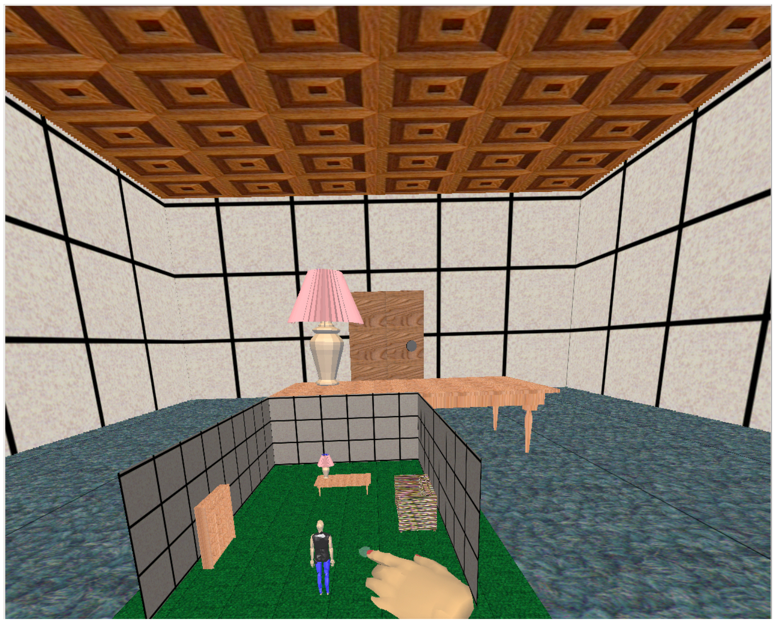


Figure 8: *An example of a WIM.*

The world-in-miniature (WIM) technique uses a small dollhouse version of the world to allow the user to do indirect manipulation of the objects in the environment (see Figure 8). Each of the objects in the WIM are selectable using the simple virtual hand technique, and moving these objects causes the full-scale objects in the world to move in a corresponding way [Stoakley et al. 1995]. The WIM can also be used for navigation by including a representation of the user, in a way similar to the map-based travel technique, but including the 3rd dimension [Pausch et al. 1995].

To implement the WIM technique, first create the WIM. Consider this a room with a table object in it.

The WIM is represented as a scaled down version of the room, and is attached to the virtual hand. The table object does not need to be scaled, because it will inherit the scaling from its parent (the WIM room). Thus, the table object can simply be copied within the scene graph.

When an object in the WIM is selected using the simple virtual hand technique, first match this object to the corresponding full-scale object. Keeping a list of pointers to these objects is an efficient way to do this step. The miniature object is attached to the virtual hand, just as in the simple virtual hand technique. While the miniature object is being manipulated, simply copy its position matrix (in its local coordinate system, relative to its parent, the WIM) to the position matrix of the full-scale object. Since we want the full-scale object to have the same position in the full-scale world coordinate system as the miniature object does in the scaled-down WIM coordinate system, this is all that is necessary to move the full-scale object correctly.

2.4 System Control

System control provides a mechanism for users to issue a command to either change the mode of interaction or the system state. In order to issue the command, the user has to select an item from a set. System control is a wide-ranging topic, and there are many different techniques to choose from such as the use of graphical menus, voice commands, gestures, and tool selectors. For the most part, these techniques are not difficult to implement, since they mostly involve selection. For example, virtual menu items might be selected using ray-casting. For all of the techniques, good visual feedback is required, since the user needs to know not only what he is selecting, but what will happen when he selects it. In this section, we briefly highlight some of the more common system control techniques.

2.4.1 Graphical Menus

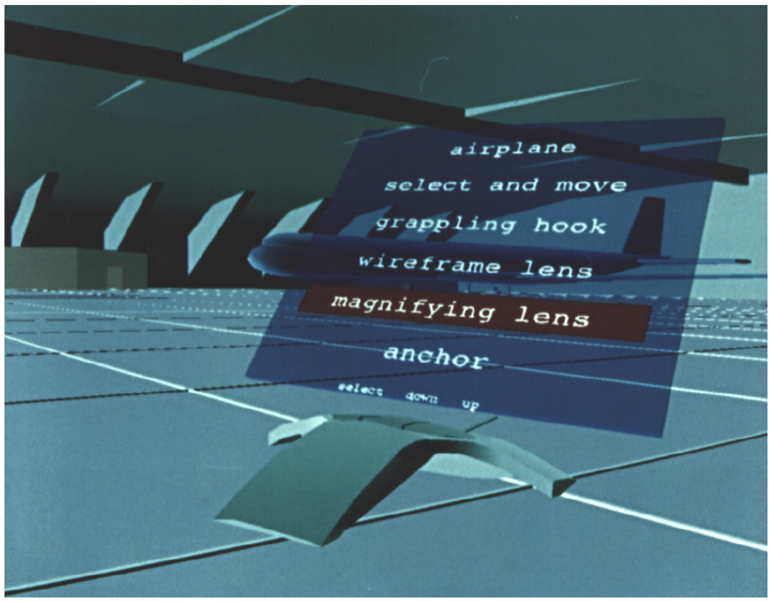


Figure 9: *The Virtual Tricorder: an example of a graphical menu with device-centered placement.*

Graphical menus can be seen as the 3D equivalent of 2D menus. Placement influences the access of the menu (correct placement can give a strong spatial reference for retrieval), and the effects of possible occlusion of the field of attention. The paper by Feiner et al. is an important source for placement issues [Feiner et al. 1993]. The authors divided placement into surround-fixed, world-fixed and display-fixed windows. The subdivision of placement can, however, be made more subtle. World-fixed and surround-fixed windows, the term Feiner et al. use to describe menus, can be subdivided into menus which are either freely placed into the world, or connected to an object. Display-fixed windows can be renamed, and made more precise, by referring to their actual reference frame: the body. Body-centered menus, either head referenced or body-referenced, can supply a strong spatial reference frame. One particularly interesting possible effect of body-centered menus is eyes-off usage, in which users can perform system control without having to look at the menu itself. The last reference frame is the group of device-centered menus. Device-centered placement provides the user with a physical reference frame (see Figure 9) [Wloka and Greenfield 1995]. A good example is the placement of menus on a responsive workbench, where menus are often placed at the border of the display device.

We can subdivide graphical menus into hand-oriented menus, converted 2D menus, and 3D widgets. One can identify two major groups of hand-oriented menus. 1DOF menus are menus which use a circular object on which several items are placed. After initialization, the user can rotate his/her hand along one axis until the desired item on the circular object falls within a selection basket. User performance is highly dependent on hand and wrist physical movement and the primary rotation axis should be carefully chosen. 1DOF menus have been made in several forms, including the ring menu, sundials, spiral menus (a spiral formed ring menu), and a rotary tool chooser. The second group of hand-oriented menus are hand-held-widgets, in which menus are stored at a body-relative position.

The second group is the most often applied group of system control interfaces: converted 2D widgets. These widgets basically function the same as in desktop environments, although one often has to deal with more DOFs when selecting an item in a 2D widget. Popular examples are pull-down menus, pop-up menus, flying widgets, toolbars and sliders.

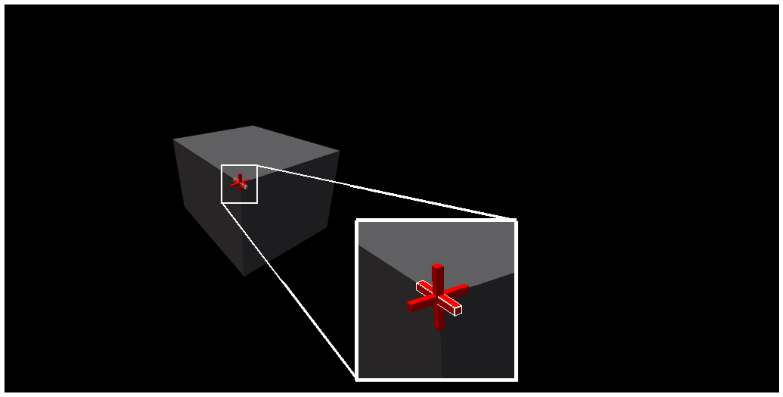


Figure 10: *An example of a 3D widget used to scale a geometric object.*

The final group of graphical menus is the group known as 3D widgets [Conner et al. 1992]. In a 3D world, widgets often mean moving system control functionality into the world or onto objects (see Figure 10). This matches closely with the definition of widgets given by Conner et al., ”widgets are the combination of geometry and behavior”. This can also be thought of as ”moving the functionality of a menu onto an object.” A very important issue when using widgets is placement. 3D widgets differ from the previously discussed menu techniques (1DOF and converted 2D menus) in the way the available functions are mapped: most often, the functions are co-located near an object, thereby forming a highly context-sensitive menu.

2.4.2 Voice Commands

Voice input allows the initialization, selection and issuing of a command. Sometimes, another input stream (like a button press) or a specific voice command is used to allow the actual activation of voice input for system control. The use of voice input as a system control technique can be very powerful: it is hands-free and natural. Still, continuous voice input is tiring, and can not be used in every environment. Furthermore, the voice recognition engine often has a limited vocabulary. In addition, the user first needs to learn the voice commands before they can be applied.

Problems often occur when applications are more complex, and the complete set of voice commands can not be remembered. The structural organization of voice commands is invisible to the user: often no visual representation is coupled to the voice command in order to see the available commands. In order to prevent mode errors, it is often very important to supply the user with some kind of feedback after she has issued a command. This can be achieved by voice output, or by the generation of certain sounds. A very interesting way of supporting the user when interacting with voice and invisible menu structures can be found in telecommunication: using a telephone to access information often poses the same problems to the user as using voice commands in a virtual environment.

2.4.3 Gestures and Postures

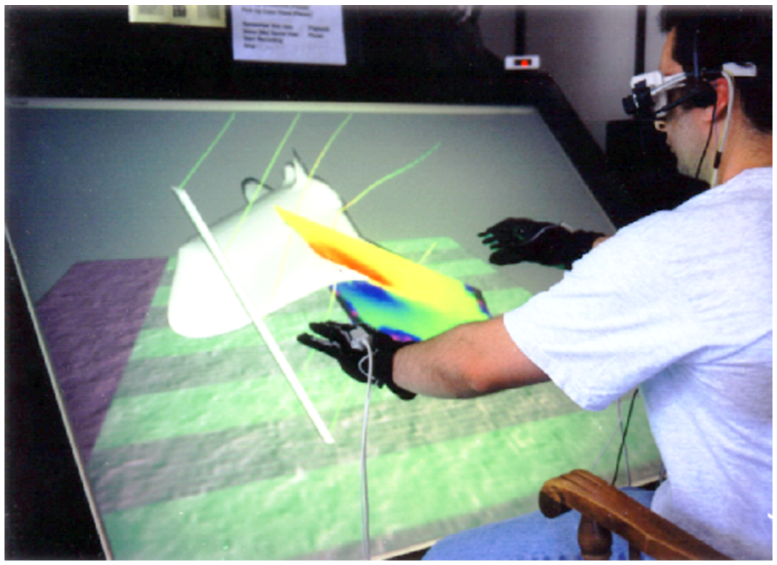


Figure 11: *A user interacting with a dataset for visualizing a flow field around a space shuttle. The user simultaneously manipulates the streamlines with his left hand and the shuttle with his right hand while viewing the data in stereo. The user asked for these tools using speech input.*

When using gestural interaction, we apply a hand-as-tool metaphor: the hand literally becomes a tool. When applying gestural interaction, the gesture is both the initialization and the issuing of a command, just as in voice input. When talking about gestural interaction, we refer, in this case, to gestures and postures, not to gestural input with pen-and-tablet or similar metaphors. There is a significant difference between gestures and postures: postures are static movements (like pinching), whereas gestures include a change of position and/or orientation of the hand. A good example of gestures is the usage of sign language.

Gestural interaction can be a very powerful system control technique. In fact, gestures are relatively limitless when it comes to their potential uses in video games. Gestures can be use to communicate with other players, to cast spells in a role playing game, call pitches or give signs in a baseball game, and issues combination attacks in action games. However, one problem with gestural interaction is that the user needs to learn all the gestures. Since the user can normally not remember more than about 7 gestures (due to the limited capacity of the working memory), inexperienced users can have significant problems with gestural interaction, especially when the application is more complex and requires a larger amount of gestures. Users often do not have the luxury of referring to a graphical menu when using gestural interaction - the structure underneath the available gestures is completely invisible. In order to make gestural interaction easier to use for a less advanced user, strong feedback, like visual cues after initiation of a command, might be needed. An example of application that used a gestural system control technique is MSVT [LaViola 2000]. This application combined gestures and voice input to create a multimodal interface for exploratory scientific visualization (see Figure 11).

2.4.4 Tools

We can identify two different kinds of tools, namely physical tools and virtual tools. Physical tools are context-sensitive input devices, which are often referred to as props. A prop is a real-world object which is duplicated in the virtual world. A physical tool might be space multiplexed (the tool only performs one function) or time multiplexed, when the tool performs multiple functions over time (like a normal desktop mouse). One accesses a physical tool by simply reaching for it, or by changing the mode on the input device itself.

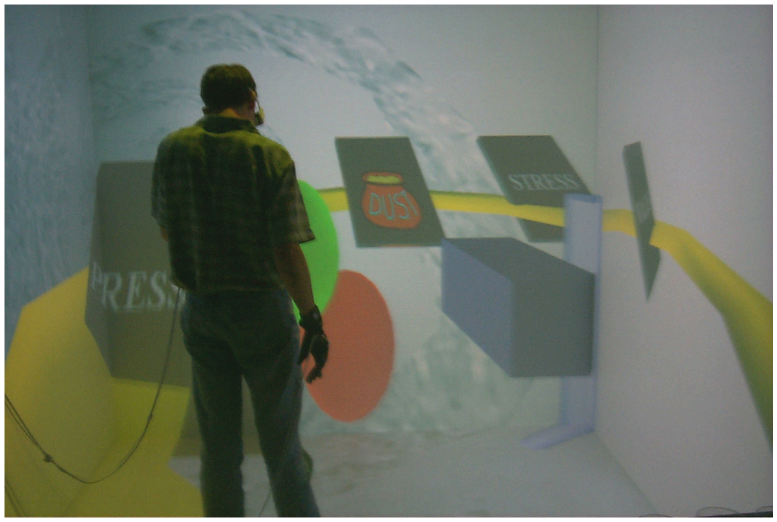


Figure 12: *An example of a virtual toolbelt. The user looks down to invoke the belt and can grab tools and use them in the virtual environment.*

Virtual tools are tools which can be best exemplified with a toolbelt (see Figure 12). Users wear a virtual toolbelt around the waist, from which the user can access specific functions by grabbing at particular places on belt, as in the real world. Sometimes, functions on a toolbelt are accessed via the same principles as used with graphical menus, where one should look at the menu itself. The structure of tools is often not complex: as stated before, physical tools are either dedicated devices for one function, or one can access several (but not many) functions with one tool. Sometimes, a physical tool is the display medium for a graphical menu. In this case, it has to be developed in the same way as graphical menus. Virtual tools often use proprioceptive cues for structuring.

It is still unclear how to best design tools for system control. Still, some general design issues can be stated. In the case of physical tools, the form of the tool often strongly communicates the function one can perform with the device, so take care with the form when developing new props. The form of a tool can highly influence the directness and familiarity with the device as well. With respect to virtual tools which can not be used without looking at them, their representation can be very similar to graphical menus.

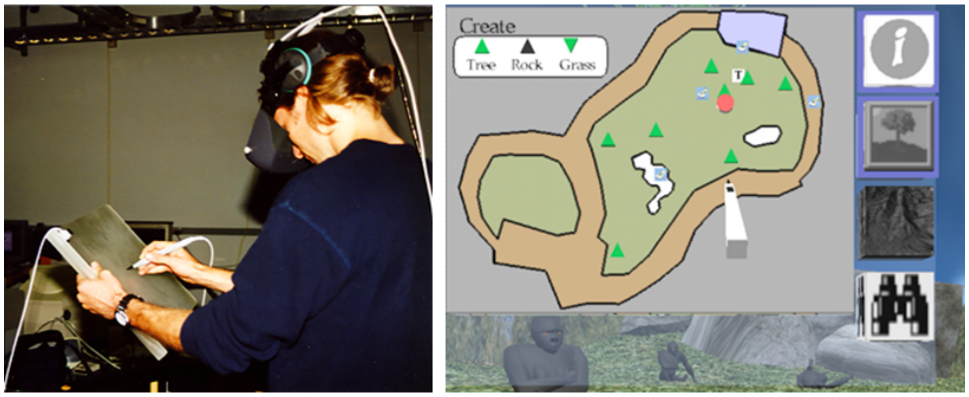


Figure 13: *The Pen and Tablet Metaphor. The image on the left shows the user holding the pen and tablet and the image on the right shows what the user's sees in the virtual world.*

One example of a commonly used physically-based tool is the pen and tablet. Users hold a large plastic tablet on which a (traditional) 2D interface is displayed in the virtual world (see Figure 13). Users are able to use graphical menu techniques and can move objects with a stylus within a window on the tablet. The tablet supplies the user with strong physical cues with respect to the placement of the menu, and allows increased performance due to faster selection of menu items [Bowman et al. 1998].

For the implementation of this technique, the most crucial thing is the registration (correspondence) between the physical and virtual pens and tablets. The tablets, especially, must be the same size and shape so that the edge of the physical tablet, which the user can feel, corresponds to the edge of the virtual tablet as well. In order to make tracking easy, the origin of the tablet model should be located at the point where the tracker is attached to the physical tablet, so that rotations work properly. Even with care, its difficult to do these things exactly right, so a final tip is to include controls within the program to tweak the positions and/or orientations of the virtual pen and tablet, so that they can be into registration if there's a problem.

Another useful function to implement is the ability for the system to report (for example when a callback function is called) the position of the stylus tip in the tablet coordinate system, rather than in the world or user coordinate systems. This can be used for things like the map-based travel technique described earlier.

3 3D Interfaces with 2D and 3D Cameras

One of the most active research areas relative to human/computer interaction has been in processing real-time video input. The no wires, no batteries aspect of video-as-input approaches holds many practical advantages, especially for consumer applications [Intel 1999]. In addition, the richness of information visual sensing offers and the natural familiarity users have with such information make visual tracking an attractive solution for many applications.

Just as with computer graphics, video imaging involves perspective projection. The simplified, often-used model for a camera involves light from a 3D scene being projected onto a 2D imaging plane, much the way computer graphics are projected onto a 2D display. However, recovering 3D information from a 2D image is impossible in the general case and complex even in highly constrained situations. Generally, it requires making strong assumptions about the scene or having a priori models of objects in the scene.

3.1 EyeToy

Figure 14: *EyeToy.*

The PlayStation EyeToy launched in 2003. Though essentially a webcam, it was designed specifically to enable players to move their bodies to interact with PlayStation 2 games. Thus, its default framerate was 60 frames/sec, twice as fast as other webcams. Also, it had a 56 degree diagonal field of view, to allow complete imaging of the upper torso and outstretched arms when standing at a reasonable distance (6 to 8 feet).

3.1.1 Motion Detection

The first EyeToy games used a ”magic mirror” paradigm in which video of the player was overlayed with computer graphics generated by the game. The player interacted with the game graphics by moving his body. To accomplish this interaction, these games relied almost exclusively upon motion detection. The most straight-forward implementation of motion detection involves comparing the current video frame with the previous video frame to generate a difference image. This difference between the frames is assumed to be caused by player motion. The difference image can be further processed to create a binary motion mask, often by performing a simple thresholding step. Many documented techniques exist for computing the difference image, and various techniques also exist for creating a motion mask from a difference image.

