Implementation of a 3D Game Using the Wii-Remote as an Enhanced User Interface

Author: Victor Radoslavov Kirov
Student Number: KRVVIC001

Supervisor: Prof. Michael Inggs

A project report submitted to the Department of Electrical Engineering, University of Cape Town, in partial fulfilment of the requirements for the degree of Bachelor of Science in Engineering.

Cape Town, October 2008

Declaration

I declare that this project is my own, unaided work. It is being submitted for the degree of Bachelor of Science in Mechatronics Engineering in the University of Cape Town. It has not been submitted before for any degree or examination in any other university.

Signature of Author . . . .

Cape Town

20 October 2008

This publication is dedicated to all the people who helped me get through varsity.

My parents who funded my many years at varsity and put up with my mental melt-downs.

My sister who went through two of these projects in her life... I don't know how you did it.

The co-members of the tripod who provided endless hours of entertainment and
gave support when it was needed.

My peers and colleagues who helped me when the going got tough and
helped me realise that I wasn't the only one that sometimes lost his
mind after 4 years of Engineering.

Also to the rest of my family, friends, Amok! trainer and peers, Super Sandwich
employees, Glass House Employees and the guy that came up with
"How I Met Your Mother" for getting my mind of work when I needed it.

Thank You All!!!

Acknowledgements

I would like to thank Professor Michael Inggs for his excellent guidance and support throughout the project. I would have been completely lost without him.

I would also like to thank all of my lecturers over the past 4 years who gave me their knowledge to make this project possible.

A further thank you must go out to Doctor Andrew Davison of the Prince of Songkla University, Thailand, for his excellent guide on 3D game developmentDavison [1] and to artist-3d.com for the 3D models used in the figures in this publication.

Abstract

The purpose of this project is to develop a 3D first-person computer game with integrated Wii input user interface. The Wii interface will include head-tracking, virtual world navigation and object interaction. The head-tracking user interface will be achieved using the IR camera on a Wiimote (from a Nintendo Wii) by placing IR LEDs on glasses on the user's head and tracking the movement of the user's head using triangulation techniques. The Wii will also be tested as an input device to navigate through the virtual world in the 3D game, using the button inputs, and also for interacting with objects, using the Wiimote's accelerometers. Once this system is completed and adequately functioning, it will be tested to justify the feasibility of using the Wiimote as an advanced user input device in future gaming development.

Contents

List of Figures

List of Tables

Part I

Overview

User Requirements

The following user requirements were submitted by the supervisor:

"Nintendo have produced a gaming controller (Wii) that allows a user to control objects on the screen with physical motion of a controller. People have reversed the process, and placed the static emitter on the head of an observer. Sample software is available which then allows a user moving his/her head to modify the image on a screen to give the illusion of 3d viewing of objects. The objective of this project is to replicate (rather, to understand) the hardware and software, but also to establish boundaries of what range of 3d vision is possible.

Once the boundaries of the Wii are established, a study will be conducted which tests the application of replacing a keyboard and mouse for 3D gaming by 2 Wii remotes, one for head tracking (giving the user the feeling of perspective and also used for basic actions such as jumping) and another for navigation through the world."

Requirements Review

The following are the reviewed requirements as agreed on by the supervisor and the student:

Materials Presented

1 Wii remote (Wiimote)

1 Bluetooth Dongle

1 PC with Java capabilities and Wii libraries

1 sensor bar or equivalent

Wiimote Capabilities

The sensor bar consists of 2 arrays of infrared lights. The Wii remote has a camera, which only picks up infrared light. The Wii uses the known distance between the lights on the sensor bar to calculate the distance between the remote and the sensor bar by means of triangulation. The angle which the lights produce on the camera is used to measure the roll of the remote. The camera is documented to work in a distance of one to five metres from the sensor bar and has a viewing angle of 45°.

It has been shown that the sensor bar can be replaced with infra-red light emitting sources such as light emitting diodes, light bulbs, natural light from the sun, candles, etc.

The Wii remote also features a three axis accelerometer which allows for 6 degrees of freedom measurements. 12 accessible buttons and 4 indicative LED lights are also featured on the Wiimote. Extensions are also available for the Wiimote.

These can be plugged into the Wiimote and piggyback on the Wiimotes Bluetooth connection to communicate with the parent device.

The Nunchuk is an extension of the Wiimote. It features 2 buttons, a similar accelerometer setup as the Wiimote and a joystick.

Requirements

A coherent API for the Wiimote must be established in a suitable programming language. This API must then be analysed and the communications between the Wiimote and its parent device must be fully understood.

The performance characteristics and limitations of the Wiimote must be established in order to find the limitations to the final project.

Once these steps are complete, create a 3D game making use of two Wiimotes for controllers to navigate and interact with the virtual world.

One Wiimote must be used in reverse as a head tracker to improve the Virtual Reality experience by adding a sense of perspective to the game. Tracking the head will also be used for basic movement functions such as ducking, dodging and jumping.

The second Wiimote will be used with a Nunchuk extension. The Nunchuk joystick will be used for navigation through the world and the buttons and accelerometers on the Wiimote and Nunchuk will be used for interaction with objects.

Document the limitations and possible improvements and future work of this setup. Establish whether the use of the Wii for an improved 3D virtual reality experience is feasible for future game development.

Method

- Find a suitable API which can communicate with the Wiimote and Nunchuk via the PC
- Establish how the Wii communicates with its parent device
- Create software for head tracking which can determine position and distance of the user from the Wiimote

- Create a 3D test application to construct a prototype for perspective using the head tracking software
- Create a 3D virtual world which implements the use of the perspective software and the Nunchuk for navigation
- Ascertain the limitations of the virtual world and possible future improvements to study the feasibility of implementing this setup into future gaming development

Document Summary

Choosing the Wiimote API

When choosing an API, the following was taken into consideration:

- Platform-Independence - allowing the software to run on different operating systems
- Functionality - accessibility to all of the Wiimote's and Nunchuk's functions
- Multi-connectivity support - it is required that 2 Wiimotes are simultaneously connected to the system for the final software to be developed

The WiiUseJ API was chosen as it fulfilled all of the above requirements and proved to be the most stable and compatible.

Navigating the 3D world

When navigating the 3D world, the user needs to be able to:

- move backwards and forwards
- strafe sideways
- turn
- look up and down

When moving backwards, forwards and strafing (assuming the user is initially not moving), the user begins to accelerate in the desired direction. Once the maximum preset allowable speed is reached, the acceleration drops to zero and the user moves with a constant velocity.

Implementation of turning and looking up/down is achieved by the use of a yaw angle and a pitch angle. When looking up and down, the pitch angle is incremented and decremented respectively. When looking left and right, the yaw angle is incremented and decremented respectively. Both of these angles are manipulated by the user by means of the mouse or the Nunchuk joystick, depending on which input is chosen at the beginning of the game.