The EyeToy: Play game ”Wishi-Washi” clearly demonstrates this technology, as the entire display is covered by virtual dirt and the player must wipe the dirt away. Pixels contained in the motion mask are removed from the virtual dirt mask, using the simple boolean logic:

$$DirtMask(i+1) \quad = \quad DirtMask(i)\ \&\ !MotionMask(i). \tag{4}$$

Other games evaluate the motion mask only in particular regions, to decide if player motion is occurring in a particular area of the screen that has some significance for the game. For example, this can be used to have the player touch fixed targets, such as the four corners of the screen. By integrating (adding) the motion mask in a region over a period of time, it is possible to see if continual motion is occurring in an area. Such areas are sometimes called ”rubby buttons” because the player feels as if they are ”rubbing” the screen area to activate. This technique is especially useful for system control because the integration is much more reliable than motion detection for a single video frame (see Figure 15).

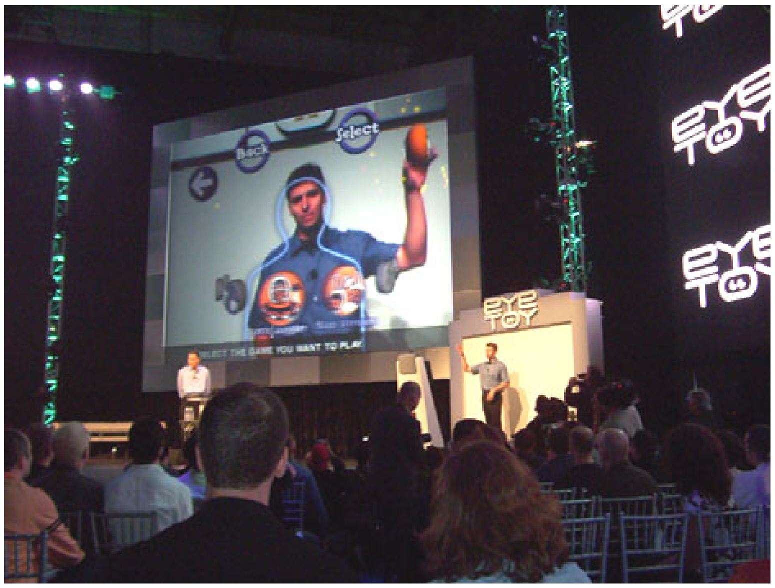


Figure 15: *Motion detection for menu navigation in EyeToy: Play.*

Many EyeToy games perform dynamic collision detection between virtual game objects and the motion mask (see Figure 16). Note that although both the player and the virtual game objects may move in 3D, this collision detection is a 2D operation because the motion mask is a 2D representation of the player motion. Thus, dynamic collision detection is essentially the same operation described above, except the motion mask region that is evaluated is dynamically changed to match the 2D projection of the virtual game object. Because the motion mask collision detection is 2D, the game interactions achievable using motion detection have a limited 2D feeling.

3.1.2 Color Tracking

Color tracking is another interaction technique that was explored using EyeToy and PlayStation 2. Players could interact with a 3D scene by moving known brightly colored objects that were visually tracked.

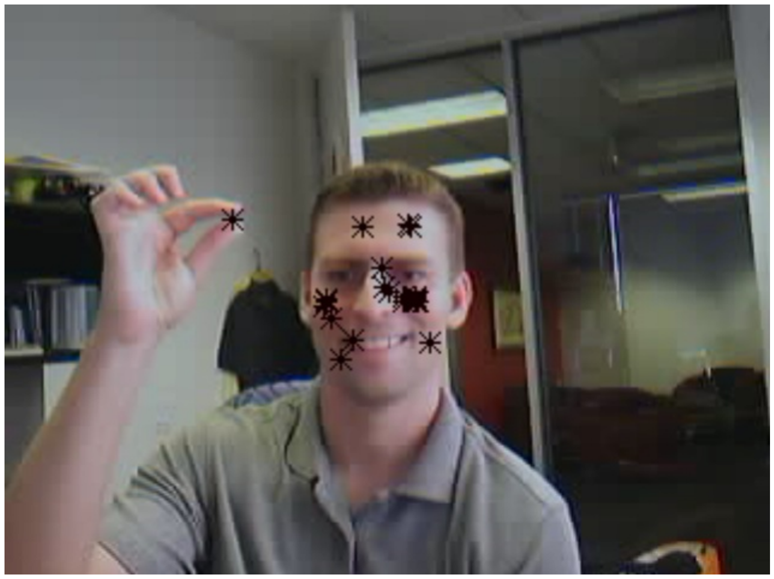


Figure 16: *The virtual spiders move based on the player motion.*

Figure 17 shows a demonstration presented at SIGGRAPH 2000 Emerging Technologies [Marks 2000]; as the player moves the bright green toy he is holding, a virtual sword object moves in direct response. At SIGGRAPH 2001, two more demonstrations were shown based on color tracking of colored foam spheres [Marks et al. 2001].

Color tracking can be broken into two main steps: segmentation and pose recovery. Segmentation consists of labeling every video pixel that corresponds to the object being tracked. General shape segmentation for arbitrary objects is a difficult and computationally expensive problem. However, by designing the tracking object to have distinctive, solid, saturated colors, the object segmentation process can be simplified to basic color segmentation. By choosing extremely saturated colors, the likelihood of these colors occurring in the environment is reduced. Color segmentation can be accomplished in a single pass through the image using chrominance banding/thresholding. Figure 18 shows the results of segmentation for a ”mace” object consisting of a large orange ball mounted on a blue stick. By using only chrominance and not luminance, the segmentation process can be more robust to variation caused by scene lighting.

Pose recovery consists of converting 2D image data into 3D object pose (position and/or orientation). Pose recovery can be accomplished robustly for certain shapes of known physical dimensions by measuring the statistical properties of the shape’s 2D projection. In this manner, for a sphere the 3D position can be recovered (but no orientation), and for a cylinder, the 3D position and a portion of the orientation can be recovered. Multiple objects can be also be combined for complete 3D pose recovery, though occlusion issues arise. By computing statistical properties using the entire segmentation region rather than using edges/features, extent, or other linear measurements, the pose measurement can be less quantized; this is especially important given EyeToy’s low-resolution video.

When color tracking works properly, it enables true 3D interaction that can lead to many interesting new entertainment experiences such as those shown at SIGGRAPH 2000 and 2001. However, primarily due to the variability of scene lighting and background, it is difficult to make robust even with brightly colored toy objects. Later, we describe how the design of the PlayStation Move motion controller addressed these issues.

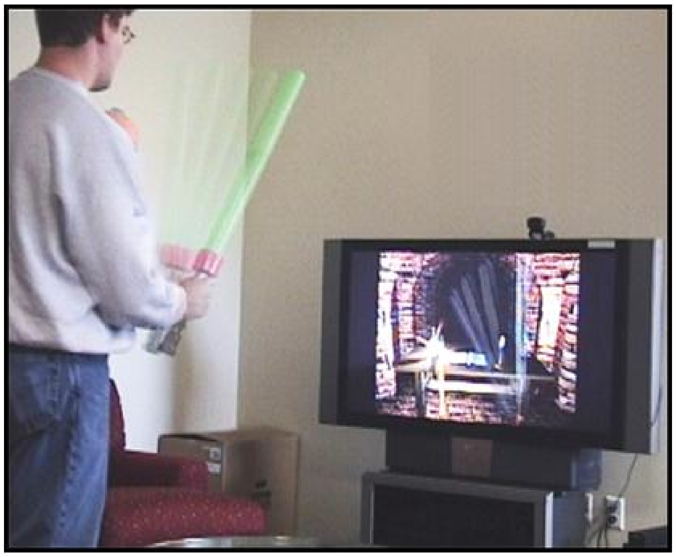


Figure 17: *Color tracking using EyeToy. A virtual game sword moves in response to the green toy.*

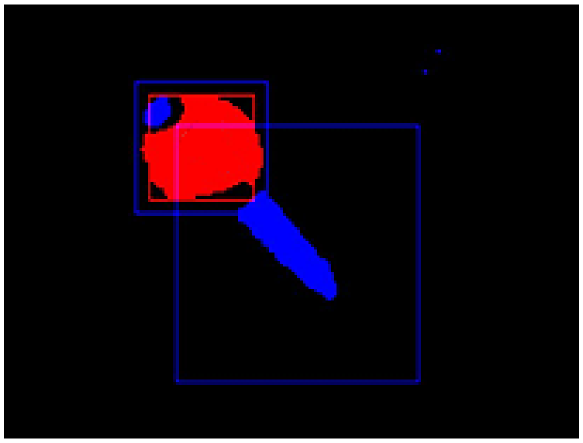


Figure 18: *Color thresholding for object segmentation. The object is a large sphere mounted on a stick.*

3.2 PlayStation Eye

The PlayStation Eye is a USB peripheral device designed to be used with PlayStation 3. Just as EyeToy was created to match the capabilities of PS2, PlayStation Eye was created specifically for PS3. Its primary intended use is for interactive entertainment similar to EyeToy games, but taken to a new level on PS3. Secondary uses include video chat, video movie capture, and still-image snapshots. Just as with EyeToy, the manufacturing cost of the device was required to be low for successful deployment, so PlayStation Eye's video capability was designed to maximize utility given cost constraints. Its resolution is four times that of EyeToy, its low-light sensitivity is significantly improved, and it transfers uncompressed video data to avoid compression artifacts that could compromise video processing.

Besides improving upon the video sensing capability of EyeToy, PlayStation Eye was also designed to enable new interactive experiences by adding 2 new features. The first feature is a lens that can be manually switched to give either a 56 degrees or 75 degrees diagonal field of view. The first setting (similar to EyeToy) is appropriate for imaging a player's upper torso, while the wider view of the second setting

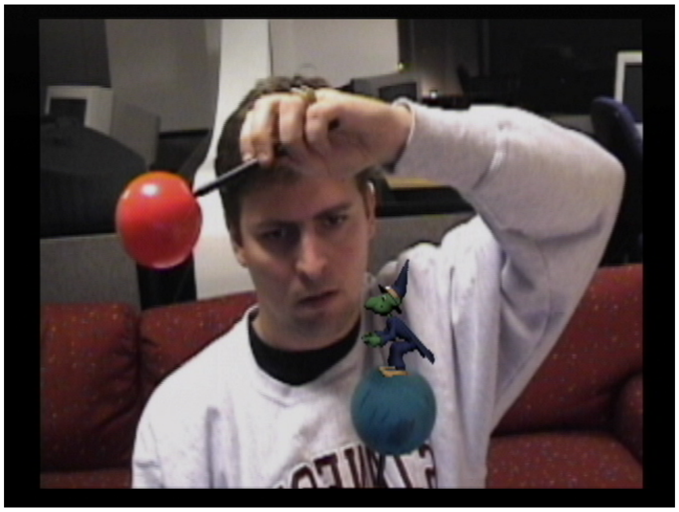


Figure 19: *3D color tracking for augmented reality. The virtual witch character "stands" on the blue sphere and "watches" the red sphere.*

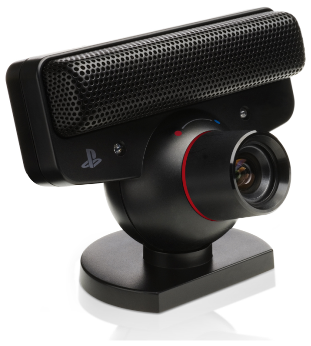


Figure 20: *The PlayStation Eye.*

allows for imaging a much larger workspace that can include the entire body of two players. The second feature is a small-baseline linear 4-microphone array that enables spatial audio filtering so that voice-input applications can achieve good signal-to-noise without the encumbrance of a headset or hand-held microphone.

PlayStation 3 game applications using PlayStation Eye are in active development today. In addition, the PlayStation Eye is an essential part of the PlayStation Move motion control system described later. There are many computer vision techniques that are possible given the combination PlayStation Eye and PlayStation 3; the following sections highlight two techniques that have shown promise for 3D spatial interaction.

3.2.1 Face Tracking

Face tracking for controlling a 3D game character was demonstrated using EyeToy and PlayStation 2 at SIGGRAPH 2003 [Marks et al. 2003]. The player could lean left or right, and jump or duck, and the virtual game character (riding on a futuristic hover board) would perform corresponding game actions. The player also held colored objects to contol the character's arms independently using color tracking. The game EyeToy:Antigrav similarly used face tracking to control the actions of a virtual character, and used motion detection to control the character's arms.

A different use of face tracking for game interaction was first shown publicly on stage at the D.I.C.E. video game summit in 2005. The position of the face was used to move the player in a 1st-person game, allowing the player to "duck" behind cover and "peek" around corners or over cover. This natural interface for games is very similar to "Desktop VR" or "Fish Tank VR" because of the 1st-person rendering of the game scene. The resulting motion parallax provides a compelling 3D effect, famously demonstrated in a 2007 video created by Johnny Chung Lee.

Though face tracking was possible using EyeToy and PlayStation 2, its viability for games is much greater using PlayStation Eye and PlayStation 3. The improved video quality of PlayStation Eye allows for more reliable tracking, and the increased processing capability of PlayStation 3 easily accommodates face tracking and enables more advanced techniques such real-time face detection and face recognition. In addition, since the PlayStation Eye is part of the PlayStation Move system, the uses of face tracking described above are ideal to combine with motion control. For example, face tracking can be used to provide a player more complete control of a virtual character or to allow the player to view the virtual scene from a different perspective while using the motion controllers to manipulate game objects.

3.2.2 Marker Tracking

Another interesting technique for 3D spatial interaction that has been used for games is marker tracking. Though various many forms of marker tracking exist, a common practice is to use "cards" with distinctive markings that make detection and 3D pose tracking possible. Two famous systems for doing this are CyberCode [Rekimoto and Ayatsuka 2000], developed by the Sony Computer Science Laboratory, and ARToolKit [Kato and Billinghurst 1999], originally developed by Hirokazu Kato of the Nara Institute of Science and Technology. These technologies can be used for real-time tracking from video, allowing a virtual object to be rendered as if it was rigidly attached to the card.

The first title for PlayStation Eye was The Eye of Judgment, which used the CyberCode technology to render virtual monsters in 3D onto playing cards. A special mode of the game allowed players to manipulate a card in 3D in front of the camera, and motion detection was used to allow players to "poke" the monsters to demonstrate their attack animations. In Europe, the PlayStation Eye game EyePet recently launched with a "Magic Card" included in the game box. The specially marked card is used by the player to interact with a virtual pet in variety of ways.

Though markers are generally designed to be black and white so as to have high contrast, tracking robustness is still an issue in poorly lit environments. Also, since the markers are often printed onto 2D planes, the tracking quality degrades as the marker plane moves away from parallel to the camera imaging plane. Finally, tracking also degrades with 3D distance as the marker becomes less visually distinct.

3.3 3D Cameras

Though a standard 2D camera can be used to extract 3D spatial data for special known objects, the same cannot be done for arbitrary scenes or for tracking people. To address this, many companies have sought to create 3D cameras, also known as depth cameras or Z cameras. Generally, such cameras provide not only color for every pixel, but also distance, or some parameter directly related to distance, such as disparity.

Knowing the distance for every pixel, or Z-sensing, provides two major benefits. First, it provides a very powerful segmentation mechanism for distinguishing between the foreground and the background.

Second, Z-sensing provides 3D information for unstructured scenes. Ultimately, this is the real power of Z-sensing, because it is an ideal complement to other computer vision methods. While motion and color detection are powerful attention mechanisms, and pattern matching is a powerful correspondence mechanism, Z-sensing provides a powerful segmentation mechanism and 3D spatial information. This combination enables many new possibilities, especially for augmented reality and real-time motion capture applications.

3.3.1 Technology Overview

Many techniques exist for obtaining 3D information in a scene. The most well known is computer stereo vision or stereopsis, which involves finding correspondences in images from two or more offset cameras and then using triangulation to obtain depth. In practice, using computer stereo vision for Z-sensing is difficult because finding correspondences at every image location for arbitrary scenes has many issues, including the aperture problem, parallax occlusion, and camera variation.

Recently, several companies have created Z-sensing cameras that use active infrared illumination and infrared camera sensors. One such approach is based on the same triangulation concept as computer stereo vision, but it uses just one camera and an offset infrared light that projects a special pattern (see Figure 21). To calculate depth, the camera data is corresponded to the known projected pattern, which is carefully designed to facilitate this process.

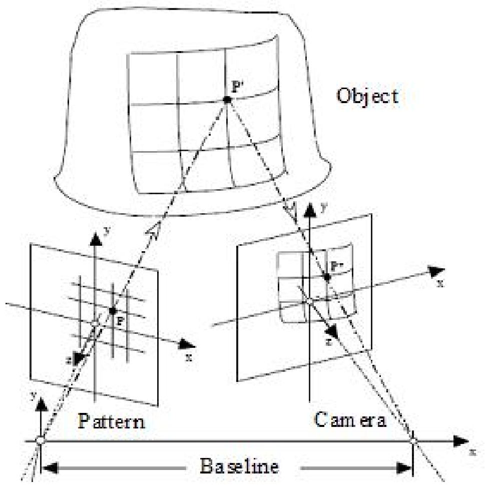


Figure 21: *3D pixel data can be computed from triangulation using one camera and patterned illumination emitted from a fixed baseline.*

Other active illumination Z-sensing cameras use time-of-flight techniques instead of triangulation to measure depth. Because the speed of light is very fast, the high-speed electronics required have only recently become available for such devices. Two very different time-of-flight approaches are: 1) pulsed light with active electronic shuttering, and 2) modulated light with phase detection. The first technique involves pulsing a light source and actively controlling an electronic shutter to truncate the amount of the reflected pulse that is captured (see Figure 22). The further away the reflection occurs, the less of the pulse that will have traveled back to the camera. To compensate for scene reflectance, the shuttered image must be normalized by an unshuttered image. By choosing the pulse length and shutter timing, the near and far limits for the Z-sensing can be set. This approach relies heavily upon the ability to precisely control the electronic shutter and light pulse.

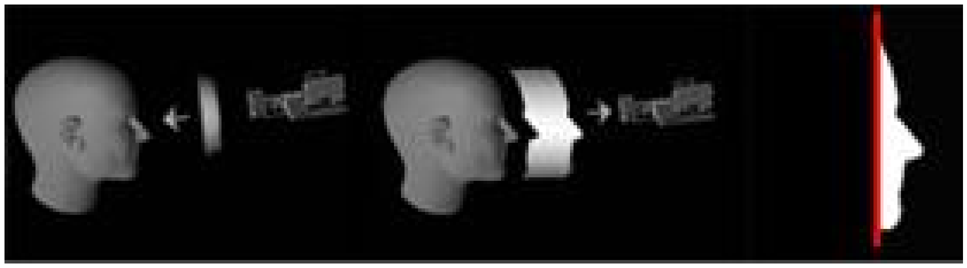


Figure 22: *3D pixel data can be computed based on time-of-flight by using active shuttering to truncate the reflection of a light pulse.*

The second technique involves modulating the light source at a known frequency. Depending on the distance, the reflected light will arrive back at the camera at a different phase compared to the emitter. This phase shift is measured to compute depth using a special sensor, and the modulation frequency can be adjusted to control the range of the camera. Note that the phase ”wraps around” beyond one wavelength, which introduces an ambiguity, so the modulation frequency must be chosen to avoid this.

One major benefit of active illumination approaches is that they do not rely on scene illumination for Z-sensing, so depth can be measured even in complete darkness. One drawback of current active illumination Z-sensing cameras is that the RGB and Z are not perfectly corresponded. This is because they use separate cameras for the RGB and Z sensing, thereby introducing some parallax. Several sensor companies are investigating the possibility of RGB-IR sensors to help address this issue and potentially lower costs by requiring only a single sensor and lens.

3.3.2 Foreground/Background Separation

Segmentation is one of the most difficult problems in traditional monocular computer vision. Separating the foreground from the background in a scene is a specific use of segmentation that is important to applications such as movie special effects and video compression. For movies, segmentation is typically solved with chroma keying (blue screen) and requires a controlled backdrop. Z-sensing provides a very similar segmentation capability but does not require a controlled backdrop (see Figure 23).

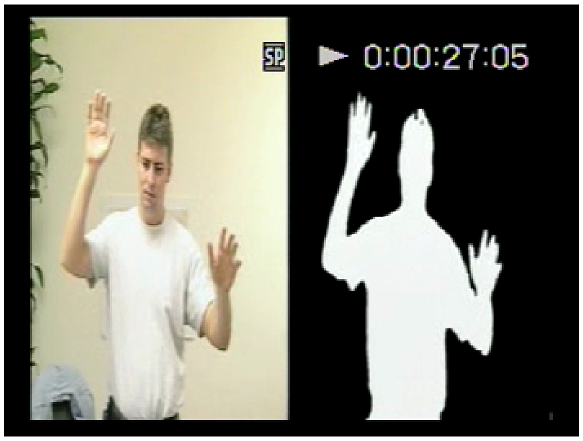


Figure 23: *Segmentation of the foreground from the background based on 3D pixel data.*

A simple approach for using Z-sensing to separate foreground and background is to simply threshold the

Z data. However, this approach typically includes portions of the scene which are near to the camera but conceptually would be thought of as background. To avoid this, a more effective approach is to use the same background subtraction techniques that are often used for RGB data. Background subtraction techniques actually work much better for Z data, because Z data is not affected by shadows and illumination change.

3.3.3 Augmented Reality

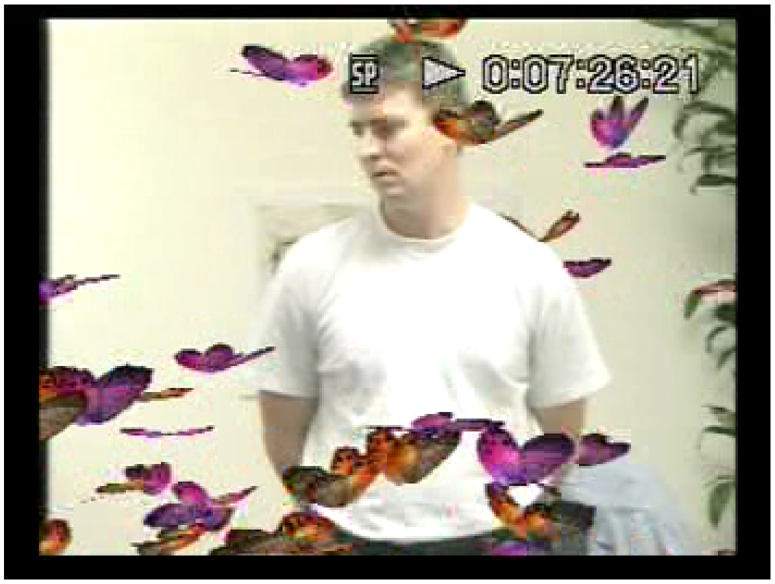


Figure 24: *3D pixel data allows real video to be mixed with virtual graphics with proper pixel occlusion. Some of the butterflies pass in front of the person, while others fly behind him.*

Inferring 3D information with a standard 2D camera is possible in scenes for which a model is known (such as EyeToy sphere and cylinder color tracking for PS2), but Z-sensing provides 3D information for completely unknown objects. This greatly extends the possibilities of augmented reality for merging virtual worlds with the real world. Because Z-sensing provides per-pixel depth information, the video image effectively becomes a Z-sprite that can be easily composited with computer graphics constructs using a standard z-buffer. The visual occlusion cues this z-compositing creates provides a much more compelling augmented reality experience than a simple graphical overlay (see Figure 24). Besides occlusion, Z-sensing also can enable other visual cues for augmented reality. Surfaces inferred from the Z data can be used as both shadow blockers and shadow receivers, allowing virtual objects to cast shadows on the scene, and to allow players to shadow virtual objects. In addition, these surfaces allow dynamic virtual 3D light sources to realistically re-light the real-world scene. By enabling these visual cues, Z-sensing helps bring the quality level of real-time augmented reality a few steps closer to movie special effects.

3.3.4 Real-time Motion Capture

One exciting use of spatial interfaces for games is using real-time motion capture to control a virtual character. This can be done by computing a dynamic virtual skeleton model that represents the player's motion, though this is difficult because human body has many degrees of freedom. Z-sensing helps to make the computation of this model more tractable by enabling reliable foreground/background separation for segmenting out the pixels that correspond to the player. Also, the per-pixel 3D data can be used for a fitting process to solve dynamically for the skeleton model (see Figure 25). As demonstrated at SIGGRAPH 2003, real-time character control can be achieved effectively using a model with very few degrees of freedom [Marks et al. 2003]. Recently, Project Natal from Microsoft has shown similar applications of real-time motion capture, with the promise of many new applications for the future.

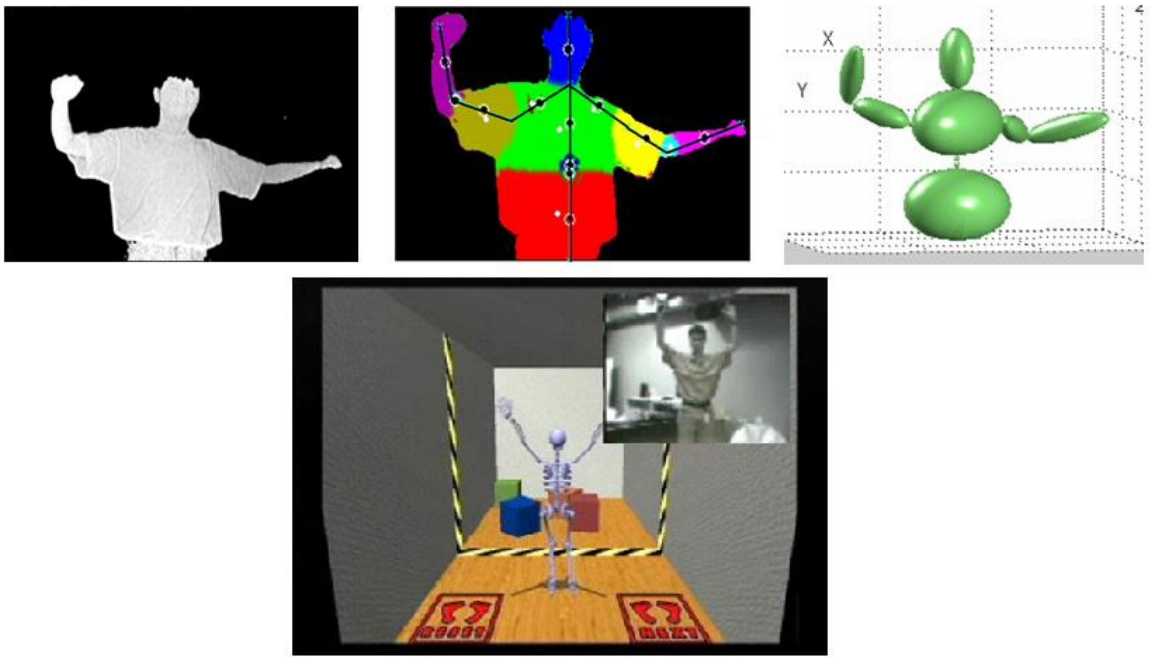


Figure 25: *A skeleton model is fit to data from a 3D camera to enable real-time motion capture.*

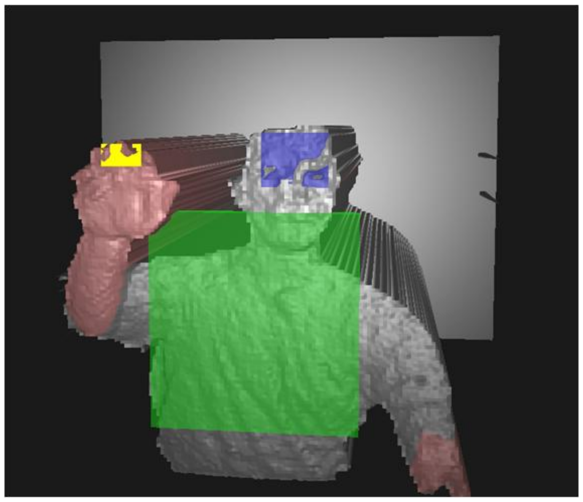


Figure 26: *Data from 3D camera used to implement a 3D mouse.*

Real-time motion capture can be used for other applications beyond controlling a virtual character. Since the positions of the head and hands are known, many of the applications described for color tracking and face tracking are also possible. For example, the location of the hand can be used as much like a 3D mouse (see Figure 26). In addition, the relative 3D positioning of body parts can also be used for spatial interfaces. For the mouse application, the player's hand can be ignored unless it is a certain distance in front of his torso. This effectively lets the player choose when to ”address” the application, thus solving the ”always-on” problem of camera interfaces, similar to how push-to-talk solves the issue for voice input. Other applications such as 3D multi-touch are enabled by using the position data for two hands simultaneously.

4 Working with the Nintendo Wiimote

As with the EyeToy and PlayStation Eye, the Nintendo Wii Remote (Wiimote) is another example of a spatially convenient device [Wingrave et al. 2010]. Spatially convenient devices involve three important components:

- Spatial data. The device provides 3D input data, be it partial, error-prone, or conditional.
- Functionality. The device packs a range of useful sensors, emitters, and interface implements.
- Commodity design. The device is inexpensive, durable, easily configurable, and robust.

Regarding spatial data, traditional 3D hardware trackers present information in 6 degrees of freedom (DOF) in a tracked space with relatively good precision. In contrast, a Wiimote presents three axes of acceleration data in no particular frame of reference, with intermittent optical sensing. (The Wii MotionPlus gyroscope attachment can add three axes of orientation change, or angular velocity.) Although this means the Wiimote's spatial data doesn't directly map to a real-world positions, the device can be employed effectively under constrained use [Shirai et al. 2007; Shiratori and Hodgins 2008]. Regarding functionality, traditional 3D hardware might come with a few buttons. The Wiimote incorporates several buttons, some in a gamepad and trigger configuration, and has a speaker, programmable LEDs, and a rumble device. Regarding commodity design, 3D hardware can require extensive installations and environment instrumentation and can be difficult to work with. The Wiimote is easy to set up, turn on, and maintain. Overall, the Wiimote incorporates many useful input and output features in an inexpensive, consumer-oriented, easy-to-replace, and easy to repurchase package. This lets game developers, researchers, and homebrew engineers use and modify it to best serve their needs.

4.1 Wiimote Basics

To use the Wiimote in 3DUIs, you need to know about

- connecting the Wiimote to the system,
- its frames of reference (FORs),
- the sensor bar connection (SBC),
- dealing with accelerometer and gyroscope data, and
- other design considerations.

For technical details and specs, see the WiiBrew Web site (www.wiibrew.org).

4.1.1 Connecting the Wiimote

The Wiimote connects to a Wii game console or computer wirelessly through Bluetooth. When you press both the 1 and 2 buttons simultaneously or press the red Sync button in the battery case, the Wiimote's LEDs blink, indicating it's in discovery mode. Make sure your computer has a Bluetooth adapter and proper drivers. Currently, each OS and library handles the connection process differently.

PCs. Here are the basic steps (for extra assistance, seek online help such as at www.brianpeek.com):

1. Open Bluetooth Devices in the Control Panel.

2. Under the Devices tab, click the Add button.
3. Put the Wiimote into discovery mode (continuously reenter this mode during this process as it times out).
4. Select Next, select the Wiimote to connect to, and select Next again.
5. Don't use a passkey, and select Next, then Finish.

If a failure occurs, repeat this process until the PC connects to the Wiimote. A simple start for a PC to retrieve data from a Wiimote is the WiimoteLib (www.wiimotelib.org).

Macs. The setup needs to run only once, and future Wiimote connections are simpler. To set up the Wiimote, run the Bluetooth Setup Assistant utility and follow these steps:

1. Select the Any Device option.
2. Put the Wiimote into discovery mode.
3. Select the device (Nintendo RVL-CNT-01) and set the passkey options to "Do not use a passkey."
4. Press Continue until the Mac is connected; then quit the Setup Assistant.

To connect in the future, in the Bluetooth section of System Preferences, select the Wiimote entry and set it to connect. Then, put the Wiimote into discovery mode, and the Wiimote will connect. The OS can keep some applications from connecting to the Wiimote. In those cases, disconnect the Wiimote and let the application connect to the Wiimote.

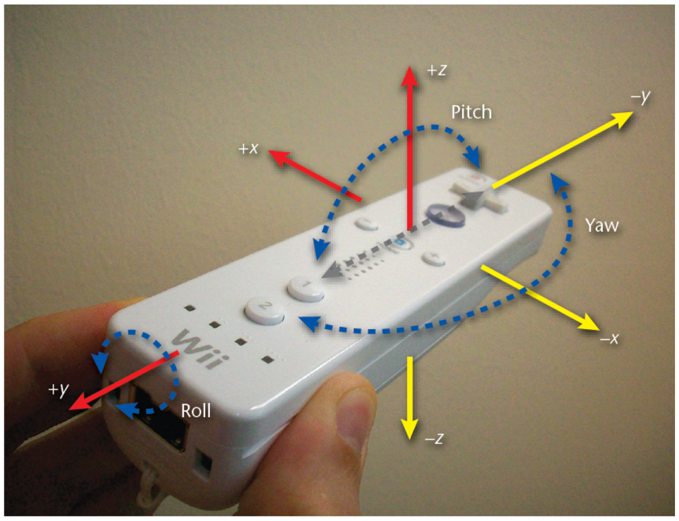


Figure 27: *The Wiimote, with labels indicating the Wiimote's coordinate system. Multiple coordinate systems and partial spatial data make the Wiimote difficult to design for.*

4.1.2 Frames of Reference

Figure 27 shows the Wiimote's FOR. The x, y and z axes are labeled, along with the rotation about each axis: pitch, roll, and yaw, respectively. A second FOR is the Earth's, important because the Wiimote's accelerometers detect the Earth's gravity. A third FOR is the Wiimote's relationship to the sensor bar. Three examples clarify how these FOR interact. To begin, a user is considered holding a Wiimote naturally when +z is up in both the Wiimote's and the Earth's FOR and the Wiimote's front points away from the user and toward a sensor bar, which is usually on top of the display. In the first example, the user moves the Wiimote toward the sensor bar. This results in acceleration reported in the y-axis of both the Earth and Wiimote FORs and the sensor bar reporting decreased distance between the Wiimote and it. In the second example, the user rotates the Wiimote down to point toward the earth (a 90 pitch). When the user again moves the Wiimote directly toward the sensor bar in front of him or her, the Wiimote reports acceleration in its z-axis. However, the Earth's FOR has acceleration in its y-axis because the 90 downward pitch didn't change the Earth's FOR. The sensor bar has no FOR because as the Wiimote rotates away from the sensor bar, the Wiimote loses contact with it. The third example has the same configuration as the second example, but with the sensor bar on the ground directly below the Wiimote, at the user's feet. When the user repeats that forward motion, the sensor bar reports the Wiimote moving up in the sensor bar's z-axis, whereas the Earth and Wiimote data are the same as in the previous example. These examples show that each FOR captures important and different information. The following sections explain this in more detail.

4.1.3 The Sensor Bar Connection

The SBC is one of the Wiimote's two primary spatial sensors. It occurs when the Wiimote's infrared optical camera points at a sensor bar and sees the infrared (IR) light emitted by its LEDs. A sensor bar has LEDs on each side (see Figure 28), with a known width between them. This produces IR blobs that the Wiimote tracks and reports in x and y coordinates, along with the blob's width in pixels. To improve tracking distance (the Wiimote can sense blobs up to 16 feet away), the sensor bar LEDs are spread in a slight arc, with the outer LEDs angled out and the inner LEDs angled in. Actually, any IR source will work, such as custom IR emitters, candles, or multiple sensor bars (provided you have the means to differentiate between the sensor bars).

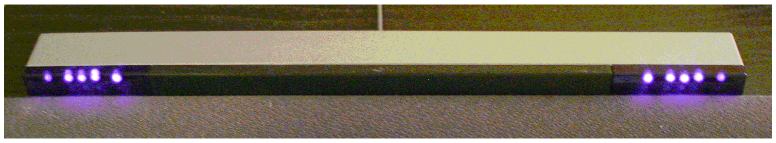


Figure 28: *The Wiimote sensor bar has two groups of IR LEDs at fixed widths.*

When the Wiimote is pointed at the sensor bar, it picks up two points $P_L = (x_L, y_L)$ and $P_R = (x_R, y_R)$ from the LED arrays. The midpoint between these P_L and P_R can easily be calculated and used as a 2D cursor or pointer on the display. In addition, if the Wiimote is rotated about the Y-axis, we can calculate the roll of the device with respect to the X-axis using

$$roll = \arccos\left(\mathbf{x} \cdot \frac{\mathbf{v}}{\|\mathbf{v}\|}\right) \tag{5}$$

where $\mathbf{x} = (1, 0)$ and $\mathbf{v} = P_L - P_R$.

Combining information from the sensor bar and the Wiimote's optical sensor also make it possible to determine how far the Wiimote is from the sensor bar using triangulation. The distance d between the sensor bar and Wiimote is calculated using

$$d = \frac{w/2}{\tan(\theta/2)} \tag{6}$$

$$w = \frac{m \cdot w_{img}}{m_{img}} \tag{7}$$

$$m_{img} = \sqrt{(x_L - x_R)^2 + (y_L - y_R)^2} \tag{8}$$

where θ is the optical sensor's viewing angle, m is the distance between the sensor bar's left and right LEDs, w_{img} is the width of the image taken from the optical sensor, and m_{img} is the distance between P_L and P_R taken from the optical sensor's image. Note that θ, w_{img}, and m are all constants. This calculation only works when the Wiimote device is pointing directly at the sensor bar (orthogonally). When the Wiimote is off-axis from the sensor bar, more information is needed to find depth. We can utilize the relative sizes of the points on the optical sensor's image to calculate depth in this case. To find d in this case, we compute the distances corresponding to the the left and right points on the optical sensor's image

$$d_L = \frac{w_L/2}{\tan(\theta/2)} \tag{9}$$

$$d_R = \frac{w_R/2}{\tan(\theta/2)} \tag{10}$$

using

$$w_L = \frac{w_{img} \cdot diam_{LED}}{diam_L} \tag{11}$$

$$w_R = \frac{w_{img} \cdot diam_{LED}}{diam_R} \tag{12}$$

where $diam_{LED}$ is the diameter of the actual LED marker from the sensor bar and $diam_L$ and $diam_R$ are the diameters of the points found on the optical sensor's image. With d_L and d_R, we can then calculate Wiimote's distance to the sensor bar as

$$d = \sqrt{d_L^2 + (m/2)^2 - 2d_L(m/2)^2 \cos(\phi)} \tag{13}$$

where

$$\cos(\phi) = \frac{d_L^2 m^2 - d_R^2}{2md_L}. \tag{14}$$

Note that with d, d_L, and m we can also find the angular position of the Wiimote with respect to the sensor bar, calculated as

$$\alpha = \arccos\left(\frac{d^2m^2 - d_L^2}{md_L}\right). \tag{15}$$

4.1.4 3-Axis Accelerometer

The second Wiimote input is the device's 3-axis accelerometer. The accelerometer reports acceleration data in the device's x, y and z directions, expressed conveniently in g's. This is common of many devices employing 3-axis accelerometers such as the cell phones like the iPhone, laptops and camcorders. With this information, the Wiimote is able to sense motion, reporting values that are a blend of accelerations exerted by the user and gravity. As the gravity vector is constantly oriented towards the Earth (or $(0, 0, 1)$ in Earth's FOR), the gravity vector can be used to discover part of the Wiimote's orientation in terms of earth's frame of reference using

$$pitch = \arctan\left(\frac{a_z}{a_y}\right) \tag{16}$$

$$roll = \arctan\left(\frac{a_z}{a_x}\right). \tag{17}$$

Unfortunately, determining yaw in the Earth's FOR isn't possible because the Earth's gravity vector aligns with its z-axis. Another unfortunate issue is that determining the actual acceleration of the Wiimote is problematic owing to the confounding gravity vector. To determine the actual acceleration, one of the following must take place:

- the Wiimote must be under no acceleration other than gravity so that you can accurately measure the gravity vector (in which case, you already know the actual acceleration is zero),
- you must make assumptions about the Wiimote's orientation, thus allowing room for errors, or
- you must determine the orientation by other means, such as by the SBC or a gyroscope (we discuss this in more detail later).