Implementation of the Virtual World

In creating the world, the following was required:

- implementation of rooms
- collision detection for walls
- collision detection for overhanging objects (if player hits their head on a platform above when jumping or standing up)
- collision detection for obstacle objects in the rooms
- changing the ground level if the user walks onto a platform or falls off of a platform
- implementation of gravity to affect the player when jumping or falling off of a platform

The rooms were created using boxes. The co-ordinates of the vertexes of each of the boxes were stored in a text file and imported on start up. This allowed for easy adding of walls. The layout consisted of one room with 4 corridors. The first corridor had low obstacles and head-height obstacles to test the implementation of jumping and ducking. The second corridor had an array of slow travelling bullets through which the player had to navigate through, getting hit as few times as possible to test a combination of jumping, ducking and dodging. The third corridor had a line of 6 boxing gloves which shot out at random intervals to test the implementation of dodging. The last corridor had a row of platforms which moved in and out of the side walls to compare the ease of manoeuvrability using both keyboard-mouse and Wiimote interfaces.

The information for the vertexes was also used in wall and overhang collision detection. When the user moved, the software first checked if the player would move into any of the boxes which represent a wall if they were allowed to move in the desired direction. If the player was not going to move into a wall, then their desired movement was performed. If they were going to move into the wall, then they were only allowed to move in the direction parallel to the wall, allowing the player to move along a wall without walking through it.

For overhang collision detection, the player would have to jump or stand up from a crouching position and hit an overhanging platform with their head. This was detected by checking if the position of the player's head, in the virtual world, moved into one of the platforms using the vertexes imported from the wall text file to compare. If this happened, then a Boolean value was set, stopping the player from standing up further or jumping higher.

When implementing collision detection with obstacle objects in the room, the player was treated as a cylinder and the objects as a combination of spheres, cylinders and boxes.When the player collided with an obstacle object, the distance which the object's binding box protruded into the player's binding box was calculated and the player was moved outward in a direction away from the centre of the objects binding box so that the player's binding box lay tangent to the object's.

When the player walked off of or onto a platform, the ground level changed to the highest platform underneath the player. This height was acquired by comparing the player's position and the height of each of the platforms available every time that the game updated. The highest platform underneath the player's feet became the new ground level. If a player walked off of a platform and the ground level changed to a lower one than previous without the player jumping, then the player was affected by gravity. This was implemented by making the player "jump" with a zero vertical velocity which decremented by forces of gravity and created the effect of the player falling.

Head Tracking

In order to implement head tracking, the IR camera on the Wiimote is used. Glasses were made with an IR LED fixed in the middle, above the nose, and an IR LED on each side next to the temples. These formed a triangle in space which the IR camera could pick up.

After calculating the focal length of the camera and the pixel size (in millimetres) of the projected image on the screen, triangulation methods were implemented in order to calculate the distance, angle of elevation and angle of rotation of the glasses with respect to the Wiimote's IR camera. This information was used to manipulate the position of the user in the virtual world.

The data captured of the glasses with respect to the Wiimote was also used to measure the height of the glasses to implement jumping and ducking in the game. This allowed the user to duck to avoid overhanging objects and to jump over obstacles by ducking and jumping in real life.

Results

When the head-tracking system was integrated into the 3D game, the resulting user input proved to perform effectively. The game was rendered at and average of 82 frames per second which was acceptable as the game was smooth to the eye and the head-tracking system was responded fast enough to have a jerk free response to rapid movement of the user's head.

A limiting factor of the system was the area in which the user had to stay in in order for the Wiimote to be able to pick up the IR LED sources on the user's glasses. This area was determined by the 30° emitting area of the LEDs and the 45° receiving range of the IR camera on the Wiimote.

Using head-tracking as a user input was tested and shown to be intuitive for the user to learn and adapt to. Test subjects found this method of interacting with the computer more entertaining than the keyboard and mouse interface, however it was not as easy to achieve accurate control with.

Part II

Implementation of a 3D Game Using the Wii-Remote as an Enhanced User Interface

Chapter 1

Introduction

Since it's release in 2006, the Wii system has captured the imagination of many researchers across the globe. One such researcher, Johny Chung Lee of Carnegie Mellon University in Pittsburgh, developed a system using the Wii remote (Wiimote) to track the position of a users head and use it to change the angle of a 3D rendering on a computer screen to provide the illusion of perspective. He concluded that if this technology were integrated into a computer environment and a second Wiimote were used by the user to input other commands, then the computer gaming industry could be revolutionised.[4]

This project is a study of the feasibility of this function of the Wiimote. It aims at justifying whether it is viable for the Wiimote to be used to enhance the virtual three-dimensional aspects of a computer game by adding an extra sense of perspective and advanced user input and whether the system will have adequate performance. This will be achieved by creating a 3D game which will use the Wiimote to implement head-tracking, virtual world navigation and object manipulation.

1.1 Literature Review

Johny Chung Lee's original head tracking application used one Wiimote which captured light from two infra-red Light Emitting Diodes (LEDs) mounted on his head.[4] This allowed the software to calculate the general direction in which the user was standing with respect to the Wiimote and it altered the perspective on a 3D rendering displayed to the user through a 2D screen. While this gave an improved sense of perspective to the user viewing the virtual 3D image, it lacked the ability to allow the user to navigate through the virtual environment, which is an important functionality in 3D games.

Another downfall of the original software was that it didn't take into account the phenomenon of "gimbal lock". This meant that the software didn't know in which direction the user was facing and, therefore, could not give an accurate change in perspective. This lies in the fact that only 2 IR points were used for head tracking. Using 2 points for triangulation will give an approximate location of the users head as the system would not be able to distinguish between the user moving away from the IR camera, the user turning their head and the user stepping to the side. In order to accurately measure distance and orientation using only 2 points, the user would have to be bound to 1 degree of freedom.

In order to implement advanced head-tracking, the location and orientation of the head need to be known. While the location of the head can easily be computed and described by an (x,y,z) co-ordinate in space, the orientation of the head is more complicated to compute. There exist 3 ways in which it can be computed.

Angle triplets, or Euler Angles, describe a 3D orientation using 3 consecutive rotations around 3 given axes (usually x,y,z). This method suffers from gimbal lock (e.g. if the axes are first rotated by 90° along the x-axis, then the new y-axis and the old z-axis would coincide and gimbal lock will occur). It is also difficult to manipulate and requires multiple evaluations of trigonometric functions. [2]

Unit Quaternions describe a 3D orientation using simultaneous rotations around 4 axes (usually x,y,z and a complex axis w)[3]. This method is more efficient than angle triplets and does not suffer from gimbal lock. [2]

Orthogonal 3x3 matrices are the most efficient method of describing a 3D orientation, however, they have 9 parameters with 6 different constraints between them [2], so implementation of this method is very complex and would require more time than is allowed by the scope of this project.