The implications for orientation tracking by the accelerometers are that the Wiimote's orientation is only certain when it is under no acceleration. For this reason, many Wii games require that users either hold the Wiimote steady for a short period of time before using it in a game trial or have it pointed at the screen and orient by the SBC.

4.1.5 Wii MotionPlus

This attachment uses two gyroscopes to report angular velocity along all three axes (one dual-axis gyro for x and y and a single-axis gyro for z). Mechanical gyroscopes would typically be too large and expensive for a Wiimote. So, it uses MEMS (microelectromechanical system) gyroscopes, which operate using a vibrating structure, are inexpensive, use little power, and are fairly accurate. A MotionPlus-augmented Wiimote provides information on changes to the Wiimote's orientation, alleviating many of the device's data limitations. With this, Nintendo is attempting to improve the orientation accuracy of the device.

The MotionPlus isn't yet fully reverse engineered. It reports orientation changes in two granularities, fast and slow, with fast being roughly four times the rate per bit. The gyroscope manufacturer reports that the two gyroscopes have a linear gain but that the different gyroscopes report values in two different scales, so there's no single scaling factor. Additionally, temperature and pressure changes can impact this scale factor and change the value associated with zero orientation change.

Merging the acceleration and gyroscopic data isn't simple; both sensors have accuracy and drift errors that, albeit small, amount to large errors over short time periods. When using the SBC, you can compensate for these accumulating errors by providing an absolute orientation and position. Researchers have improved orientation by merging accelerometer and gyroscopic data but didn't test a system under translational motion [Luinge et al. 1999]. Other research has shown that you can combine accelerometers and gyroscopes for accurate position and orientation tracking [Williamson and Andrews 2001]. In addition, researchers have successfully used Kalman filters to merge accelerometer and gyroscopic data [Azuma and Bishop 1994; Williamson et al. 2010]

4.2 Design Considerations

Because of the Wiimote's many limitations, you must take into account two design considerations.

4.2.1 Waggling Motions

”Cheating” motions (or less fatiguing and comfortable movements, depending on your perspective) are a side effect of Wiimote input limitations. Game designers might intend for games to encourage exercise, breaking the stereotype of the lazy video game player. However, the Wiimote's inability to detect actual position change lets users make only small or limited ”waggling” motions, which the Wiimote interprets as full movement. The result is boxing games played by tapping the Wiimote, tennis games played with wrist flicks, and many games winnable by simply moving the device randomly. Although this is still fun for gamers, it limits the Wiimote's utility for 3DUIs and exercise and health gaming, unless you employ better hardware and data-interpretation methods.

4.2.2 Compensation by ”Story”

Whenever possible, the easiest means of compensating for input hardware limitations is through the use of story. By ”use of story,” we mean that by careful manipulation of the users' tasks and their goals, the shortcomings of the hardware can be avoided. This is common in Wii gaming, in which accuracy is second to enjoyment and playability. As a first example, consider the inherent drift in Wiimotes. You can compensate for the drift by requiring the user to return to a known position from which you can assume the Wiimote's orientation, or you can create an SBC by requiring the user to point at an on-screen button to begin a task.

As a second example, story can engage users, instructing them to freeze their hand at a known orientation. From this orientation assumption, the gravity vector can be assumed and actual acceleration calculated. Several examples come from games. The We Cheer game instructs players to hold the Wiimote as if they're holding pom-poms, WarioWare tells players how to hold the Wiimote for each minigame, and Wii Sports Boxing makes players raise their hands when preparing for a fight. Researchers commonly guide participants by story, instructing them to enter start positions before a trial. For example, in studies of 3DUI selection techniques, researchers often have users select an object to begin a trial.

4.3 Interpreting Wiimote Data

Here we review two basic types of Wiimote data interpretation, integration and recognition through heuristics. Note we also discuss traditional gesture recognition in the last section of these course notes.

4.3.1 Integration

Because acceleration is the rate of velocity change, which is the rate of positional change, you can theoretically integrate the Wiimote acceleration data to find velocity and reintegrate it to find the Wiimote's actual position change. You should also be able to integrate the gyroscope's angular velocity to achieve absolute orientation. However, this approach has three significant limitations [Giansanti et al. 2003]. First, acceleration jitter can lead to significant positional deviations during a calculation. So, you can ignore accelerations below a threshold because they're most likely the result of jitter. Alternatively, a smoothing function can help reduce jitter. Second, you must remove the gravity vector from the reported acceleration before computing velocity. If the SBC or a gyroscope is available, you can possibly infer the Wiimote's orientation and realize the gravity vector as a 1-g downward force; otherwise, you must determine the gravity vector on the fly. One way to compute the gravity vector is by watching the derivative of the acceleration (or jerk data). When this is close to zero (that is, no acceleration change is computed) and the reported acceleration magnitude is close to 1 g, thereby eliminating cases of user-induced constant acceleration, you can assume the Wiimote is reporting only the gravity vector. Subtracting this vector from future accelerations results in the acceleration being directly attributable to user action, assuming the Wiimote maintains its orientation. Third, orientation must be accurate; even slight errors produce large positional errors after movement. For example, one meter of travel after a five-degree orientation change (well within the variance of the hand) results in an error of nearly 9 cm. A larger 30-degree orientation change results in an error of 50 cm. Additionally, unaccounted-for orientation changes can give very wrong results. For example, raising the Wiimote a foot, inverting it, and lowering it to its original location will result in no actual positional change, but as computed will result in a two-foot upward movement in the Wiimote's FOR. You can address this only by using the SBC or a gyroscope, because the Wiimote's accelerometers can't easily provide a gravity vector while the Wiimote is moving, as we discussed earlier.

In work with a locomotion interface for American football [Williamson et al. 2010], double integration let users control the application using a Wiimote and have their body movements maneuver the quarterback. To achieve this, a Wiimote was attached to the center of the user's chest, which was close to the body's center of mass. Although this improved the reported acceleration data, it broke the SBC. The Wiimote acceleration data was then passed through an exponential smoothing filter:

$$\mathbf{a}_{current} = \alpha\mathbf{a}_i + (1-\alpha)\mathbf{a}_{i-1} \tag{18}$$

where $\alpha = 0.9$. An alternate approach is to use a Kalman filter, but this requires more computation [LaViola 2003]. Finally the double-integration step was performed on the smoothed acceleration data. Owing to these three steps, the Wiimote was responsive, and the user seemed to move relatively accurately for that application.

To further evaluate the control's accuracy, tests were performed in which the user moved from a starting location and then back. These tests showed little error in position over short time periods (5 to 10 seconds), but only when users maneuvered in an unnaturally upright and stiff fashion.

4.3.2 Recognition through Heuristics

The Wiimote provides a raw data stream, and you can use heuristics to interpret and classify the data. Whether you use heuristics on their own or as features for gesture recognition (which we discuss later), their higher-level meaning is more useful for 3DUI design and implementation. In this section, we discuss several heuristic-based approaches for interpreting Wiimote data used in two prototype game applications. One Man Band used a Wiimote to simulate the movements necessary to control the rhythm and pitch of several musical instruments [Bott et al. 2009]. Careful attention to the Wiimote data enabled seamless transitions between all musical instruments. RealDance explored spatial 3D interaction for dance-based gaming and instruction [Charbonneau et al. 2009]. By wearing Wiimotes on the wrists and ankles, players followed an on-screen avatar's choreography and had their movements evaluated on the basis of correctness and timing.

Poses and underway intervals. A pose is a length of time during which the Wiimote isn't changing position. Poses can be useful for identifying held positions in dance, during games, or possibly even in yoga. An underway interval is a length of time during which the Wiimote is moving but not accelerating. Underway intervals can help identify smooth movements and differentiate between, say, strumming on a guitar and beating on a drum.

Because neither poses nor underway intervals have an acceleration component, you can't differentiate them by accelerometer data alone. To differentiate the two, an SBC can provide an FOR to identify whether the Wiimote has velocity. Alternatively, you can use context, tracking Wiimote accelerations over time to gauge whether the device is moving or stopped. This approach can be error prone, but you can successfully use it until you reestablish the SBC.

Poses and underway intervals have three components. First, the time span is the duration in which the user maintains a pose or an underway interval. Second, the gravity vector's orientation helps verify that the user is holding the Wiimote at the intended orientation. Of course, unless you use an SBC or a gyroscope, the Wiimote's yaw won't be reliably comparable. Third, the allowed variance is the threshold value for the amount of Wiimote acceleration allowed in the heuristic before rejecting the pose or underway interval.

In RealDance, poses were important for recognizing certain dance movements. For a pose, the user was supposed to stand still in a specific posture beginning at time t_0 and lasting until $t_0 + N$, where N is a specified number of beats. So, a player's score could be represented as the percentage of the time interval during which the user successfully maintained the correct posture.

Impulse motions. An impulse motion is characterized by a rapid change in acceleration, easily measured by the Wiimote's accelerometers. A good example is a tennis or golf club swing in which the Wiimote motion accelerates through an arc or a punching motion, which contains a unidirectional acceleration.

An impulse motion has two components, which designers can tune for their use. First, the time span of the impulse motion specifies the window over which the impulse is occurring. Shorter time spans increase the interaction speed, but larger time spans are more easily separable from background jitter. The second component is the maximum magnitude reached. This is the acceleration bound that must be reached during the time span in order for the Wiimote to recognize the impulse motion.

You can also characterize impulse motions by their direction. The acceleration into a punch is basically a straight impulse motion, a tennis swing has an angular acceleration component, and a golf swing has both angular acceleration and even increasing acceleration during the follow-through when the elbow bends.

All three of these impulse motions, however, are indistinguishable to the Wiimote, which doesn't easily sense these orientation changes. For example, the punch has an acceleration vector along a single axis, as does the tennis swing as it roughly changes its orientation as the swing progresses. You can differentiate the motions only by using an SBC or a gyroscope or by assuming that the Wiimote orientation doesn't change.

RealDance used impulse motions to identify punches. A punch was characterized by a rapid deceleration occurring when the arm was fully extended. In a rhythm game, this instant should line up with a strong beat in the music. An impulse motion was scored by considering a one-beat interval centered on the expected beat. For the Wiimote corresponding to the relevant limb, the time sample in the time span corresponding to the maximal acceleration in the Wiimote's longitudinal axis was selected. If this maximal acceleration was below a threshold, no punch occurred, and the score was zero. Otherwise, the score was computed from the distance to the expected beat. If the gesture involved multiple limbs, the maximal acceleration value had to be greater than the threshold for all Wiimotes involved. The average of all individual limb scores served as the gesture's overall score.

Impact events. An impact event is an immediate halt to the Wiimote due to a collision, characterized by an easily identifiable acceleration bursting across all three dimensions. Examples of this event include the user tapping the Wiimote on a table or a dropped Wiimote hitting the floor. To identify an impact event, compute the change in acceleration (jerk) vectors for each pair of adjacent time samples. Here, t_k corresponds to the largest magnitude of jerk:

$$t_k = \underset{T}{\operatorname{argmax}} \|\mathbf{a}_t - \mathbf{a}_{t-1}\|. \tag{19}$$

If the magnitude is larger than a threshold value, an impact has occurred. RealDance used impact motions to identify stomps. If the interval surrounding a dance move had a maximal jerk value less than a threshold, no impact occurred, and the score was zero. Otherwise, the score was calculated the same way as for an impulse motion. One Man Band also used impact events to identify when a Nintendo Nunchuk controller and Wiimote collided, which is how users played hand cymbals.

Modal differentiation. You can use easily recognized modes in an interface to differentiate between functionality. For example, a button's semantics can change as the Wiimote changes orientation, or the Wiimote's pitch might differentiate between a system's states. Although modes can lead to errors when users are unaware of them, constant user action such as holding a button or pose can lead to quasimodal states that are easily understood and useful. This is important because the Wiimote has several attachments, such as the Nunchuk, and multiple buttons that you can use to create quasimodes. In One Man Band, the multi-instrument musical interface (MIMI) differentiated between five different instruments by implementing modal differences based on the Wiimote's orientation. Figure 29 shows four of these. If the user held the Wiimote on its side and to the left, as if playing a guitar, the application interpreted impulse motions as strumming motions. If the user held the Wiimote to the left, as if playing a violin, the application interpreted the impulse motions as violin sounds. To achieve this, the MIMI's modal-differentiation approach used a normalization step on the accelerometer data to identify the most prominent orientation:

$$\mathbf{a}_{norm} = \frac{\mathbf{a}}{\|\mathbf{a}\|} \tag{20}$$

followed by two exponential smoothing functions (see Equation 18). The first function, with an $\alpha = 0.1$, removed jitter and identified drumming and strumming motions. The second function, with an $\alpha = 0.5$, removed jitter and identified short, sharp gestures such as violin strokes. The MIMI also used the Nunchuk's thumbstick to differentiate between the bass and guitar.

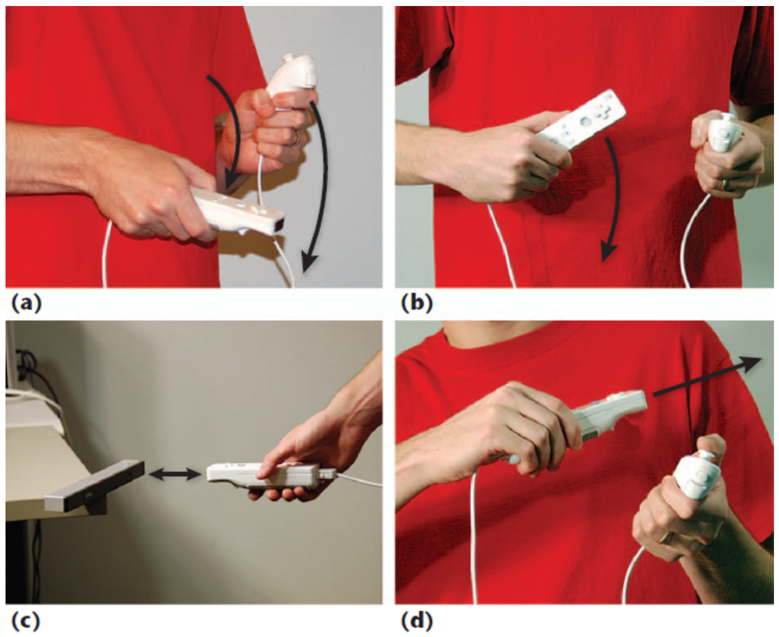


Figure 29: *One Man Band differentiated between multiple Wiimote gestures using mostly simple modal differentiations for (a) drums, (b) guitar, (c) violin, and (d) theremin. To the player, changing instruments only required orienting the Wiimote to match how an instrument would be played.*

4.4 Designing for Universal 3D Interaction

Although the Wiimote has limitations for universal 3D interaction tasks, no single limitation completely prevents its use in a 3DUI. On the basis of this, we've developed the following guidelines.

4.4.1 Selection

This task-choosing one or more objects-is a useful starting point for manipulation tasks or for interacting with buttons or other widgets. Two basic egocentric metaphors exist for selection. Virtual hand techniques involve reaching to grab objects, with a correspondence between the virtual hand and the user's hand. These techniques are useful for differentiating between occluding objects and near and far objects during selection. Virtual pointer techniques involve defining a vector along which the user selects intersecting or nearly intersecting objects. These techniques require less arm movement and can be faster than virtual-hand techniques.

Because virtual-hand techniques require depth data, the Wiimote's ability to specify depth relates directly to its effectiveness. This is critical because Wiimote depth data is error prone, even under optimal conditions when an SBC exists. Unfortunately, SBCs aren't always maintained because users often hold a Wiimote in a power grip pointing up (like a hammer). So, pointing at the sensor bar requires effort and effectively operates as a virtual-pointing technique anyway.

3DUI techniques can help you design around these limitations. Originally, virtual-pointer selection employed reeling to reel objects forward or backward along the selection vector.4 Therefore, you can also effectively use reeling to reel the virtual hand along its orientation vector. Discrete buttons or incorporating the user's reaching (that is, extending and pulling in the arm control reeling) can achieve this. You could even steer the virtual hand around the scene by mapping the Wiimote's offset vector from a predefined position to the virtual hand's velocity vector. In this way, Wiimote depth errors only minimally impact the virtual hand's control. Both reeling and steering limit the jitter effect because velocity jitter is smoother for control than positional jitter. For virtual-pointer techniques, the Wiimote alone can't define a pointing vector without an SBC or a gyroscope. This is because you can't determine yaw from accelerometer data alone. Fortunately, although typically it's difficult to assume an SBC exists, a selection task by definition has the user pointing at something on the screen. So, the SBC assumption holds in a natural way. An exception to this would be head-mounted displays (HMDs), in which the display is attached to the user and not the environment. However, most commodity setups don't use HMDs. You can use gyroscopes to generate a yaw vector, but their inherent drift requires constant realignment, which again requires an SBC.

Another option enables the Wiimote alone to specify a partial pointing vector. Yaw is vital to specifying the selection vector. However, if the user orients and holds the Wiimote such that roll and not yaw is lost (that is, pointing it up or down), virtual pointing becomes possible with yaw and pitch. This is because roll about the selection vector is lost because the selection vector is a ray (although manipulation after selection might be affected). To achieve this partial pointing vector, you can use story to affect the user's Wiimote grip.

In summary:

- Virtual-hand techniques can have difficulties because specifying depth is error-prone. Improving their depth-specifying capabilities can improve their effectiveness.
- Virtual pointing can be effective when the Wiimote remains pointed at the sensor bar or you can cajole the user into holding the Wiimote in a different orientation.
- Unless objects occlude or exist at many depths, you should use virtual pointing.

4.4.2 Manipulation

This task changes a selected object's position and orientation. It commonly occurs after a successful selection, implying that the object is being manipulated as if it were attached to a ray or virtual hand. Both attachment types rely on the specification of accurate depth information, which is difficult with an SBC. Doubly integrated position is a possibility, but orientation changes greatly impact its accuracy, unless assisted by an SBC or a gyroscope. In either case, problems will occur that require compensating design.

Manipulation control can be considered in terms of its position and orientation components. For position manipulation of objects attached to a ray, reeling is applicable. For position manipulation of objects attached to a virtual hand, the steerable virtual hand is applicable. Both manipulations are nonisomorphic and result in a loss of naturalness. For controlling just orientation manipulation, a Wiimote is adequate, but most manipulation tasks pair position and orientation manipulation. A Wiimote with a gyroscope can sufficiently manipulate orientation. However, orienting the Wiimote can lose the SBC, which is vital for position manipulation. To avoid losing the SBC, you can use nonisomorphic rotations. A rate-based approach can map Wiimote orientations to angular rotations. Or, you can use alternate rotational mappings-such as those for mapping 2D mouse movements to 3D orientation manipulation-to map smaller orientation changes to larger ones.

In summary:

- Isomorphic manipulation is problematic, especially for orientation specification.
- Many nonisomorphic techniques hold great potential for both position and orientation specification.

4.4.3 Travel

This task physically changes the user's position and orientation. You can achieve travel in two main ways. The first is by declarative means-that is, selecting a position to travel to. The second is by directional means-that is, indicating a velocity vector to travel along. Whereas you can achieve declarative travel by using a selection technique, directional travel requires the problematic specification of a vector to travel along. So, travel in 3D environments by Wiimote is limited in ways similar to selection and manipulation using a Wiimote. Both types of travel require

1. selecting position and/or orientation,
2. selecting velocity or acceleration, and
3. indicating when to start, continue, and end travel.

Position and orientation selection can employ virtual-hand techniques. Velocity and acceleration selection can be a static value chosen by you or dynamically by the application or the user. Examples of the latter include having the user's reach or the Wiimote's pitch indicate velocity. Indicating when to start, continue, and end travel is typically achieved by discrete events such as pushing a button, but you can also use voice commands or clearly definable modal differentiations (for example, extending the Wiimote begins travel).

You can produce a travel vector in several ways. Although the Wiimote has enough buttons to control travel in a 3D space, this would be limiting and would feel unnatural. However, the discrete buttons could also produce precise, constrained movements if the task requires. Otherwise, the typical means of specifying a travel vector is to point in the direction of travel. With Wiimotes, this is hindered by the inability to determine yaw by accelerometers alone. Even with an SBC, there's a limitation of 45 degrees. However, as we mentioned before, a gyroscope enables the Wiimote to report yaw information. Pairing one with an SBC to deal with gyroscope drift will create a fairly robust travel vector. Alternatively, a position offset of the Wiimote from a predefined point can indicate a 3D vector useful for travel. For example, the user can select an arbitrary point by pressing and holding a button. Any displacement from that point can be detected by the SBC or as impulse motions. This creates a vector that you can map to a velocity. This design leaves the Wiimote's orientation information available.

Several compensating design alternatives exist for when an SBC isn't available. For many 3D environments, travel is constrained to the environment's 2D surface, so only a 2D vector is required. In this case, you could easily map the Wiimote orientation's 2D roll and pitch to a 2D travel vector. Alternatively, you could use story to compel the user to hold the Wiimote upright for two possible uses:

- making roll and pitch appear as yaw and thrust and
- a virtual analog joystick.

You can extend these 2D vector specification designs to 3D travel. To travel off the 2D plane, you can map Wiimote buttons or raising and lowering the Wiimote to a magnitude vector. Raising and lowering are identifiable by impulse motions. However, this is limited by the Wiimote's inability to detect rotation and pitch while moving, unless the orientation is held constant, a gyroscope is attached, or an SBC is used.

Orientations could easily break the SBC, but these cases tend to be rare owing to the constraining range of comfortable wrist movements. Although these compensating designs are sufficient for travel, they limit the ability to control orientation during travel.

In summary:

- With a gyroscope and an SBC, a well-configured travel technique can practically eliminate the Wiimote's limitations for travel tasks.
- If travel is required only along a 2D plane, the Wiimote alone can be sufficient.

4.4.4 System Control and Symbolic Input

These tasks inject command-based and discrete events into an application. In 2D interfaces, the keyboard and WIMP (windows, icons, menus, pointing devices) metaphor are the dominant means of generating this input. However, neither is optimally suited for the Wiimote. First, typing can be difficult while holding a Wiimote. Second, 3D interfaces let users move, so users aren't necessarily within reach of a keyboard. Third, mouse emulation with a Wiimote is possible, but even with filtering to remove jitter, the emulation lacks the precision users can achieve by manipulating a mouse with their fingertips. If you use WIMP interfaces with Wiimotes, larger on-screen elements should be used.

You can improve your Wiimote system control and symbolic input in four additional ways. First, non-isomorphic mapping of the Wiimote's orientation and position can result in more precise control of the Wiimote's cursor. Alternatively, using the Wiimote's position to control its cursor can improve accuracy but can quickly lead to fatigue. Second, identifying a small set of Wiimote gestures can lead to a simple vocabulary for system control. Keeping the gesture set small improves user recall and improves gesture recognition. Third, the Wiimote can provide feedback in the form of LEDs and sound. Incorporating better feedback with on-screen widgets could improve user interaction. Finally, low-frequency movements tend to indicate narrowing-in on a target during selection. Adaptively varying the Wiimote's sensitivity can improve the user's accuracy yet still allow large movements.

Designing a Wiimote keyboard is also a possibility. Symbolic input of an alphabet with the Wiimote has often relied on reproducing a virtual qwerty keyboard. Although these keyboards are familiar to users, they aren't optimally designed for all uses. Alternatives include using rotation to select symbols or devising keyboard layouts better suited to the Wiimote. Chorded interaction can also improve symbolic input. Such interaction relies on the product of separate measures to create many usable symbols. For example, you could differentiate the Wiimote's orientation into four states: forward, back, left, and right. With just two buttons (1 and 2), which produce three button-press states (1, 2, and 1 & 2), chorded interaction can produce 12 distinct symbols (through four orientations and the three states). Users' ability to recall all these chorded combinations is another matter.

Overall, system-control and symbolic-input tasks for the Wiimote are similar to those for existing 3D hardware. They require special attention to the task and the equipment ergonomics. So, existing design guidelines are directly applicable, and a few become especially important:

- Understand the wrist's limitations for comfortable rotation.
- The Wiimote can be prone to jitter, especially as the distance from the screen increases.
- Chorded interaction can lead to many symbols, but without assistance, whether mnemonics or visual representations, recall quickly becomes the limiting factor.

4.5 Case Studies

The Wiimote has been used in a variety of different ways. Of course, Johnny Chung Lee has done a variety of interesting projects and is probably the most famous Wiimote hacker [Lee 2008]. However, there are many other researchers who are using Wiimotes for research in spatial 3D interfaces. Typically, there are two main approaches people use. The first is to actually wear the sensor bar and mount the Wiimote in some stationary position, using it as a camera. The second approach is to simply hold the Wiimote or attach it to the body. This approach makes more use of the accelerometer data than the image sensor data. In this section, we highlight some interesting research that focuses on using Wiimotes to create better gameplay experiences.

4.5.1 RealNav

Figure 30: *A user interacting with RealNav.*

RealNav (see Figure 30) is a reality-based locomotion study directly applicable to video game interfaces; specifically, locomotion control of the quarterback in American football [Williamson et al. 2010]. Focusing on American football drives requirements and ecologically grounds the interface tasks of: running down the field, maneuvering in a small area, and evasive gestures such as spinning, jumping, and the ”juke.”

The locomotion interface is constructed by exploring data interpretation methods on two commodity hardware configurations. The choices represent a comparison between hardware available to video game designers, trading off traditional 3D interface data for greater hardware availability. Configuration one matches traditional 3D interface data, with a commodity head tracker and leg accelerometers for running in place. Configuration two uses a spatially convenient device with a single accelerometer and infrared camera. Data interpretation methods on configuration two use two elementary approaches and a third hybrid approach, making use of the disparate and intermittent input data combined with a Kalman filter. Methods incorporating gyroscopic data are used to further improve the interpretation.

4.5.2 One Man Band

One Man Band (see Figure 31) is a prototype video game for musical expression that uses novel 3D spatial interaction techniques using accelerometer-based motion controllers [Bott et al. 2009]. One Man Band

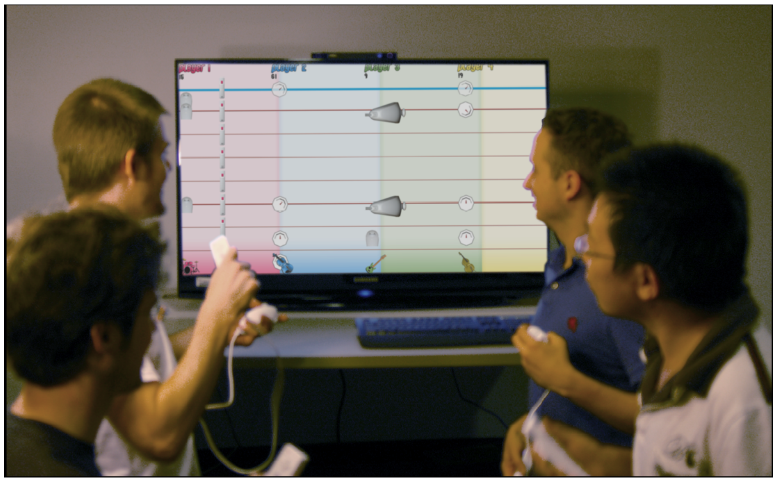


Figure 31: *Users playing One Man Band.*

provides users with 3D gestural interfaces to control both the timing and sound of the music played, with both single and collaborative player modes. The current system supports the guitar, violin, bass, drums, trombone, and therein. It also has a multi-instrument musical interface called the MIMI. The idea behind the MIMI is that game players might want to quickly and easily transition from one musical instrument to another without any mode switching. The MIMI uses heuristics recognition and exponential smoothing to detect 5 different instruments. A study was recently conducted comparing One Man Band and Wii Music and the results showed users significantly preferred One Man Band.

4.5.3 RealDance

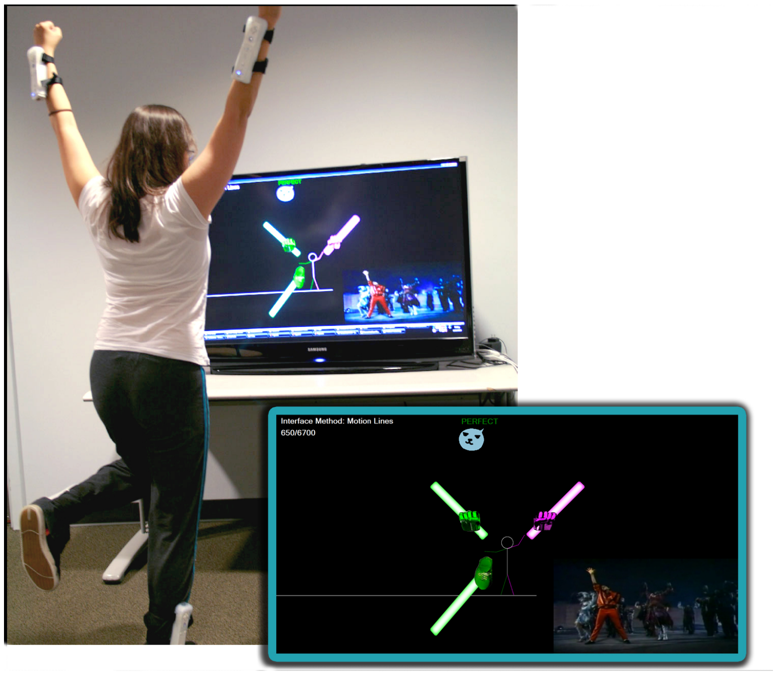


Figure 32: *A user playing RealDance.*

Real Dance (see Figure 32) is a game prototype that is exploring more natural, full body interfaces for dancing games [Charbonneau et al. 2009]. In the game, users wear four Wiimotes attached to their wrists and ankles using velcro strips. This provides an untethered experience, meaning that the user does not need to stand in one place or position. Another goal of Real Dance is to explore how to increase the number of recognizable dance movements. The current prototype detects kicks, stomps, punches, and static poses. It also employs a visual interface for teaching users to perform the various dances. The visual interface provides either a timeline with icons, motion lines, or beat circles as well as score feedback with avatars and an animated instructor figure.

4.5.4 WoW Navigation

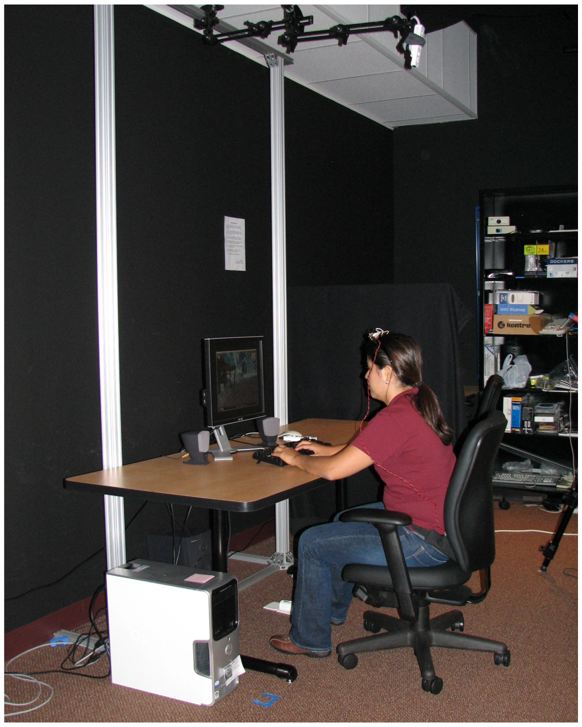


Figure 33: *A user playing World of Warcraft using body-based controls to navigate.*

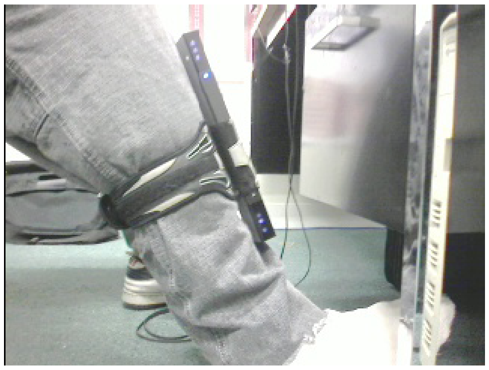


Figure 34: *A user wears the sensor bar in Wiisoccer.*

World of Warcraft has a complex interface and work is being done to determine how body-based interaction can be used to complement and reduce this complexity [Silva and Bowman 2009]. The idea is to using body based controls to offload keyboard and mouse navigation, helping players to concentrate on other tasks (see Figure 33). In this configuration, the user wears a modified sensor bar and mounts a Wiimote on the ceiling, using the device as a camera. Navigation is based on a leaning metaphor. Starting from a neutral body position, a small amount of forward or backward movement makes the character walk, and leaning farther forward makes the character run (rate control). There is a dead zone surrounding the neutral point in which the character stands still. Leaning to the side rotates the character, with the amount of rotation proportional to the distance the player leaned (position control). Note that a foot pedal is used to activate and deactivate movement. Preliminary experimentation has shown that body-based interaction in addition to keyboard and mouse can help players perform more tasks at the same time and can be especially attractive and helpful to new players.

4.5.5 Wiisoccer

The focus of this project, developed at Brown University under the direction of Chad Jenkins, is on exergaming, an important and up-and-coming research area that examines how to build effective exercise-based games. Wiisoccer is a game that uses natural locomotion to move players on a soccer field. The key innovation is that the IR sensor bar is attached to the users leg (see Figure 34) and a Wiimote is used as a camera and placed on the side of the player. The Wiimote detects the players running motion as well as kicks and passes.

5 3D Spatial Interaction with the PlayStation Move

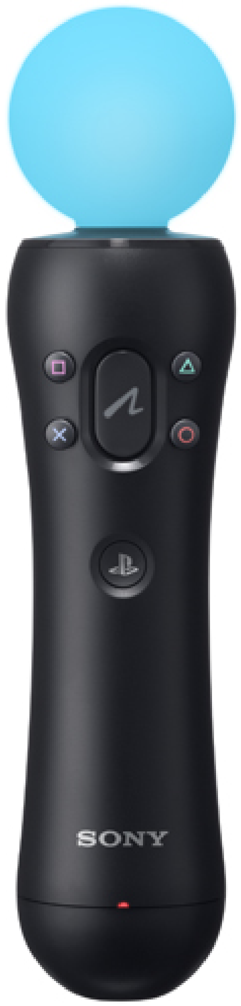


Figure 35: *The PlayStation Move motion controller.*

The PlayStation Move is a new controller for PlayStation 3 that will become available autumn of 2010. It is not meant as a replacement for the standard DualShock3 controller, but rather as an alternative that will enable new game experiences. The PlayStation Move design addresses many of the issues discovered during EyeToy development, combining the advantages of a camera-based interface with those of motion sensing and buttons. The following sections describe the PlayStation Move system and how it can be used to provide 3D input to games.

5.1 PlayStation Move Characteristics

The PlayStation Move system consists of a PlayStation Eye, one to four PlayStation Move motion controllers, and a PS3 SDK. It combines the advantages of camera tracking and motion sensing with the reliability and versatility of buttons. The wireless controller is used with one hand, and it has several digital buttons on the front and a long-throw analog T button on the back. Internally, it has several MEMS inertial sensors, including a three-axis gyroscope and three-axis accelerometer. But the most distinctive feature of the controller is the 44mm-diameter sphere on the top which houses a RGB LED that games can set to any color. The sphere color can be varied to enhance game play, but the primary purpose of the sphere is to enable reliable recovery of the controller 3D position using color tracking with the PlayStation Eye.

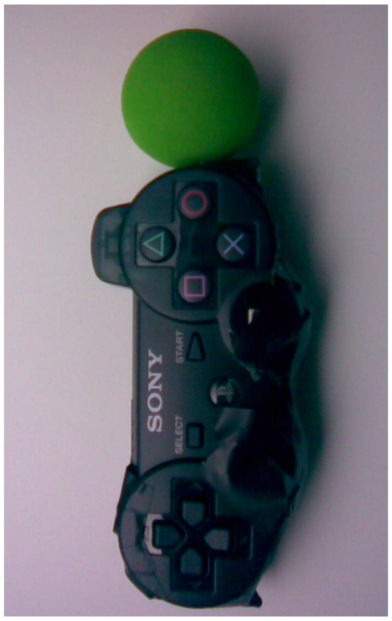


Figure 36: *Very early PlayStation Move prototype, based on a ”modified” SIXAXIS controller.*

The illuminated sphere design solves many of the color tracking issues experienced with EyeToy. Because the sphere generates its own light for the camera to see, scene lighting is much less of an issue. Tracking works perfectly in a completely dark room, and for scenes with highly varied lighting, the generated light mitigates the variability. Also, the light color can be adjusted to ensure it is visually unique with respect to the rest of the scene. Finally, the choice of the sphere shape makes the color tracking invariant to rotation; this simplifies position recovery and improves position precision by allowing a strong model to be fit to the projection.

Though the PlayStation Move hardware and the algorithms for combining the disparate sensor data are quite complex, the SDK provides a simple interface for PS3 developers. The high-level state for each controller state can be queried at any time, and consists of the following information:

- 3D position
- 3D orientation (as a quaternion)
- 3D velocity
- 3D angular velocity
- 3D acceleration
- 3D angular acceleration
- Buttons status
- Tracking status (e.g. visible)

Thus, a virtual object can be rendered that moves in one-to-one correspondence with a controller by using only the 3D position and orientation (the 3D pose). This allows game developers to focus on using the data effectively rather than struggle just to obtain the data. For completeness, the scaled and bias-corrected sensor values can also be queried directly, though using the high-level data is recommended instead.

5.2 PlayStation Move Technology

To use PlayStation Move for a game does not require understanding its underlying technology, because the high-level SDK abstracts such low-level details. However, understanding how the high-level state is derived from the various sensing components is helpful for understanding any potential tracking failure situations. Deriving the state involves two major steps: image analysis and sensor fusion. Though the exact details of these steps are beyond the scope of these notes, the following overview provides a qualitative understanding of each step.

5.2.1 Image Analysis

Conceptually, the image analysis can be broken up into two stages: finding the sphere in the image and then fitting a shape model to the sphere projection. Color segmentation is used to find the sphere, much like described previously for EyeToy color tracking. The exact color thresholds used for segmentation are based on results from initial calibration of the sphere LED brightness and room lighting which is performed at the start of play (by simply pointing the controller at the camera and pressing a button). The approximate size and location in the image are derived from the area and centroid of the segmented pixels.