1.2 Goals and Theory Development

The goal of this project is to improve on Johnny Chung Lee's head tracking implementation of the Wii and integrate it into a 3 dimensional computer game.

Improvements on the original system would include the accurate distance, rotation and elevation tracking of the users head with respect to the Wiimote allowing for 6 degrees of freedom (DOF). Accurate tracking of the head will allow for an alternative user input, for instance, if a user jumps in real life, they will jump in the game.

A further improvement will make use of a second Wiimote as an input device which will allow the user to navigate through the virtual environment.

Therefore, the objectives of this project are to:

- Find a suitable API which can communicate with the Wiimote and Nunchuk via the PC
- Establish how the Wii communicates with its parent device
- Create software for head tracking which can determine position and distance of the user from the Wiimote
- Create a 3D test application to construct a prototype for perspective using the head tracking software
- Create a 3D virtual world which implements the use of the perspective software and the Nunchuk for navigation
- Ascertain the limitations of the virtual world and possible future improvements to study the feasibility of implementing this setup into future gaming development

Chapter 2

Available Materials

2.1 Personal Computer (PC)

The PC which was used had the following specifications[1]:

Hardware

- Processor - Intel Core 2 Duo T7200, 2GHZ dual core processor
- RAM - 2 GB
- Graphics card - NVidia GForce 7900GTX, 512mb

Software

- Operating System - Windows Vista
- Java Compiler - Netbeans IDE v6.1
- Java Development Kit and Runtime Environment - Version 6 update 7
- LyX v1.5.3

[1]Only the specifications relevant to this project are shown

2.2 Bluetooth

The Wiimote connects to its host device via Bluetooth as a human interface device(HID).

2.2.1 Dongle

An X-Micro V1.2 Bluetooth dongle was used.

2.2.2 Stack

A Bluetooth stack is a software implementation of the Bluetooth protocol stack. The stack needs to be tested for connectivity to the Wiimote, stability and HID compatibility. The following Bluetooth stacks exist for windows:

- Bluesoleil
- Widcomm Bluetooth Stack
- Microsoft Bluetooth Stack
- Toshiba Bluetooth Stack

The Bluesoleil and Widcomm stacks both gave errors and crashed the operating system when connected to the Wiimote.

The Microsoft stack was stable but had limited functionality when connected with the Wiimote.

The Toshiba stack was used as it gave the best stability, connectivity and functionality and it was compatible with Wiimote.

2.3 The Wiimote

2.3.1 Sensors

The Wiimote has an ADXL330 Accelerometer, which can pick up up to 3 G's of force, and a PixArt Optical IR camera sensor, which can track the x-y co-ordinates of up to 4 points. These can both be used for user input.

2.3.2 Buttons

The Wiimote has 11 available buttons which can be assigned to different functions.

2.3.3 Rumble Function

The Wiimote has a rumble function which vibrates the remote when activated.

2.3.4 Speaker

The Wiimote has a built-in speaker.

2.3.5 LEDs

The Wiimote has 4 LEDs which are traditionally used by the console to indicate which remote is which when multiple remotes are connected to the system.

2.4 Nunchuk

2.4.1 Accelerometer

The Nunchuk has an ST Microelectronics LIS3L02AL three-axis accelerometer which can measure tilt and G-Force acceleration for use as user input.

2.4.2 Joystick

The Nunchuk's joystick can measure how much and in which direction it has been tilted by the user by returning an angle and magnitude of tilting.

2.4.3 Buttons

The Nunchuk has 2 assignable buttons.

2.5 Wii Sensor Bar

The Wii sensor bar has 2 arrays of 5 infra-red light emitting diodes(LED), separated by a known distance. These LEDs are picked up by the PixArt optical sensor on the Wiimote and, by means of triangulation, the distance and position of the Wiimote, with respect to the sensor bar, is calculated.

2.6 Wii API

Of the available API's (Appendix C), WiiUseJ was used as it proved to be the most compatible with the Wiimote and Nunchuk features and it was the most universal API.

2.7 3D API

Two popular API's exist for 3D software development, OpenGL and DirectX. DirectX is a Microsoft API and only works under Microsoft Windows. OpenGL has lower functionality than DirectX, but it is platform independent, so this project was based on OpenGL for 3D applications.

Since the chosen programming language for the Wii API was java, the chosen API for OpenGL was the Java OpenGL package (JOGL).

Chapter 3

The Wii Application Programming Interface

In order to access the functions of the Wiimote and Nunchuk, an Application Programming Interface (API) must be used. This chapter discusses how the WiiUseJ API responds to inputs from the Wiimote and what is sent to the Wiimote by the API to activate its functions.

3.1 API Requirements

When choosing an API, the following was taken into consideration:

- Platform-Independence - allowing the software to run on different operating systems
- Functionality - accessibility to all of the Wiimote's and Nunchuk's functions
- Multi-connectivity support - it is required that 2 Wiimotes are simultaneously connected to the system for the final software to be developed

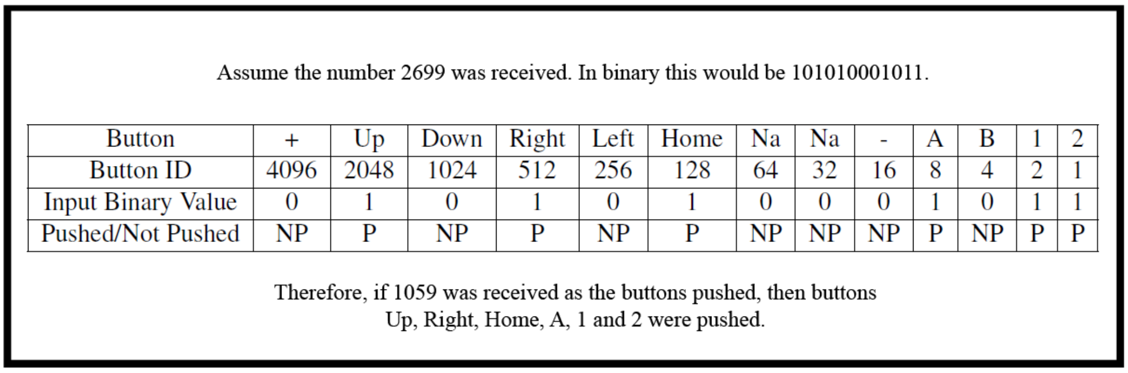
As can be seen in Appendix A, there are a few API's which satisfy these criteria, however, the WiiUseJ API proved to be the most stable and universal API.

3.2 The WiiUseJ API Communications - Wiimote

When an external input event occurs on the Wiimote, the WiiUseJ API generates an interrupt in the form of an Event. The main program thread is paused while the Event is processed. The following sections discuss the manner in which the Wiimote was found to interact using the WiiUseJ API.

Button	ID
2	1
1	2
B	4
A	8
-	16
Home	128
Left	256
Right	512
Down	1024
Up	2048
+	4096

Table 3.1: API button IDs



Assume the number 2699 was received. In binary this would be 101010001011.

Button	+	Up	Down	Right	Left	Home	Na	Na	-	A	B	1	2
Button ID	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
Input Binary Value	0	1	0	1	0	1	0	0	0	1	0	1	1
Pushed/Not Pushed	NP	P	NP	P	NP	P	NP	NP	NP	P	NP	P	P

Therefore, if 1059 was received as the buttons pushed, then buttons
Up, Right, Home, A, 1 and 2 were pushed.

Figure 3.1: Example of finding which buttons were pushed through a binary conversion

3.2.1 Buttons

Pushing a button on the Wiimote causes the API to generate a Buttons Event. Each button has an integer ID assigned to it, as in Table 3.1. If multiple buttons are pushed, the sum of the IDs of the pushed buttons is returned.

It can be seen from table 3.1 that each button has an ID which is an exponential of base 2. This means that if multiple buttons are pushed, a simple conversion of the returned integer to a binary format would reveal which buttons have been pushed (Figure 3.1).

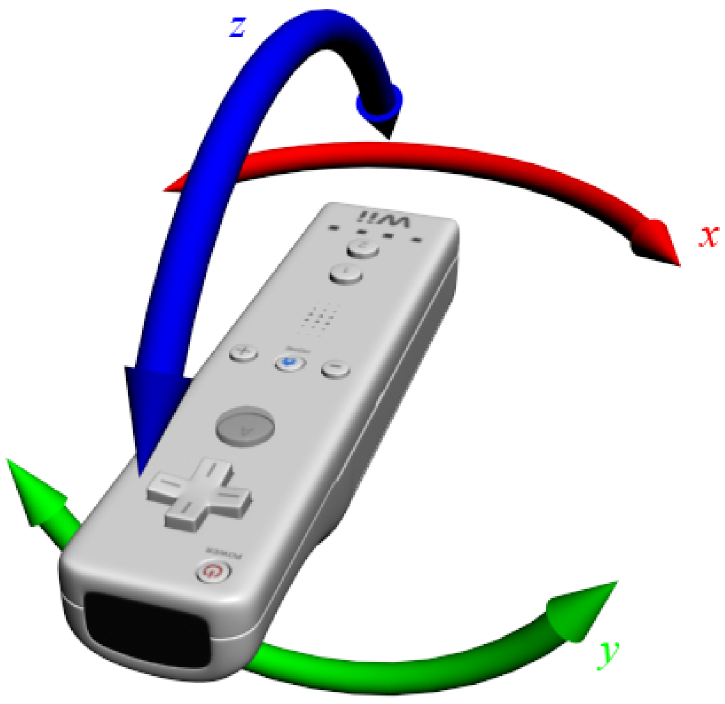


Figure 3.2: Illustration of the effect on the X, Y and Z axes of the accelerometer when the remote is tilted

3.2.2 Accelerometer

If the accelerometer is activated, moving the Wiimote and changing the effective acceleration on any of the 3 accelerometers in the package would cause the API to generate a MotionSensing Event. Through this event, the G-force acting on each of the three accelerometers can be retrieved in the form of a decimal value.

The three axes are perpendicular to each other and are labeled X,Y and Z in the API. The X-Axis measures roll, the Y-axis measures yaw and the Z-axis measures pitch (as in Figure 3.2).

3.2.3 IR Camera

If the IR camera is activated, then changing the relative position of the sensor bar to the Wiimote would cause the API to generate an IR Event. This event returns the raw X and Y co-ordinates in the form of an integer value in the range 0-1023.

Since the resolution of the camera is 1024x768 pixels, this range is appropriate for the X-axis, however, it causes the Y-axis to be stretched. In order to compensate for this stretch, the returned Y-value can be divided by 1024 and multiplied by 768.

If there are multiple IR sources being sensed by the camera (up to 4), then the API returns an array of integer values for the X and Y co-ordinates of each point.

3.2.4 LEDs

The LEDs each have a Boolean value associated to their status. If the Boolean value is true, then the LED is on and if it's false, then the LED is off. These values can be set through the setLeds() method of the API.

3.2.5 Rumble Feature

The rumble feature can be activated and de-activated using simple activateRumble() and deactivateRumble() methods of the API.

3.3 The WiiUseJ API Communications - Nunchuk

When an external input event occurs on the Nunchuk, the WiiUseJ API generates an interrupt in the form of a NunchukEvent. The main program thread is paused while the NunchukEvent thread is processed. The following sections discuss the manner in which the Nunchuk was found to interact using the WiiUseJ API.

3.3.1 Buttons

Pushing a button on the Nunchuk causes the API to generate a NunchuckButtonsEvent. When button Z is pushed, the API returns a value of 1 and it returns 2 when button C is pushed. If both buttons are pushed, the API returns a value of 3.