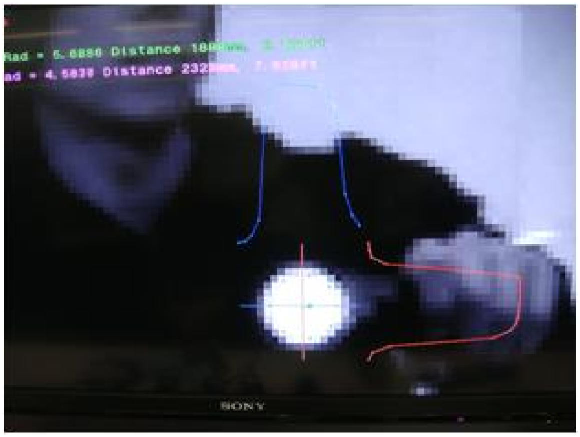


Figure 37: *Visualizing the 2D pixel intensity distribution of the PlayStation Move sphere is useful for debugging the model fitting stage.*

This size and location in the image are used as a starting point for fitting the shape model. To achieve the sub-pixel precision necessary for acceptable 3D position recovery, the shape model is fit to the original image RGB data rather than only the segmented pixels and accounts for camera lens blur. It is well-known that the 2D perspective projection of a sphere is an ellipse [Shivaram and Seetharaman 1998], though many tracking systems introduce significant error by approximating the projection as a circle. In theory, fitting such a model to the image data is straightforward, but in practice many issues arise. Motion of the sphere causes motion blur in the image proportional to the exposure time. Motion also causes the sphere to appear warped due to rolling shutter effects; like most low-cost CMOS webcams, PlayStation Eye uses a rolling electronic shutter. This means every line in the video is imaged at a slightly different time, which leads to stretching or shortening for vertical motions and shear for horizontal motions. Subtle artifacts can also be introduced depending on the method used to convert the raw sensor data to RGB data. Like most low-cost cameras, the PlayStation Eye sensor uses a Bayer pattern such that only R,G, or B is actually measured at every pixel, and RGB must be inferred from surrounding pixels (a process known as Bayer-pattern demosaicking, for which many approaches exist [Gunturk et al. 2005]). Another major concern is partial occlusion, which generally causes large errors in a global fit. To address this, a 2D ray-casting approach is used to identify areas of partial occlusion and remove them from the fit.

Because the size of the sphere and the focal length of the PlayStation Eye are known, the 3D position of the sphere relative to the camera can be computed directly from the 2D ellipse parameters. The output of the image analysis stage is a timestamp and a measurement of the 3D camera-relative position for each sphere, updated at the framerate of the camera (typically 60Hz).

5.2.2 Sensor Fusion

The results of the image analysis are combined with the inertial sensor data using a modified unscented Kalman filter. The details of this powerful state estimation technique are beyond the scope of these notes, but there are many excellent explanations available [Crassidis and Markley 2003][Julier and Uhlmann 1997][Wan and Van Der Merwe 2002]. Though the sensors all contribute to the final state in a complex manner, each has a fundamental contribution that is necessary for the complete state computation. For example, the camera tracking provides an absolute measure of the 3D position. When the controller is not moving, the accelerometer provides the direction of gravity, which gives an absolute measure of the pitch and roll angles. In addition, when the orientation is known, gravity can be ”subtracted” from the accelerometer data to recover the controller acceleration. The acceleration is part of the state, and it can also be used to reduce noise in the 3D position and to derive the 3D velocity. The gyroscope data is also crucial because it directly provides angular velocity. When integrated, this provides a responsive measure of 3D rotation (relative orientation) and can be used to derive angular acceleration. The remaining unknown, absolute yaw, is the most tricky, but it can be inferred by comparing the motion direction computed from body-relative inertial measurements to the motion direction computed from camera-relative image measurements.

Again, though theoretically straightforward, in practice there are many issues for sensor fusion. Because each controller, the PlayStation Eye, and the PS3 are driven by independent clocks, each piece of sensor data must be carefully timestamped so that it can be combined with other sensor data properly. Inertial sensor biases must be computed dynamically as they vary, most often due to changes in temperature. When the sphere is temporarily occluded or beyond the camera view, the state must be updated despite the lack of direct position measurements. And wireless data dropouts, though infrequent, must be hidden by extrapolating the state based on previous inertial data.

5.3 Spatial Input vs. Buttons

Despite the complexity of the underlying technology, the PlayStation Move SDK makes accessing the high-level state straightforward. At any instant in time, the 3D state elements represent a complete spatial description of the controller, and the button status captures which buttons are being pressed. Developers are free to decide how this spatial and button input will be used to interface to their game.

How to interpret the controller state data is an important game design decision. As we learned when developing EyeToy, though it is possible to make some great experiences without buttons, for other great

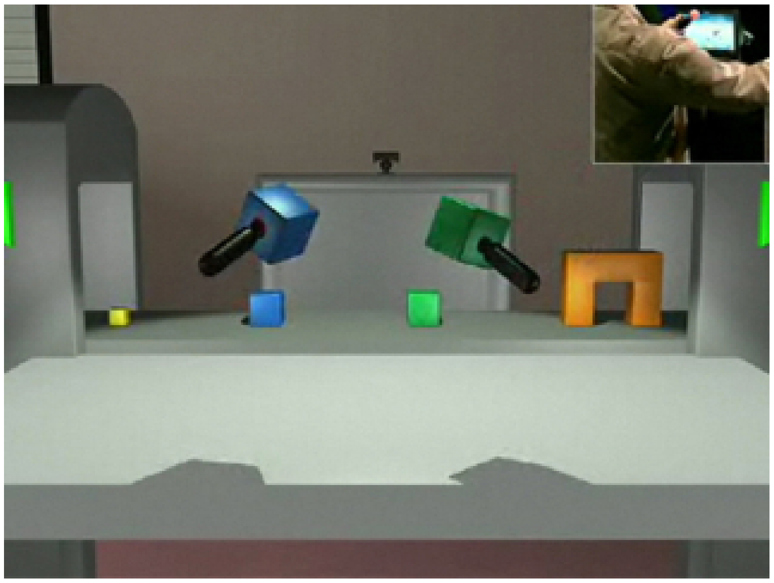


Figure 38: *PlayStation Move E3 2009 demonstration showing object manipulation.*

experiences buttons are both necessary and desirable. Spatial data is appropriate for some actions, but difficult to map to others. Alternatively, buttons can be abstracted to trigger nearly any action, but this abstraction can cause a disconnect between the player and the game. Spatial interaction is effective for actions such as manipulation, selection, pointing, or movement. Also, spatial interaction is applicable when multiple degrees of freedom must be modified simultaneously. An early study into the design issues for spatial input can be found in [Hinckley et al. 1994]. Buttons are most effective when an action is binary, or when it needs to occur often. Buttons are also ideal for time-sensitive actions, since very little time is required for the actual player action.

Providing a combination of spatial input and buttons was a key design strategy for PlayStation Move, as it provides the most interaction possibilities. The analog T button in particular was designed to complement the spatial input. Though many might initially dismiss this button as just a nice-feeling trigger, the long-throw design allows a player to easily control the analog level by altering the position of the button (not the pressure, which is much harder to control effectively). This feature of the analog T button makes it ideal as a modifier for spatial actions, such as controlling the line thickness while drawing or the rate of a variable-speed drill. It offers great affordance for grasping actions in particular, because squeezing the T button is directly analogous to grasping.

5.4 1-to-1 Data Mapping vs. Other Mappings

The spatial input for the PlayStation Move available to a game is the complete 3D pose and motion of the controller, and the game must decide how best to use this state. The controller pose may be used without modification to achieve the special case of ”1-to-1” mapping of controller movement to movement of a virtual object. Alternatively, the 3D state can be mapped in a less direct manner, ranging from a simple geometric transformation of the state to an abstracted interpretation of the data (such as gesture recognition).

5.4.1 1-to-1 Benefits and Challenges

True 1-to-1 mapping of the controller state to a virtual object creates a very realistic and intuitive interface. The player feels very connected to the virtual experience, because of the close correspondence between his own physical motion and the observed motion of a game object. Since the absolute position and orientation are measured, not only does the player's motion matter, but also where he is performing the action also matters. A strong sense of control results as players quickly realize their actions are mimicked perfectly, and little training or explanation is needed in most cases. This manifests as a very "first-person" experience, with the player feeling as if he is directly performing an action himself rather than just issuing a command that it should occur.

An even stronger first-person feeling arises for applications that use a true 1-to-1 mapping to make the controller behave as a pointing device, such as a gun or a flashlight. This is implemented by using the full 3D controller pose and the display size/location to calculate the intersection of a ray cast from the controller with the display. The result feels as if the display is a window to a virtual world, and the controller can point from the real-world through this window and into the virtual world. Similarly, if the 1-to-1 mapping is used to position the virtual vantage point by updating the display view transform, the rendering is as if the virtual world is being imaged from the point of view of the controller. This is much like the DeskTop VR effect for face tracking described earlier, except that because the controller is held in the player's hand, it is as if the virtual world is being see through a "camcorder" that the player is holding.

Several challenges can arise when using 1-to-1 mapping, mostly related to a discrepancy between the constraints that exist in the real world or virtual worlds. For example, in the real-world, there is a limit to the volume over which the controller can be tracked, which therefore limits the virtual coverage space. For some games, this can be addressed by keeping the experience "on rails" so that the coverage space automatically moves in the virtual space as dictated by the game. To achieve more free-form movement about the virtual space, the spatial navigation techniques described earlier could be applied by using a "clutch" to to disengage the 1-to-1 mapping temporarily [Hinckley et al. 1994] or by using a second motion controller. Alternatively, a 1-to-1 mapped motion controller in one hand can be used in conjunction with a more traditional game controller in the other (such as the DualShock, or the PlayStation Move navigation controller).

Conversely, in the virtual world, sometimes there are obstacles such that a 1-to-1 mapping of the controlled virtual object would result in an impossible game situation for certain real-world poses of the controller. The game cannot control the player's actions to avoid such situations, so it must handle them. Simple collision detection and response may suffice if the obstacles are much lighter than the controlled virtual object, but generally other techniques must be used. One option is to simply break the 1-to-1 mapping temporarily and restrict the game object motion, though that may create a disconnect in the player experience. Another option is to "ghost" the game object and allow it complete freedom in the virtual space, though that may break the desired game behavior. Finally, a combination approach can be used, such that when an impossible situation arises, the game object is temporarily disconnected but a "ghost" version is drawn until the situation is reconciled.

Finally, if the controlled virtual object is to be held by a virtual game character, there can be a mismatch in the range of motion achievable for the character vs. the player. For simple humanoid game characters, this can be addressed by calibrating the player size to compute an appropriate scaling factor to map the player motion to the character. For more exotic characters, it may not be possible to use a 1-to-1 mapping directly; instead, a 1-to-1 mapping could be used to calculate a target goal, and a fitting function could be used to compute the character state that comes closest to achieving the goal. A similar effect may be achieved by applying forces to move the character generated by virtual springs between the target goal location and the current object location. Note that though such a technique used the 1-to-1 mapping for computation, the resulting player experience may not be perceived as 1-to-1, and an alternative mapping that is not 1-to-1 might be equally effective.

5.4.2 Other Possible Data Mappings

There are an endless number of other possible data mappings for the PlayStation Move controller state. The analog, multidimensional nature of the data makes it ideal for mapping to other parameterizations. Simple direct mappings using a subset of the pose are often useful, such as mapping the controller XY position to a 2D cursor position on the screen, or perhaps the controller yaw and pitch to the same 2D cursor position. Parts of the controller pose can also be mapped to non-spatial parameters such as color or a character's smile. Nonlinear mapping functions can be used to further extend the mapping possibilities; for example, trigonometric functions are useful for creating a periodic mapping, while exponential and logarithmic functions are commonly used to increase either the range or the precision as desired.

Other input data can be mapped to game parameters in addition to the absolute pose data. The relative pose with respect to a temporary reference can be used, or more generally the differential pose between two motion controllers. This data can be distilled down to a single value, like the magnitude of the distance between two controllers, to provide a simple control parameter. The controller velocities and accelerations can also be mapped to game parameters, with the magnitude and direction being particularly useful. For example, the speed at which the controller is moved can be used to control the intensity of a magic spell, or the direction of acceleration can be used to aim an attack.

The possibilities for mapping spatial data are limitless, but unlike1-to-1, other mappings may not be obvious or intuitive. Because of this, immediate graphical feedback is important, and extra work may be needed by the game to educate the player on how to use the control scheme. This is especially important for highly abstracted mappings such as gestural input, which can be useful for triggering and influencing actions that are not possible or practical with direct mapping (e.g. doing a back flip).

6 3D Gesture Recognition Techniques

Recognizing 3D gestures has had a long history in the virtual reality and 3D user interface communities. The motion controllers we have discussed thus far can all be used to detect 3D gestures for different video game applications. We divide gesture recognition into heuristic-based and machine-learning approaches. With heuristic-based approaches, the designer manually identifies the heuristics that classify a particular gesture and differentiate it from other gestures that the system recognizes. Although this manual approach can be time consuming and tedious, especially as the set of recognized gestures grows, it's also highly understandable. Understandability is useful when new gestures are added to the gesture set and must be differentiated from past gestures. Simple heuristics such as the ones we've discussed in the section on Wiimotes can be effective, especially when composed into more complex heuristics. With machine learning approaches, gestures can be recognized by finding useful features in the input data stream. These features define a feature vector that is used as part of a machine learning algorithm. Typically, a set of gestures are collected to train the machine learning algorithm, which is used to match the feature vector to the set of possible gestures, differentiating between them.

In this section, we will examine two commonly used machine learning algorithms, a linear classifier and an algorithm using the AdaBoost framework, and conduct experiments to examine their accuracy [Hoffman et al. 2010]. There are, of course, a variety of different machine learning algorithms that could be used to recognize 3D gesture and the reader is encouraged to examine [Duda et al. 2001]. We make use of the Wiimote to evaluate these algorithms, but data from any of the devices we have discussed in this course could be used. The only difference would be in how the features are calculated.

6.1 Linear Classifier

The linear classifier is based on Rubine's gesture recognition algorithm [Rubine 1991]. Given a feature vector $\mathbf{f}$, associated with a given 3D gesture g in a gesture alphabet C, a linear evaluation function is derived over the features. The linear evaluation function is given as

$$g_c = w_{c0} + \sum_{i=1}^{F} w_{ci} f_i \tag{21}$$

where $0 \leq c < C$, F is the number of features, and w_{ci} are the weights for each feature associated with each gesture in C. The classification of the correct gesture g is the c that maximizes g_c.

Training of this classifier is done by finding the weights w_{ci} from the gesture samples. First, a feature vector mean $\mathbf{f_c}$ is calculated using

$$\bar{f}_{ci} = \frac{1}{E_c} \sum_{e=0}^{E_c - 1} f_{cei} \tag{22}$$

where f_{cei} is the i^{th} feature of the e^{th} example of gesture c and $0 \leq e < E_c$ where E_c is the number of training samples for gesture c. The sample covariance matrix for gesture c is

$$\Sigma_{cij} = \sum_{e=0}^{E_c-1} (f_{cei} - \bar{f_{ci}})(f_{cej} - \bar{f_{cj}}). \quad (23)$$

The Σ_{cij} are averaged to create a common covariance matrix

$$\Sigma_{ij} = \frac{\sum_{c=0}^{C-1} \Sigma_{cij}}{-C + \sum_{c=0}^{C-1} E_c}. \quad (24)$$

The inversion of Σij then lets us calculate the appropriate weights for the linear evaluation functions,

$$w_{cj} = \sum_{i=1}^{F} (\Sigma^{-1})_{ij} \bar{f_{ci}} \quad (25)$$

and

$$w_{c0} = -\frac{1}{2} \sum_{i=1}^{F} w_{ci} \bar{f_{ci}} \quad (26)$$

where $1 \leq j \leq F$.

6.2 AdaBoost Classifier

The AdaBoost algorithm is based on LaViola's pairwise AdaBoost classifier [LaViola and Zeleznik 2007]. AdaBoost [Schapire 1999] takes a series of weak or base classifiers and calls them repeatedly in a series of rounds on training data to generate a sequence of weak hypotheses. Each weak hypothesis has a weight associated with it that is updated after each round, based on its performance on the training set. A separate set of weights are used to bias the training set so that the importance of incorrectly classified examples are increased. Thus, the weak learners can focus on them in successive rounds. A linear combination of the weak hypotheses and their weights are used to make a strong hypothesis for classification.

More formally, for each unique 3D gesture pair, our algorithm takes as input training set $(\mathbf{x}_1, y_1), ..., (\mathbf{x}_m, y_m)$, where each $\mathbf{x}_i$, represents a feature vector containing J features. Each y_i labels $\mathbf{x}_i$ using label set $Y = \{-1, 1\}$, and m is the total number of training samples. Since we are using a pairwise approach, our algorithm needs to train all unique pairs of gestures. For each unique pair, the AdaBoost algorithm is called on a set of weak learners, one for each feature discussed in Section 6.3. We chose this approach because we found, based on empirical observation, that our features can discriminate between different 3D gesture pairs effectively. We wanted the features to be the weak learners rather than having the weak learners act on the features themselves. Thus, each weak learner C_j uses the j^{th} element in the $\mathbf{x}_i$ training samples, which is noted by $\mathbf{x}_i(j)$ for $1 \leq j \leq J$.

6.2.1 Weak Learner Formulation

We use weak learners that employ a simple weighted distance metric, breaking $(\mathbf{x}_1, y_1), ..., (\mathbf{x}_m, y_m)$ into two parts corresponding to the training samples for each gesture in the gesture pair. Assuming the training samples are consecutive for each gesture, we separate $(\mathbf{x}_1, y_1), ..., (\mathbf{x}_m, y_m)$ into $(\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)$ and $(\mathbf{x}_{n+1}, y_{n+1}), ..., (\mathbf{x}_m, y_m)$ and define $D^1(i)$ for $i = 1, ..., n$ and $D^2(i)$ for $i = n+1, ..., m$ to be training weights for each gesture. Note that in our formulation, D^1 and D^2 are the training weights calculated in the AdaBoost algorithm (see Section 6.2.2).

For each weak learner C_j in each feature vector $\mathbf{x}_i(j)$ in the training set, the weighted averages are then calculated as

$$\mu_{j1} = \frac{\sum_{k=1}^{n} x_k(j) D^1(k)}{\sum_{l=1}^{n} D^1(l)} \tag{27}$$

and

$$\mu_{j2} = \frac{\sum_{k=n+1}^{m} x_k(j) D^2(k)}{\sum_{l=n+1}^{m} D^2(l)}. \tag{28}$$

These averages are used to generate the weak hypotheses used in the AdaBoost training algorithm. If a given feature value for a candidate gesture is closer to μ_{j1}, the candidate is labeled as a 1, otherwise the candidate is labeled as a -1. If the feature value is an equal distance away from μ_{j1} and μ_{j2}, we simply choose to label the gesture as a 1. Note that it is possible for the results of a particular weak classifier to obtain less than 50% accuracy. If this occurs the weak learner is reversed so that the first gesture receives a -1 and second gesture receives a 1. This reversal lets us use the weak learner's output to the fullest extent.

6.2.2 AdaBoost Algorithm

For each round $t = 1, ..., T * J$ where T is the number of iterations over the J weak learners, the algorithm generates a weak hypothesis $h_t : X \to \{-1, 1\}$ from weak learner C_j and the training weights $D_t(i)$ where $j = mod(t-1, J) + 1$ and $i = 1, ..., m$. This formulation lets us iterate over the J weak learners and still conform to the AdaBoost framework [Schapire 1999]. Indeed, the AdaBoost formulation allows us to select weak classifiers from different families at different iterations.