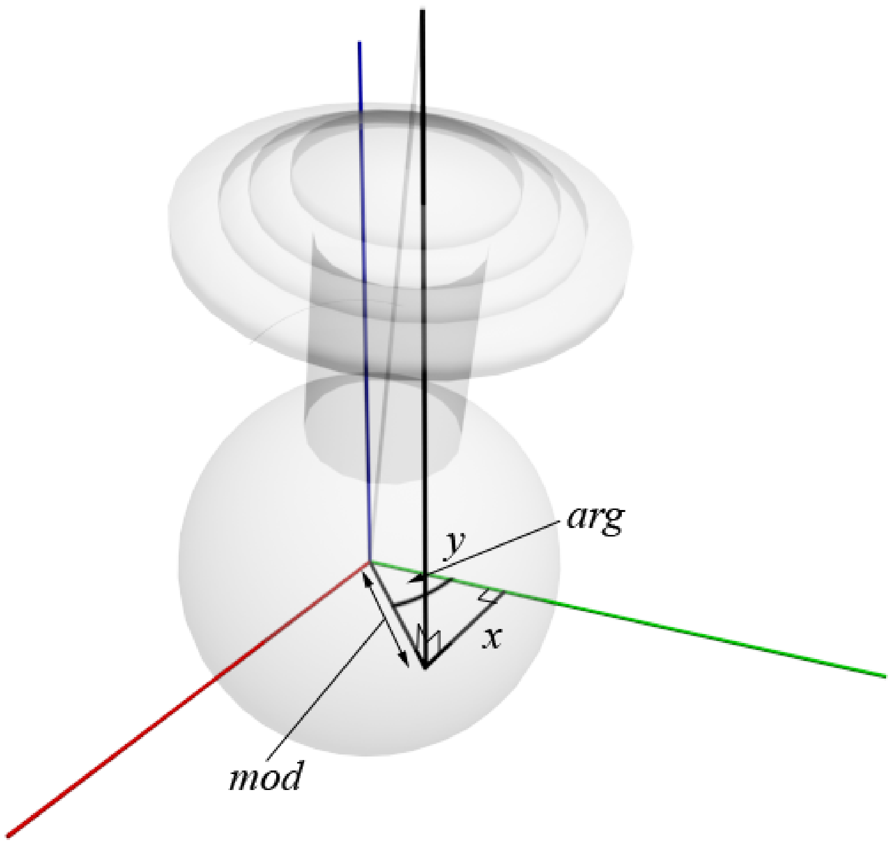


Figure 3.3: Illustration of the joysticks position calculation

3.3.2 Joystick

When the joystick is tilted, the API returns the position of the joystick in a modular-argument form (Radial co-ordinates). The modular value ranges from 0 to 1 and the argument (or angle) ranges from 0° to 360°. Using these values, we can see by how much the joystick has been tilted in the side and forward-backward directions (Figure 3.3). The side movement can be measured by using equation 3.1 and the forward-backward movement can be measured using equation 3.2.

$$x = mod.sin(arg) \tag{3.1}$$

$$y = mod.cos(arg) \tag{3.2}$$

3.3.3 Accelerometer

The accelerometer of the Nunchuk works in the same way and returns the same values as the Wiimote accelerometer (See 3.2.2).

3.4 Results

Using the information discussed in this chapter, the user inputs could be processed to implement the navigation of the user through the virtual world and the IR camera's interaction with the API was the first step to the implementation of head-tracking.

Chapter 4

Virtual World Navigation

Traditional games use the keyboard and mouse as input devices for games, whether it be for navigating through the environment or interacting with objects. For this project, this traditional form of user input will be compared to input using a Wiimote. In order to do this, both forms of input will be implemented and, when tested, the user will navigate through the game using both and compare their experiences.

4.1 Defining the Player's Position

In order to implement any form of navigation,it must first be defined how the program will interpret the player's position in the world. The OpenGL 3D API uses an *x,y* and *z* co-ordinate system to define the virtual world. When the player first enters the world they are placed at the origin of the three axes with the *x,y* plane parallel to the screen and the *z* axis perpendicular[1].

When the player moves through the virtual world, the OpenGL API defines their position using 2 vectors (figure 4.1). The first vector points to the exact position of the player in the virtual world. The vector's *x*,*y* and *z* co-ordinates of this vector are the co-ordinates of the player's head. The second vector defines the direction in which the player is looking. This vector originates at the head of the first vector and points in the direction in which the user is looking. It is of arbitrary length since only its direction is important.

[1]The *x*-axis will be pointing to the left, the *y*-axis upward and the *z*-axis will point into the screen.

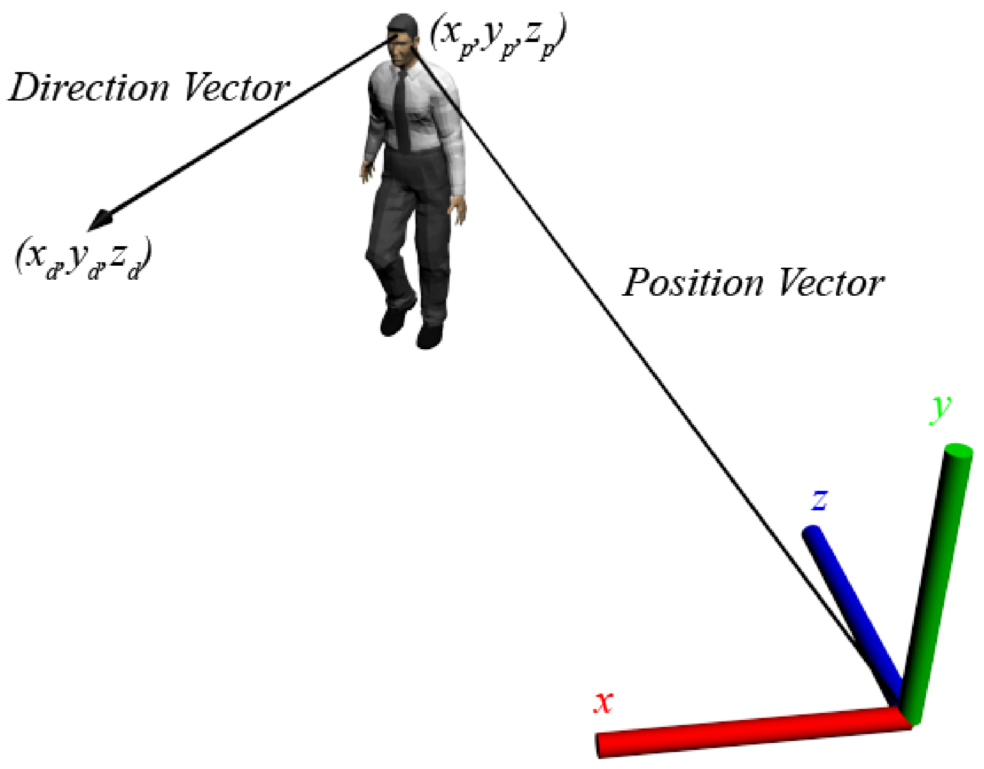


Figure 4.1: The 2 vectors defining the position of the player in the world

4.2 Defining the Navigation Variables

When we think of a person in real life, that person has a specific position, looking direction, movement speed, etc. These are the physical characteristics which define the persons current location in the world and allow us to predict their future location. A person in real life is also affected by the physical laws which governs how objects interact with each other, such as the law of conservation of momentum or gravity. So, in order to navigate through the virtual world, we must define the characteristics of the player and the laws by which they will be affected.

The physical laws which govern the real world are complex and would require a lot more time, than the scope of this project allows, to implement. Therefore the only physical law implemented in the game is that of gravity.

Defining a set of characteristics for the player will give us a means by which we could manipulate the players position in the virtual world. The following characteristics were identified which would be required for the game:

- The forward/backward acceleration of the player - Stored as a decimal value with size dependant on the user input
- The forward/backward speed of the player - Stored as a decimal value acquired from integrating the forward/backward acceleration
- The sideways acceleration of the player - Stored as a decimal value with size dependant on the user input
- The sideways speed of the player - Stored as a decimal value acquired from integrating the sideways acceleration
- The vertical speed of the player - Sored as a decimal value acquired from a combination of the user input and the integration of the gravity of the world
- Jump height - When a player jumps or walks off of a platform, this decimal value stores the height of the player above the ground. This changes with the vertical speed of the player
- Player height - This is a decimal value which stores the height of the player. The height of the player changes when the player crouches or stands up from a crouching position
- The player position - Stored as a point with x, y and z co-ordinates. The x and z values change by integrating the forward/backward and sideways speeds while the y value is a sum of the player height and the jump height

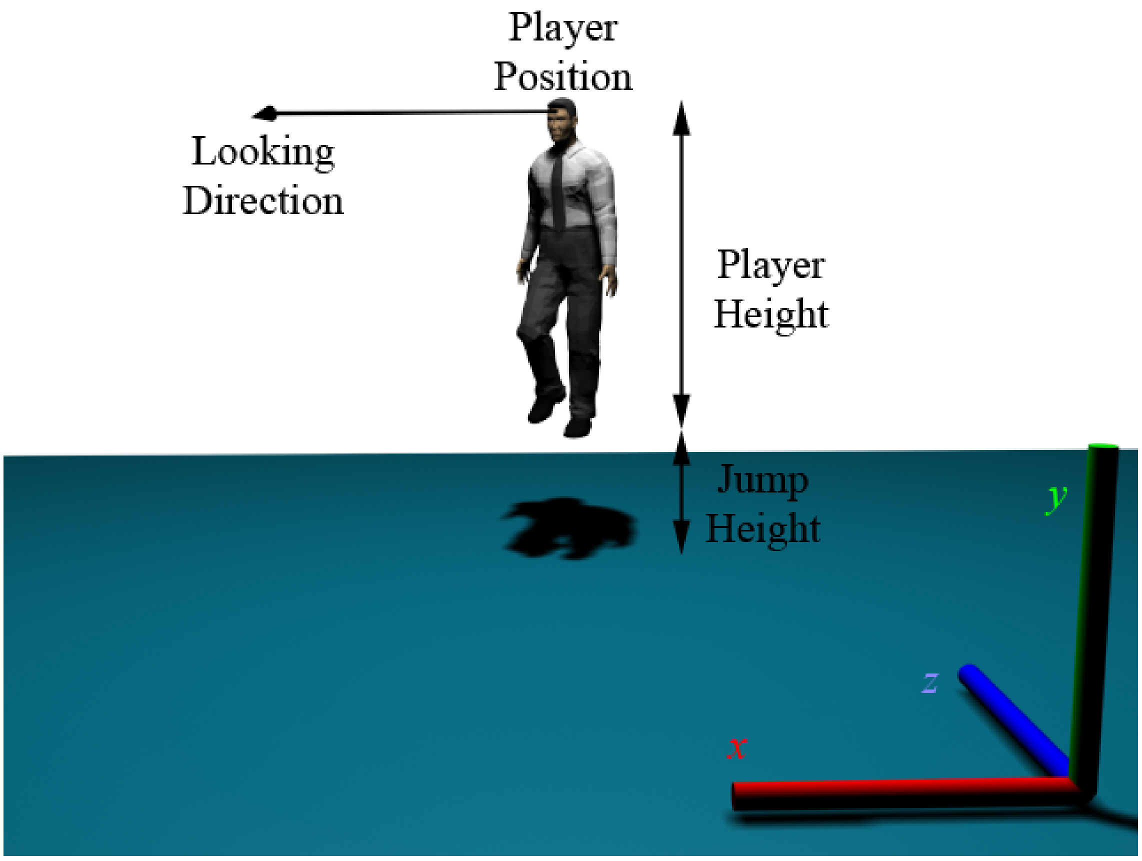


Figure 4.2: Illustration of the variables used to achieve navigation in the virtual world

- The looking direction of the player - Stored as yaw and pitch angles. In order to acquire the looking direction from the player position, the player position point is added to a vector defined using these angles in cylindrical co-ordinates (yaw angle, pitch angle and length)

4.3 Implementing Traditional (Mouse and Keyboard) Controls

4.3.1 Movement in the Horizontal Plane

Since the horizontal position of the player is dependent horizontal speed, and the speed is dependent on the acceleration, in order for the user to move around the virtual world, they would only need to control the acceleration.

When the player pushes the key for forward movement, the forward acceleration is set to a fixed value, greater than zero. The forward speed now begins to increase as it is the integral of acceleration.

When the desired forward speed is reached, the acceleration is set back to zero and speed stays at a constant value. Once the speed begins to increase, the player's position changes by the integral of the speed and the player begins to move in the direction that they are looking (specified by the yaw angle as discussed in section 4.2). This is achieved by incrementing the x and z values which define the player's position by the integral of the speed in the direction of the direction vector (figure 4.1).

When the forward key is released, the acceleration is set to a fixed value less than zero. The speed decreases and when it reaches zero, the acceleration is set back to zero. The player keeps on moving in the direction which they are facing until the speed has reached zero.

The above description also applies when the player pushes the backward movement key, but the acceleration, speeds and change in positions are of opposite signs.

When moving sideways (either left or right key is pushed), then the same as above applies, except the motion is not in the direction of the direction vector (figure 4.1), but perpendicular to it.

4.3.2 Jumping (Vertical Movement)

The vertical position of the player is dependent on the jump height of the player which is dependent on the vertical speed, which in turn is dependent on the vertical acceleration. Since the vertical acceleration is a fixed value for gravity in the virtual world, the value which the user would need to influence when jumping would be the initial vertical speed.

When the jump key is pushed, the vertical speed is set to a value greater than zero. Since the jump height is the integral of the vertical speed, it starts to increase and the player moves upward. This is achieved by incrementing the y value defining the player's position. The fixed gravitational acceleration begins to decrease the vertical speed until it reaches zero and the jump height reaches a peak. The vertical speed then begins to decrease to a negative value and, hence, the jump height begins to decrease. When the jump height reaches zero, the vertical speed is set back to zero and the jump is over.

4.3.3 Crouching

When crouching, the player height is the only thing which changes. When the player pushes the crouch button, the player height begins to decrease until it reaches a predefined minimum height. When the crouch button is released, the player height increases until it reaches it's original predefined maximum height. Since the y co-ordinate of the player position is a sum of the current ground level, jump height and the player height, it will appear that the player is crouching or standing.

4.3.4 Looking Around

Looking around the virtual world is achieved by the use of the mouse. When the game begins, the mouse is moved to the centre of the screen. When the mouse is moved, the change in pixels in its vertical and horizontal position on the screen are noted. The vertical change changes the pitch angle and the horizontal change changes the yaw angle (pitch and yaw angles as discussed in4.2). Once the angles have been incremented, the mouse position is set back to the centre of the screen.

4.4 Implementing Wii Controls

4.4.1 Movement in the Horizontal Plane

This movement is controlled by the 'Up', 'Down', 'Left' and 'Right' keys on the Wiimote. These have the same effect as the keyboard keys as discussed in section 4.3.1.

4.4.2 Jumping (Vertical Movement)

When using the Wiimote, jumping is not controlled by a button but rather using the head-tracking. The head-tracking module of the program (Chapter **??**) can calculate the position of the user's head relative to the stationary Wiimote. When the game begins, the user must stand still facing the Wiimote and in a standing position. The program then calculates the user's height and stores it.