Initially, $D_t(i)$ are set equally to $\frac{1}{m}$, where m is the number of training examples for the gesture pair. However, with each iteration the training weights of incorrectly classified examples are increased so the weak learners can focus on them. The strength of a weak hypothesis is measured by its error

$$\epsilon_t = Pr_{i \sim D_t}[h_t(\mathbf{x}_i(j)) \neq y_i] = \sum_{i : h_t(\mathbf{x}_i(j)) \neq y_i} D_t(i). \tag{29}$$

Given a weak hypothesis, the algorithm measures its importance using the parameter

$$\alpha_t = \frac{1}{2} \ln\left(\frac{1 - \epsilon_t}{\epsilon_t}\right). \tag{30}$$

With α_t, the distribution D_t is updated using the rule

$$D_{t+1}(i) = \frac{D_t(i)\exp(-\alpha_t y_i h_t(\mathbf{x}_i(j)))}{Z_t} \tag{31}$$

where Z_t is a normalization factor ensuring that D_{t+1} is a probability distribution. This rule increases the weight of samples misclassified by h_t so that subsequent weak learners will focus on more difficult samples. Once the algorithm has gone through $T * J$ rounds, a final hypothesis

$$H(x) = \text{sgn}\left(\sum_{t=1}^{T*J} \alpha_t h_t(x)\right) \tag{32}$$

is used to classify gestures where α_t is the weight of the weak learner from round t, and h_t is the weak hypothesis from round t. If $H(x)$ is positive, the new gesture is labeled with the first gesture in the pair and if $H(x)$ is negative it is labeled with the second gesture in the pair.

These strong hypotheses are computed for each pairwise recognizer with the labels and strong hypothesis scores tabulated. To combine the results from each strong hypothesis we use the approach suggested by Friedman [Friedman 1996]; the correct classification for the new gesture is simply the one that wins the most pairwise comparisons. If there is a tie, then the raw scores from the strong hypotheses are used and the one of greatest absolute value breaks the tie.

6.3 Feature Set

The features used in the linear and AdaBoost classifier on Rubine's feature set [Rubine 1991]. Since Rubine's features were designed for 2D gestures using the mouse or stylus, they were extended to work for 3D gestures. The Wiimote and Wii MotionPlus sense linear acceleration and angular velocity respectively. Although we could have derived acceleration and angular velocity-specific features for this study, we chose to make an underlying assumption to treat the acceleration and angular velocity data as position information in 3D space. This assumption made it easy to adapt Rubine's feature set to the 3D domain and the derivative information from the Wiimote and Wii MotionPlus.

The first feature in the set, quantifies the total duration of the gesture in milliseconds, followed by features for the maximum, minimum, mean and median values of x,y and z. Next we analyzed the coordinates in 2D space by using the sine and cosine of the starting angle in the XY and the sine of the starting angle in the XZ plane as features. Then the feature set included features with the sine and cosine of the angle from the first to last points in the XY and the sine of the angle from the first to last points in the XZ plane. After that, features for the total angle traversed in the XY and XZ planes, plus the absolute value and squared value of that angle, completed the features set which analyzes a planar surface. Finally, the length of the diagonal of the bounding volume, the Euclidian distance between the first and last points, the total distance traveled by the gesture and the maximum acceleration squared fulfill Rubine's list.

Initially, this feature set was used for both the Wiimote and the Wii MotionPlus[1], totaling 58 features used in the two classifiers. However, after some initial pilot runs, a singular common covariance matrices was forming with the linear classifier. A singular common covariance matrix cause the matrix inverse needed to

[1]The Wii MotionPlus used maximum angular velocity squared instead of maximum acceleration squared.

find the weights for the linear evaluation functions impossible. These singular matrices were formed when trying to use the feature set with the Wii MotionPlus attachment. Because of this problem, the MotionPlus feature set was culled to use only the minimum and maximum x, y, and z values, the mean x, y, and z, values, and the median x, y, and z values. Thus, 29 features were used when running the experiments with the Wiimote and 41 features were used when running experiments with the Wiimote coupled with the Wii MotionPlus.

6.4 Gesture Set

The majority of the work on 3D gesture recognition with accelerometer and gyroscope-based input devices contain experiments using only four to 10 unique gestures [Kratz et al. 2007; Rehm et al. 2008; Schlömer et al. 2008]. In order to better understand how many gestures these types of devices can accurately recognize and how many training samples would be needed to do so, we chose to more than double that set and use 25 gestures. The gestures, depicted graphically in Figure 39, are performed by holding the Wiimote in various orientations. For a majority of the gestures, the orientation and rotation are the same for both left and right handed users. The only exceptions are the gestures Tennis Swing, Golf Swing, Parry, and Lasso. The gestures Tennis Swing and Golf Swing are performed on either the left or right side of the user, corresponding to the hand that holds the Wiimote. For the Parry gesture, a left-handed person will move the Wiimote towards the right, while a right-handed person will move the Wiimote towards the left. Finally, the Lasso gesture requires a left- and right-handed user to rotate in opposite directions, clockwise versus counterclockwise, respectively.

This gesture set was developed by examining existing video games that make use of 3D spatial interaction, specifically from the Nintendo Wii gaming console. We examined a variety of different games from different genres including sports games, first person shooters, fighting games, and cooking games. Although the gesture set is game specific, it is general enough to work in a wide variety of virtual and augmented reality applications. Initially, we focused on simple movements such as line to the right and line to the left which could be applied to various actions. From there, slightly more complex gestures were added which involved closed figures such as the square, triangle and circle. With this group, we add user variations in velocity, duration and distance traversed. Finally the last set of maneuvers allowed for more freedom in body movement in an effort to help disambiguate gestures during feature analysis. Examples of these gestures include golf swing, whip and lasso.

6.5 3D Gesture Data Collection

Machine learning algorithms such as the linear and AdaBoost classifiers need data to train on. In order for the algorithms to learn and then recognize gestures, a gesture database was created to use in both training and testing. We recruited 17 participants (4 female and 13 male) from the University of Central Florida, of which four men were left handed, to provide the gesture samples. Each participant had experience using the Nintendo Wii gaming console, with two participants doing so on a weekly basis.

Several steps were taken to ensure that participants provided good gestures samples. First, they were given a brief overview of the Wiimote gesture data collection interface and a demonstration of how to interact with the application. After the demonstration, participants were presented with an introduction screen asking them to choose what hand they would hold the Wiimote with to perform the gestures. This decision also told the application to present either left or right-handed gesture demonstration videos. Next, an orientation window (see Figure 40) was displayed to help participants learn how to perform the

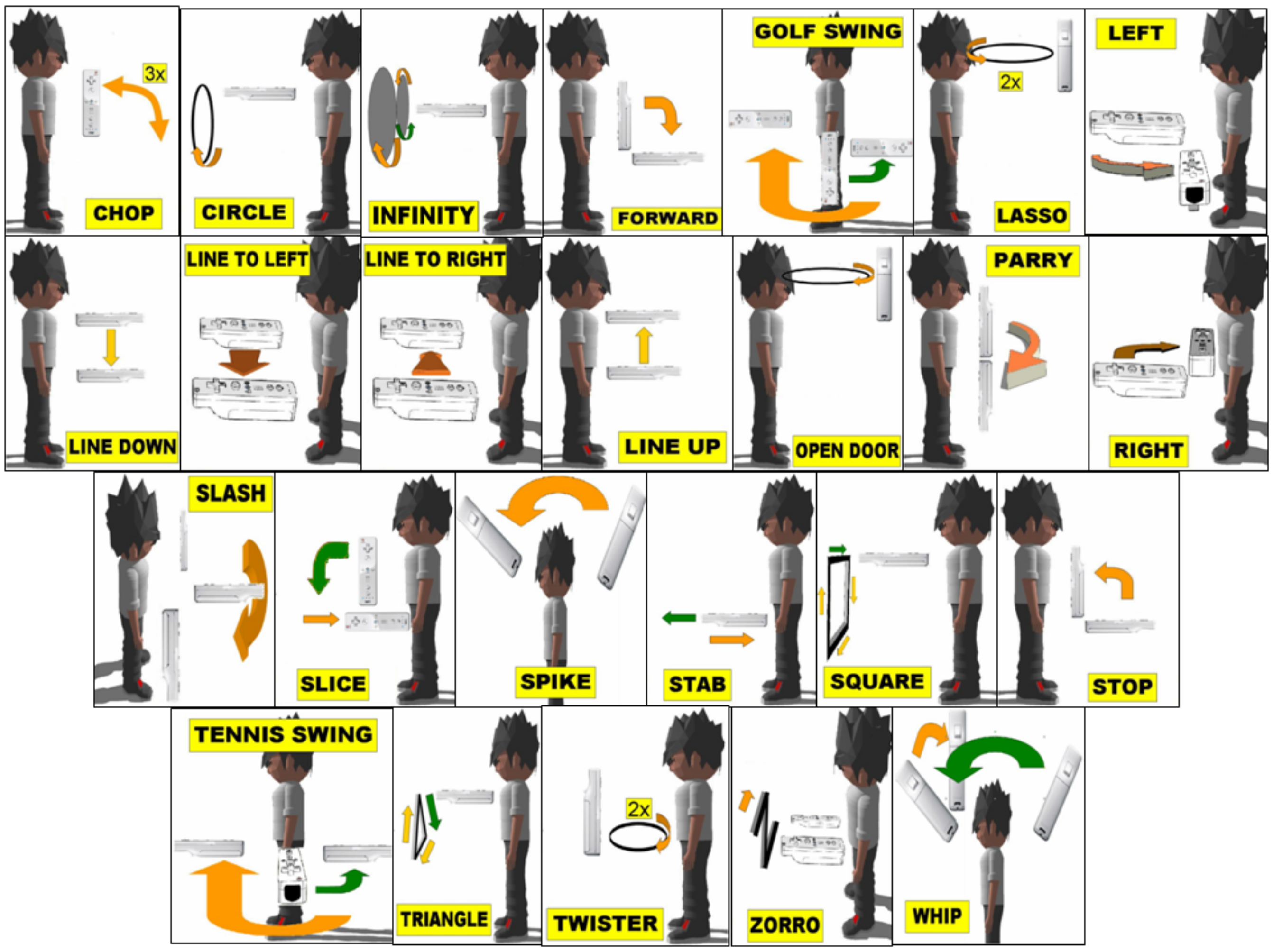
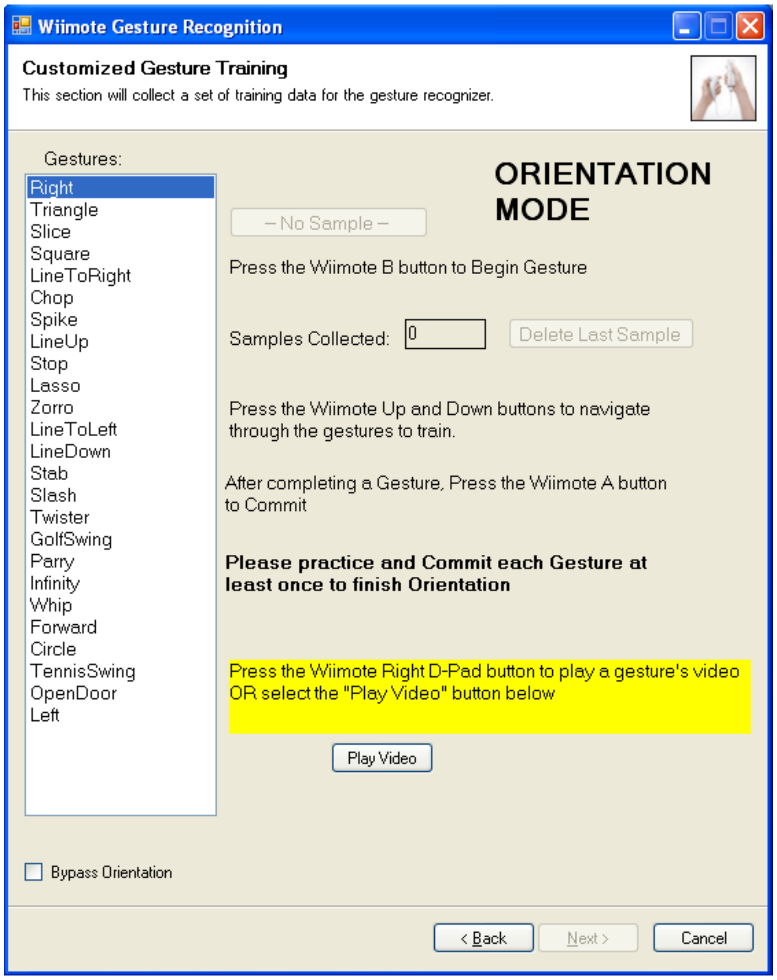


Figure 39: *Illustration showing the 25 gestures used in our study performed with a Wiimote. For compound movements the green arrows show the initial movement and the orange arrows show the remaining movements.*

gestures before data collection began. Each of the 25 gestures are randomly listed on the left hand side of the window to reduce the order effect due to fatigue. Participants moved through the list of gestures using the up and down buttons on the Wiimote. At any time, participants could view a video demonstrating how to perform a gesture by pressing the Wiimote's right button. These videos demonstrated how to hold the Wiimote in the proper orientation in addition to showing how to actually perform the motion of the gesture. Before the gesture collection experiment began, an orientation phase required participants to perform and commit one sample for each gesture. This reduced the order effect due to the learning curve. To perform a gesture, participants pressed and held down the Wiimote's "B" button to make a gesture. After participants released the "B" button, they could either commit the gesture by pressing the Wiimote's "A" button or redo the gesture by pressing and holding the "B" button and performing the gesture again. This feature enabled participants to redo gesture samples they were not happy with. Once a gesture sample is committed to the database, the system pauses for two seconds, preventing the start of an additional sample. This delay between samples allows participants to reset the Wiimote position for the next gesture sample. In addition, the delay prevents the user from trying to game the system by quickly performing the same gesture with little to no regard of matching the previous samples of that gesture.


Wiimote Gesture Recognition

Customized Gesture Training

This section will collect a set of training data for the gesture recognizer.

Gestures:

Right
Triangle
Slice
Square
LineToRight
Chop
Spike
LineUp
Stop
Lasso
Zorro
LineToLeft
LineDown
Stab
Slash
Twister
GolfSwing
Parry
Infinity
Whip
Forward
Circle
TennisSwing
OpenDoor
Left

ORIENTATION MODE

– No Sample –

Press the Wiimote B button to Begin Gesture

Samples Collected: 0

Delete Last Sample

Press the Wiimote Up and Down buttons to navigate through the gestures to train.

After completing a Gesture, Press the Wiimote A button to Commit

Please practice and Commit each Gesture at least once to finish Orientation

Press the Wiimote Right D-Pad button to play a gesture's video OR select the "Play Video" button below

Play Video

Bypass Orientation

< Back | Next > | Cancel

Figure 40: *The gesture collection window that each participant interacted with in order to provide 20 samples for each of the the 25 gestures.*

After participants created one sample for each of the 25 gestures, they entered data collection mode. The only difference between orientation mode and data collection mode is the amount of samples which need to be committed. When participants commit a training sample for a particular gesture, a counter is decremented and displayed. A message saying the collection process for a given gesture is complete is shown after 20 samples have been entered for a gesture. This message indicates to the participant that they can move on to providing data for a remaining gesture. A total of 20 samples for each gesture, or 500 samples in all, is required for a full user training set. In total, 8,500 gesture samples were collected[2]. Each data collection session lasted from 30-45 minutes.

6.6 3D Gesture Recognition Experiments

With the 3D gesture database, we were able to run a series of experiments to examine both learning algorithms in terms of user dependent and user independent recognition. The primary metric analyzed in our experiments is gesture classification accuracy, while having the over-arching goal of maximizing the number of gestures correctly recognized at varying degrees of training the machine learning algorithms.

[2]The gesture database can be downloaded at http://www.eecs.ucf.edu/isuelab/downloads.php.

For both the dependent and independent experiments, the linear and AdaBoost classifiers were tested using the Wiimote data only and using the Wiimote in conjunction with the Wii MotionPlus attachment. Thus, the experiments attempted to answer the following questions for both the user dependent and independent cases:

- How many of the 25 gestures can each classifier recognize with accuracy over 90%?
- How many training samples are needed per gesture to achieve over 90% recognition accuracy?
- How much accuracy improvement is gained from using the Wii MotionPlus device?
- Which classifier performs better?
- Which gestures in our gesture set cause recognition accuracy degradation?

6.6.1 User Dependent Recognition Results

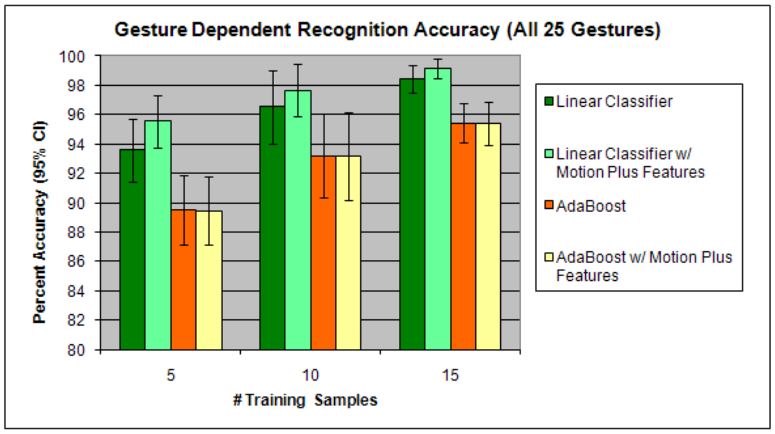


Figure 41: *The average recognition results for the user dependent experiments over all 25 gestures. The results are grouped by the number of training samples (5, 10, or 15) provided for each gesture within an experiment. A single user dependent experiment utilized two classifiers (linear and AdaBoost), each executing with either the Wiimote or Wiimote + MotionPlus input device producing four accuracy values.*

For the user dependent recognition experiments, we used a subset of samples for each gesture from a single user to train the machine learning algorithms and the remaining samples to test the recognizers. This approach is equivalent to having a user provide samples to the recognizer up front, so the algorithm can be tailored to that particular user. For each user dependent test, two categories of experiments were created: classification over all 25 gestures and finding the maximum recognition accuracy rate over as many gestures as possible. Both categories were then broken into three experiments providing 5, 10, or 15 training samples per gesture with the remaining samples used for testing accuracy. Each experiment was executed on all four of the classifiers mentioned earlier. The experiment set was conducted on each of the 17 participant's data.

The results of trying to recognize all 25 gestures in each experiment are shown in Figure 41. We analyzed this data using a 3 way repeated measures ANOVA and found significance for the number of training samples ($F_{2,15} = 17.47, p < 0.01$), the classification algorithm ($F_{1,16} = 119.42, p < 0.01$), and the use of the Wii MotionPlus data ($F_{1,16} = 8.23, p < 0.05$). To further analyze the data, pairwise t tests were run. The most notable observation is the high level of accuracy across all experiments. In particular, the

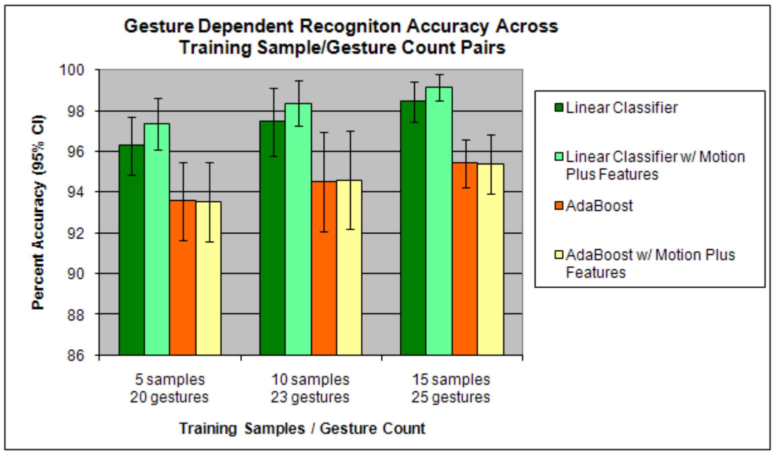


Figure 42: *The average recognition results for the user dependent experiments over a varying number of gestures. The goal of this experiment was to recognize, with high accuracy, as many gestures as possible with different levels of training. The results are grouped by the number of training samples (5, 10, or 15) provided for each gesture within an experiment. A single user dependent experiment utilized two classifiers (linear and AdaBoost), each executing with either the Wiimote or Wiimote + MotionPlus input device producing four accuracy values.*

Comparison	Test Statistic	P Value
Linear5 - Ada5	$t_{16} = 7.17$	$p < 0.01$
LinearMP5 - AdaMP5	$t_{16} = 8.36$	$p < 0.01$
Linear10 - Ada10	$t_{16} = 6.48$	$p < 0.01$
LinearMP10 - AdaMP10	$t_{16} = 6.54$	$p < 0.01$
Linear15 - Ada15	$t_{16} = 7.69$	$p < 0.01$
LinearMP15 - AdaMP15	$t_{16} = 7.16$	$p < 0.01$

Table 1: *Results from a set of t-tests showing significance differences between the linear classifier and AdaBoost indicating the linear classifier outperforms AdaBoost in our test cases. Note that under the comparison column, MP stands for whether the Wii MotionPlus was used and the number represents how many samples were used for training the algorithms.*

linear classifier gave a mean accuracy value of 93.6% using only 5 training samples for each gesture and 98.5% using 15 training samples per gesture ($t_{16} = -5.78, p < 0.01$). Furthermore, the linear classifier outperformed AdaBoost in every situation by at least 3% (see Table 1 for the statistical results). Another important result shown in Figure 41 is the role of the Wii MotionPlus in both recognition algorithms. The Wii MotionPlus significantly increased the recognition accuracy for the linear classifier to 95.5% using 5 training samples ($t_{16} = -2.81, p < 0.05$) per gesture and 99.2% using 15 training samples per gesture ($t_{16} = -2.54, p < 0.05$). For AdaBoost, the change in accuracy was negligible.