If the user physically jumped in real life, then the height position of their head would be higher than the user's normal standing height. The program would then know that the user has jumped in real life and would respond by making the character jump in the game as if the jump key were pressed on the keyboard.

4.4.3 Crouching

As with jumping, crouching will be controlled using the head-tracking module of the program. Using the same principle, the user's height will initially be stored and if the user crouches in real life, then the program will calculate the height of the users head to be lower than the standing height of the user and it will respond by decreasing the height variable in the game.

4.4.4 Looking Around

Looking around will be achieved by the use of the Nunchuk's joystick. As discussed in section 3.3.2, when the joystick is tilted, it is possible to separate the magnitude of the tilt into an x and y magnitude. When the joystick is tilted, the x magnitude of the joystick will be used to change the yaw angle (as discussed in4.2) and the y magnitude will be used to change the pitch angle. So when the joystick is tilted sideways, then the looking direction of the player will be rotated horizontally and if the joystick is tilted backward or forward, then the looking direction will move up or down respectively. If a combination of sideways and backward/forward tilting occurs, then the looking direction will change in a combination of rotating horizontally and looking up/down.

4.5 Results

Using the methods discussed in this chapter, navigation in the virtual world was successfully implemented with both the keyboard and the Wiimote. Applying accelerations when moving instead of instantaneous velocity changes gave a sense of much smoother movement, as did the gradual decrementing of height when crouching as apposed to an instant change in height.

Chapter 5

Implementation of the Virtual World

5.1 Creating the Virtual Environment

The virtual world consists of one room which has 4 corridors (figure 5.1). The user starts in the bottom right corner of the room in the first corridor which consists of 3 low lying platforms and 2 overhanging platforms. In order to get through he corridor, the user must jump over the low platforms and duck under the high ones. This corridor serves to compare the difference between the jumping and ducking functions using the keyboard input and the Wiimote input.

The second corridor has a constant stream of small spheres which travel from the back wall to the front. The user must navigate through the spheres which will push them back if the user collides with a sphere. This corridor is used to compare the 3D perspective effect with and without head-tracking.

The third corridor contains an array of 6 boxing gloves which extend from the front wall to the back. as the user moves through the corridor, if they collide with a glove, then they are pushed back. This is used to compare dodging using the traditional keyboard inputs and by using head-tracking, which is achieved by the user physically moving to the side in the real world.

The final corridor consists of 5 platforms which move in and out of the wall. The user needs to jump from platform to platform when the platforms are extended. This is used to test the feasibility and ease of jumping using head-tracking compared to keyboard input.

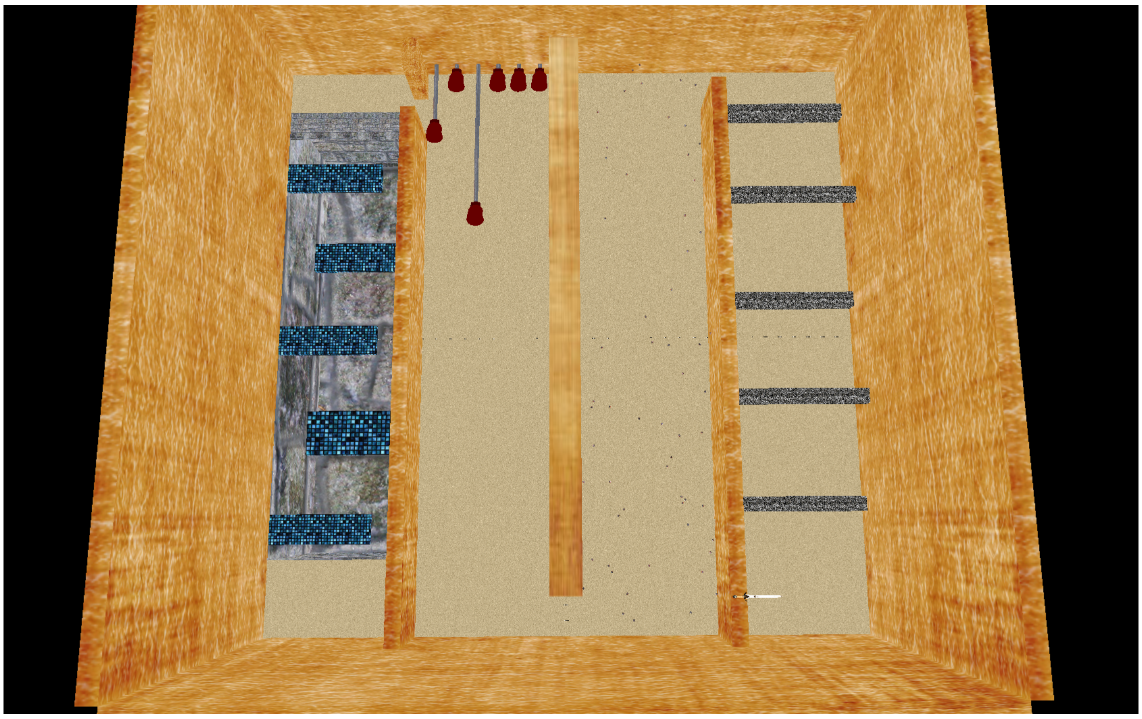


Figure 5.1: Screen capture of the 4 corridors in the virtual room.

5.1.1 Rendering Walls

The data for the walls in the room was stored in a text file. In this way, adding extra walls was simplified as it just required the data for the new wall to be appended to the file. Each wall had the following data stored (figure 5.2):

- Minimum x-value (x_{min})
- Maximum x-value (x_{max})
- Minimum y-value (y_{min})
- Maximum y-value (y_{max})
- Minimum z-value (z_{min})
- Maximum z-value (z_{max})
- The texture image to be used for the wall