While these accuracy levels are good for some applications, it is important to know which gestures would need to be removed from the classification set in order to improve recognition results. To begin this examination, the set of gestures frequently recognized incorrectly from the previous three experiment categories were removed. Within this set we noticed a few gestures with similar attributes such as Tennis Swing and Golf Swing or Square and Triangle. These gesture pairs involve similar movement which caused the classifiers to misidentify each type as the other. Once the poorly classified gestures were extracted the accuracy results easily increased to above 98%. We systematically added each of the removed gestures back into the gesture set on a case by case basis, leaving one gesture from each similar pairing out in order to improve the recognition on the other included gesture from each pair. The results are shown in Figure 42. As with the previous experiment, we ran a 3 way repeated measures ANOVA as well as t tests and found significant results for the training sample/number of gestures pairs ($F_{2,15} = 5.01, p < 0.05$), the classification algorithm ($F_{1,16} = 48.55, p < 0.01$), and the use of the Wii MotionPlus data ($F_{1,16} = 9.69, p < 0.01$).

In the experiment using 5 training samples, the gestures Forward, Golf Swing, Spike, Triangle and Line to Left were removed, thereby producing over 93.5% accuracy for AdaBoost and over 96.3% for the linear classifier. Once the number of training samples was increased to 10, the set of removed gestures included only Spike and Triangle. This experiment yielded higher accuracy with the linear classifier ($t_{16} = -1.90, p = 0.075$) and AdaBoost ($t_{16} = -1.07, p = 0.3$), but these results were not significant. Finally, since the recognition rates for 15 training samples were already greater than 95%, no gestures were removed during this last test. These results show that recognition accuracy rates as high as 97% can be achieved for 20 gestures using only 5 samples per gesture and 98% for 23 gestures using 10 samples per gesture with the Wiimote coupled with the Wii MotionPlus attachment. In fact, for the linear classifier, the recognizer obtained significantly higher accuracy using the Wii MotionPlus attachment in the 5 training sample/20 gesture case ($t_{16} = -2.32, p < 0.05$) and the 10 training sample/23 gesture case ($t_{16} = -2.18, p < 0.05$).

6.6.2 User Independent Recognition Results

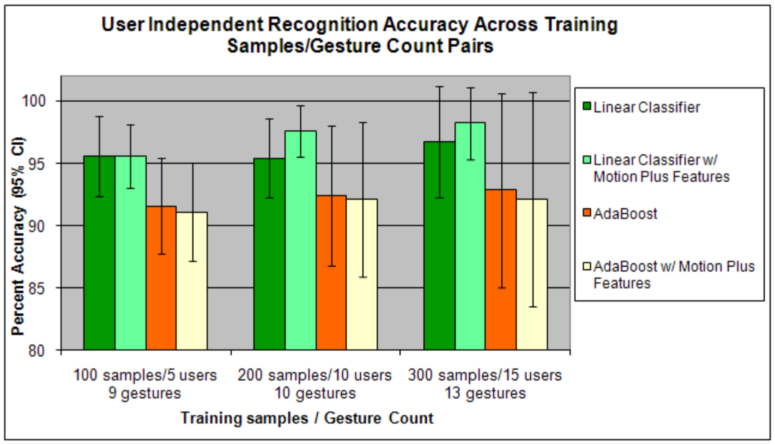


Figure 43: *The average recognition results for the user independent experiments over a varying number of gestures. The goal of this experiment was to recognize, with high accuracy, as many gestures as possible when given different levels of training. The results are grouped by the number of user training samples (100,200, or 300) per gesture used for training within an experiment. These numbers are analogous to using 5, 10, and 15 users' data for training. A single user independent experiment utilized two classifiers (linear and AdaBoost), each executing with either the Wiimote or Wiimote + MotionPlus input device producing four accuracy values.*

For the user independent recognition experiments, a subset of the 17 user study gesture data files was used for training. From the remaining files, recognition is performed and reported on a per user basis. This approach is equivalent to having the recognizers trained on a gesture database and having new users simply walk up and use the system. The benefit of this approach is there is no pre-defined training needed for each user at a potential cost in recognition accuracy. We ran three tests in this experiment, using data from 5, 10, and 15 users from our gesture database for training with the remaining user data for testing. As in the user dependent study, each scenario was executed on all four of the classifiers mentioned earlier. Note that for the independent recognition results, we did not perform any statistical analysis on the data because we did not have an equal number of samples for each condition and the sample size for the 15 user case as two small (only two test samples). An approach to dealing with this issue is to perform cross validation on the data so proper statistical comparisons can be made.

The results, shown in Figure 43, shows that the linear classifier outperforms AdaBoost by at least 3% and using the Wii MotionPlus attachment improved recognition for only the linear classifier. However, unlike in the user dependent tests, the Wii MotionPlus slightly hindered AdaBoost accuracy. After removing the poorly classified gestures, we followed the reintroduction method we used for the user dependent tests. The linear classifier was able to recognize a total of 9, 10, and 13 gestures with mean accuracy values of 95.6%, 97.6%, and 98.3% respectively using the Wiimote coupled with the Wii MotionPlus attachment. The gestures enabled for the 5 user training set included Forward, Stop, Open Door, Parry, Chop, Circle, Line to Right, Line Up and Stab. When using the 10 user training set, the Twister and Square gestures were added while maintaining slightly higher accuracy levels. On the other hand, the gesture Open Door was removed because the newly introduced training data increased confusion among samples of that type. Finally, for the 15 user training set, the gestures Open Door (again), Infinity, Zorro and Line Down were added but the Twister gesture was removed.

6.7 Discussion

From these experiments, we can see that 25 3D gestures can be recognized using the Wiimote coupled with the Wii MotionPlus attachment at over 99% accuracy in the user dependent case using 15 training samples per gesture. This result significantly improves upon the results in the existing literature in terms of the total number of gestures that can be accurately recognized using a spatially convenient input device. There is, of course, a tradeoff between accuracy and the amount of time needed to enter training samples in the user dependent case. 15 training samples per gesture might be too time consuming for a particular application. As an alternative, the results for 5 training samples per gesture only shows a small accuracy degradation. For the user independent case, we can see the accuracy improves and the number of gestures that can reliably be recognized also increases as the number of training samples increases. Although the overall recognition accuracy and the number of gestures was higher in the dependent case (as expected), the independent recognizer still provides strong accuracy results.

Comparing the two classifiers in the experiments shows the linear classifier consistently outperforms the AdaBoost classifier. This is somewhat counterintuitive given the AdaBoost classifier is a more sophisticated technique. However, as we discussed in Section 6.3, the linear classifier can suffer from the singular matrix problem which can limit its utility when new features that could improve accuracy are added.

With the Wii MotionPlus, we also tried to adjust for gyroscopic drift in data. Our attempt at calibration involved setting the Wiimote coupled with the MotionPlus device on a table with the buttons down for approximately 10 seconds, followed by storing a snap shot of the roll, pitch and yaw values. That snapshot was then used as a point of reference for future collection points. However, using those offsets caused decreased accuracy in AdaBoost and singular matrices in the linear classifier leading to the use of raw gyroscope data instead.

Acknowledgements

A special thanks to Doug Bowman, Ernst Kruijff, Ivan Poupyrev, Chad Wingrave, and members of the Interactive Systems and User Experience Lab for assisting in the development of the material for this course.

References

AZUMA, R., AND BISHOP, G. 1994. Improving static and dynamic registration in an optical see-through hmd. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 197–204.

BOTT, J., CROWLEY, J., AND LAVIOLA, J. 2009. Exploring 3d gestural interfaces for music creation in video games. In *Proceedings of The Fourth International Conference on the Foundations of Digital Games 2009*, 18–25.

BOWMAN, D. A., AND HODGES, L. F. 1997. An evaluation of techniques for grabbing and manipulating remote objects in immersive virtual environments. In *SI3D '97: Proceedings of the 1997 symposium on Interactive 3D graphics*, ACM, New York, NY, USA, 35–ff.

BOWMAN, D. A., KOLLER, D., AND HODGES, L. F. 1997. Travel in immersive virtual environments: An evaluation of viewpoint motion control techniques. In *Proceedings of the Virtual Reality Annual International Symposium*, 45–52.

BOWMAN, D. A., WINEMAN, J., HODGES, L. F., AND ALLISON, D. 1998. Designing animal habitats within an immersive ve. *IEEE Comput. Graph. Appl. 18*, 5, 9–13.

BOWMAN, D. A., KRUIJFF, E., LAVIOLA, J. J., AND POUPYREV, I. 2004. *3D User Interfaces: Theory and Practice*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

CHARBONNEAU, E., MILLER, A., WINGRAVE, C., AND LAVIOLA, JR., J. J. 2009. Understanding visual interfaces for the next generation of dance-based rhythm video games. In *Sandbox '09: Proceedings of the 2009 ACM SIGGRAPH Symposium on Video Games*, ACM Press, 119–126.

CONNER, B. D., SNIBBE, S. S., HERNDON, K. P., ROBBINS, D. C., ZELEZNIK, R. C., AND VAN DAM, A. 1992. Three-dimensional widgets. In *SI3D '92: Proceedings of the 1992 symposium on Interactive 3D graphics*, ACM, New York, NY, USA, 183–188.

CRASSIDIS, J. L., AND MARKLEY, F. L. 2003. Unscented filtering for spacecraft attitude estimation. *Journal of Guidance, Control, and Dynamics 26*, 4, 536–542.

DUDA, R. O., HART, P. E., AND STORK, D. G. 2001. *Pattern Classification*. John Wiley and Sons.

FEINER, S., MACINTYRE, B., HAUPT, M., AND SOLOMON, E. 1993. Windows on the world: 2d windows for 3d augmented reality. In *UIST '93: Proceedings of the 6th annual ACM symposium on User interface software and technology*, ACM, New York, NY, USA, 145–155.

FRIEDMAN, J. H. 1996. Another approach to polychotomous classification. Tech. rep., Department of Statistics, Stanford University.

GIANSANTI, D., MACELLARI, V., AND MACCIONI, G. 2003. Is it feasible to reconstruct body segment 3-d position and orientation using accelerometric data. *IEEE Trans. Biomed. Eng 50*, 2003.

GUNTURK, B. K., GLOTZBACH, J., ALTUNBASAK, Y., AND SCHAFER, R. W. 2005. Demosaicking: color filter array interpolation. *IEEE Signal processing magazine 22*, 44–54.

HINCKLEY, K., PAUSCH, R., GOBLE, J. C., AND KASSELL, N. F. 1994. A survey of design issues in spatial input. In *Proc. ACM UIST'94 Symposium on User Interface Software & Technology*, ACM, 213–222.

HOFFMAN, M., VARCHOLIK, P., AND LAVIOLA, J. 2010. Breaking the status quo: Improving 3d gesture recognition with spatially convenient input devices. In *IEEE Virtual Reality 2010*, IEEE Press, 59–66.

INTEL. 1999. Video as input (vai) white paper. Tech. rep., Intel Architecture Labs.

JULIER, S., AND UHLMANN, J. 1997. A new extension of the kalman filter to nonlinear systems. In *Int. Symp. Aerospace/Defense Sensing, Simul. and Controls*.

KATO, H., AND BILLINGHURST, M. 1999. Marker tracking and hmd calibration for a video-based augmented reality conferencing system. In *IWAR 99: Proceedings of the 2nd IEEE and ACM International Workshop on Augmented Reality*, 85.

KATZOURIN, M., IGNATOFF, D., QUIRK, L., LAVIOLA, J., AND JENKINS, O. C. 2006. Swordplay: Innovating game development through vr. *IEEE Computer Graphics and Applications 26*, 6, 15–19.

KRATZ, L., SMITH, M., AND LEE, F. J. 2007. Wiizards: 3d gesture recognition for game play input. In *Future Play '07: Proceedings of the 2007 conference on Future Play*, ACM, New York, NY, USA, 209–212.

LAVIOLA, J. J., AND ZELEZNIK, R. C. 2007. A practical approach for writer-dependent symbol recognition using a writer-independent symbol recognizer. *IEEE Transactions on Pattern Analysis and Machine Intelligence 29*, 11, 1917–1926.

LAVIOLA, J. 2000. Msvt: A virtual reality-based multimodal scientific visualization tool. In *Proceedings of the Third IASTED International Conference on Computer Graphics and Imaging*, 1–7.

LAVIOLA, J. J. 2003. Double exponential smoothing: an alternative to kalman filter-based predictive tracking. In *EGVE '03: Proceedings of the workshop on Virtual environments 2003*, ACM, New York, NY, USA, 199–206.

LEE, J. 2008. Hacking the nintendo wii remote. *Pervasive Computing, IEEE 7*, 3 (July-Sept.), 39–45.

LUINGE, H., VELTINK, P., AND BATEN, C. 1999. Estimating orientation with gyroscopes and accelerometers. *Technology and Health Care 7*, 6, 455–459.

MAPES, D., AND MOSHELL, M. 1995. A two-handed interface for object manipulation in virtual environments. *Presence: Teleoper. Virtual Environ. 4*, 4, 403–416.

MARKS, R., SCOVILL, T., AND MICHAUD-WIDEMAN, C. 2001. Enhanced reality: A new frontier for computer entertainment. In *SIGGRAPH 2001: ConferenceAbstracts and Applications*, ACM, 117.

MARKS, R., DESHPANDE, R., KOKKEVIS, V., LARSEN, E., MICHAUD-WIDEMAN, C., AND SARGAISON, S. 2003. Real-time motion capture for interactive entertainment. In *SIGGRAPH 2003: Emerging Technologies*, ACM.

MARKS, R. 2000. Medieval chamber. In *SIGGRAPH 2000: Emerging Technologies*, ACM, etech58.

MINE, M. R., BROOKS, JR., F. P., AND SEQUIN, C. H. 1997. Moving objects in space: exploiting proprioception in virtual-environment interaction. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 19–26.

MINE, M. 1995. Virtual environment interaction techniques. Tech. rep., UNC Chapel Hill CS Dept.

PAUSCH, R., BURNETTE, T., BROCKWAY, D., AND WEIBLEN, M. E. 1995. Navigation and locomotion in virtual worlds via flight into hand-held miniatures. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 399–400.

PIERCE, J. S., FORSBERG, A. S., CONWAY, M. J., HONG, S., ZELEZNIK, R. C., AND MINE, M. R. 1997. Image plane interaction techniques in 3d immersive environments. In *SI3D '97: Proceedings of the 1997 symposium on Interactive 3D graphics*, ACM, New York, NY, USA, 39–ff.

POUPYREV, I., BILLINGHURST, M., WEGHORST, S., AND ICHIKAWA, T. 1996. The go-go interaction technique: non-linear mapping for direct manipulation in vr. In *UIST '96: Proceedings of the 9th annual ACM symposium on User interface software and technology*, ACM, New York, NY, USA, 79–80.

REHM, M., BEE, N., AND ANDRÉ, E. 2008. Wave like an egyptian: accelerometer based gesture recognition for culture specific interactions. In *BCS-HCI '08: Proceedings of the 22nd British HCI Group Annual Conference on HCI 2008*, British Computer Society, Swinton, UK, UK, 13–22.

REKIMOTO, J., AND AYATSUKA, Y. 2000. Cybercode: designing augmented reality environments with visual tags. In *DARE 2000: Proceedings of the 2000 ACM Conference on Designing Augmented Reality Environments*, ACM, 1–10.

RUBINE, D. 1991. Specifying gestures by example. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 329–337.

SCHAPIRE, R. E. 1999. A brief introduction to boosting. In *IJCAI '99: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1401–1406.

SCHLÖMER, T., POPPINGA, B., HENZE, N., AND BOLL, S. 2008. Gesture recognition with a wii controller. In *TEI '08: Proceedings of the 2nd international conference on Tangible and embedded interaction*, ACM, New York, NY, USA, 11–14.

SHIRAI, A., GESLIN, E., AND RICHIR, S. 2007. Wiimedia: motion analysis methods and applications using a consumer video game controller. In *Sandbox '07: Proceedings of the 2007 ACM SIGGRAPH symposium on Video games*, ACM, New York, NY, USA, 133–140.

SHIRATORI, T., AND HODGINS, J. K. 2008. Accelerometer-based user interfaces for the control of a physically simulated character. *ACM Trans. Graph. 27*, 5, 1–9.

SHIVARAM, G., AND SEETHARAMAN, G. 1998. A new technique for finding the optical center of cameras. In *ICIP 98: Proceedings of 1998 International Conference on Image Processing*, vol. 2, 167–171.

SILVA, M. G., AND BOWMAN, D. A. 2009. Body-based interaction for desktop games. In *CHI EA '09: Proceedings of the 27th international conference extended abstracts on Human factors in computing systems*, ACM, New York, NY, USA, 4249–4254.

STOAKLEY, R., CONWAY, M. J., AND PAUSCH, R. 1995. Virtual reality on a wim: interactive worlds in miniature. In *CHI '95: Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 265–272.

WAN, E. A., AND VAN DER MERWE, R. 2002. The unscented kalman filter for nonlinear estimation. 153–158.

WILLIAMSON, R., AND ANDREWS, B. 2001. Detecting absolute human knee angle and angular velocity using accelerometers and rate gyroscopes. *Medical and Biological Engineering and Computing 39*, 3, 294–302.

WILLIAMSON, B., WINGRAVE, C., AND LAVIOLA, J. 2010. Realnav: Exploring natural user interfaces for locomotion in video games. In *Proceedings of IEEE Symposium on 3D User Interfaces 2010*, IEEE Computer Society, 3 – 10.

WINGRAVE, C., WILLIAMSON, B., VARCHOLIK, P., ROSE, J., MILLER, A., CHARBONNEAU, E., BOTT, J., AND LAVIOLA, J. 2010. Wii remote and beyond: Using spatially convenient devices for 3duis. *IEEE Computer Graphics and Applications 30*, 2, 71–85.

WLOKA, M. M., AND GREENFIELD, E. 1995. The virtual tricorder: a uniform interface for virtual reality. In *UIST '95: Proceedings of the 8th annual ACM symposium on User interface and software technology*, ACM, New York, NY, USA, 39–40.