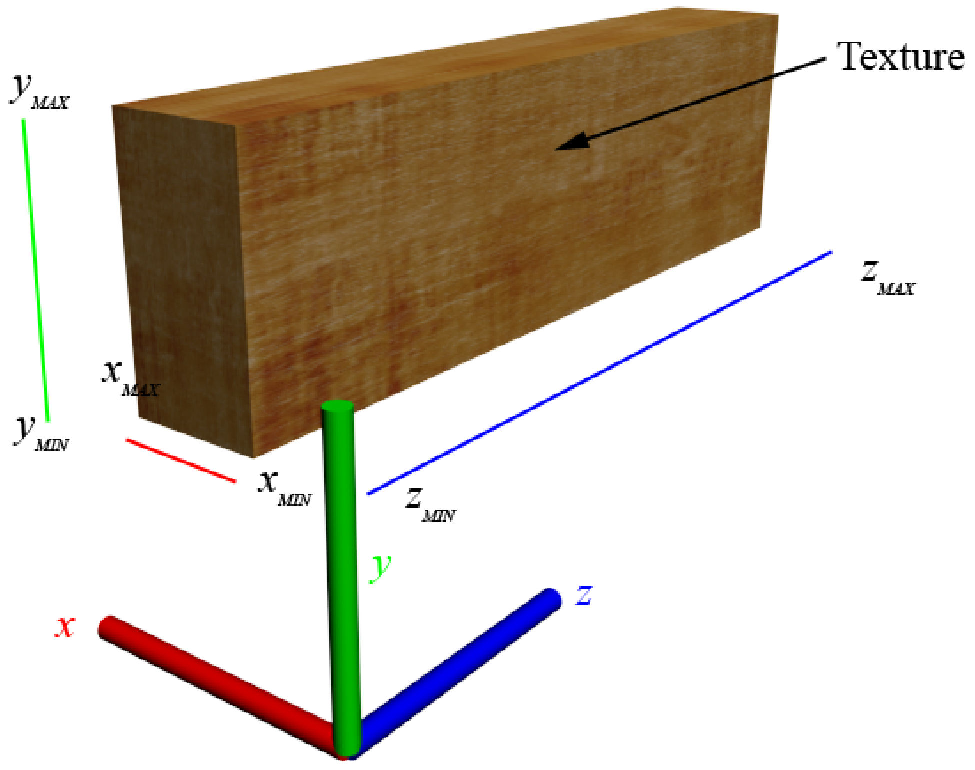


Figure 5.2: The 7 parameters which define a wall

When the program initialised, each of these variables were stored in an array whose length was equal to the number of walls. When a wall was rendered, it was rendered as 6 quadrilaterals (one per side of the wall) using the values from the array to determine the 8 vertexes of the wall. These were:

$$(x_{min}, y_{min}, z_{min}) \text{ ; } (x_{max}, y_{min}, z_{min}) \text{ ; } (x_{min}, y_{max}, z_{min}) \text{ ; } (x_{max}, y_{max}, z_{min}) \text{ ; } (x_{min}, y_{min}, z_{max}) \text{ ; } (x_{max}, y_{min}, z_{max}) \text{ ; } (x_{min}, y_{max}, z_{max}) \text{ ; } (x_{max}, y_{max}, z_{max})$$

5.1.2 Rendering Obstacles

Obstacles were stored as an object file (.obj) with an associated material file (.mtl). The object file defines each vertex of the object while the material file maps points in an image file to each vertex defined by the object file. Each object file is read into an OBJModel variable which converts all of the vertexes and their texture mapping into an array of faces which the OpenGL API can easily render.

When rendering, the position of each obstacle was described by a point with *x,y* and *z* variables and a further three rotation variables to define its orientation by rotations around each of the axes.

5.1.3 Rendering Moving Platforms

Moving platforms are stored in the same way as walls, but in a separate text file. Unlike the static walls, each moving platform has one value of data (e.g. z_{min}) which is dynamic and which the program increments and decrements at a specific rate, effectively moving 4 co-planar vertexes further away or closer to the opposite vertexes (figure 5.3). The platforms are rendered in the same way as the walls.

5.2 Collision Detection

When the user walks around the virtual world, they may hit an obstacle or wall. Without collision detection the user would walk through these walls and obstacles, so a collision detection algorithm was needed to create a more realistic environment in which the user is affected by walking into objects.

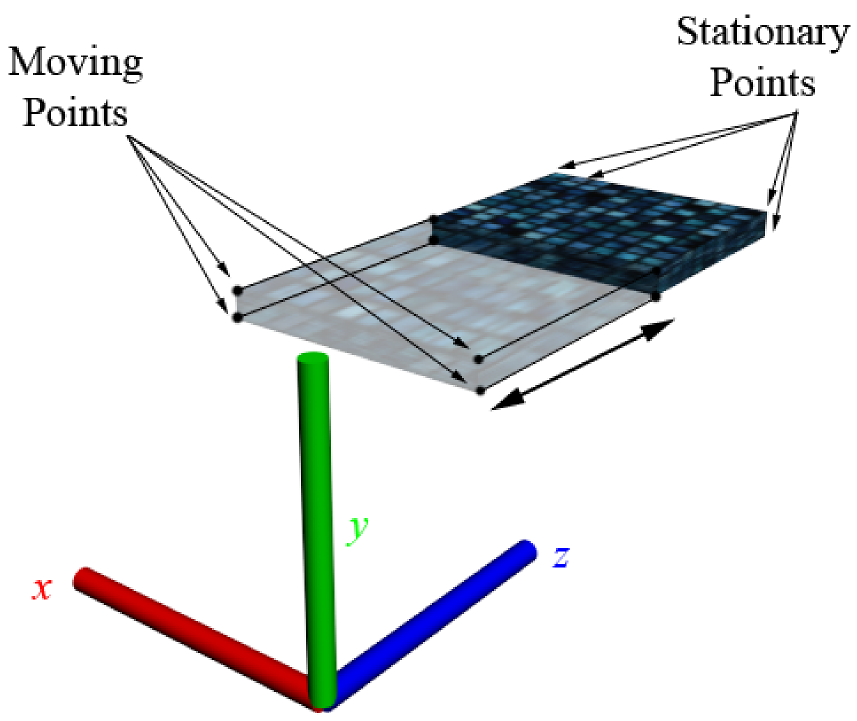


Figure 5.3: A moving platform has 4 co-planar points which move away from or toward the opposite 4 points in the block

5.2.1 Sideways Wall Collision

As discussed in Chapter 4, the position of the player's head in space is described using an *x*-*y*-*z* co-ordinate. In the collision detection section of the program, a second point is defined, the position of the player's feet. This point lies directly below the point which describes the head-position by a distance of the player height variable (Section 4.2).

When the user moves in the virtual environment, the values of these points change to define the new position of the player. Before the position of the player is updated, the program first checks if the new position of the player would be inside a wall.

There are 4 possible ways in which a player can collide sideways with a wall (figure 5.4):

1. the wall starts from the ground and is taller than the player, then both the head and the feet will collide with it
2. the wall starts from the ground but does not reach the player's head (e.g. a step), then only the feet will collide with it
3. the wall starts higher than the ground but below the head and is higher than the head, then only the head will collide with it
4. the wall starts higher than the feet and ends lower than the head, then the "torso" of the player will collide with it

Detecting these four types of collisions was achieved using one method defined by 2 criteria. Firstly, in all 4 types, if a collision occurred, then the new head position point would lie directly above or inside the volume of the wall (Described by the 8 points in section 5.1.1) with which the player collided. A further similarity between the 4 types of collisions is that the head position point would lie higher than the lowest point of the wall, and the feet position point would lie lower than the highest point of the wall (figure 5.5). When the player moves, before the position of the player is changed, the new head and feet positions points are checked according to these criteria and if both criteria hold true, then a collision is detected and the player position is not allowed to change, hence the player stops moving just before they hit the wall.

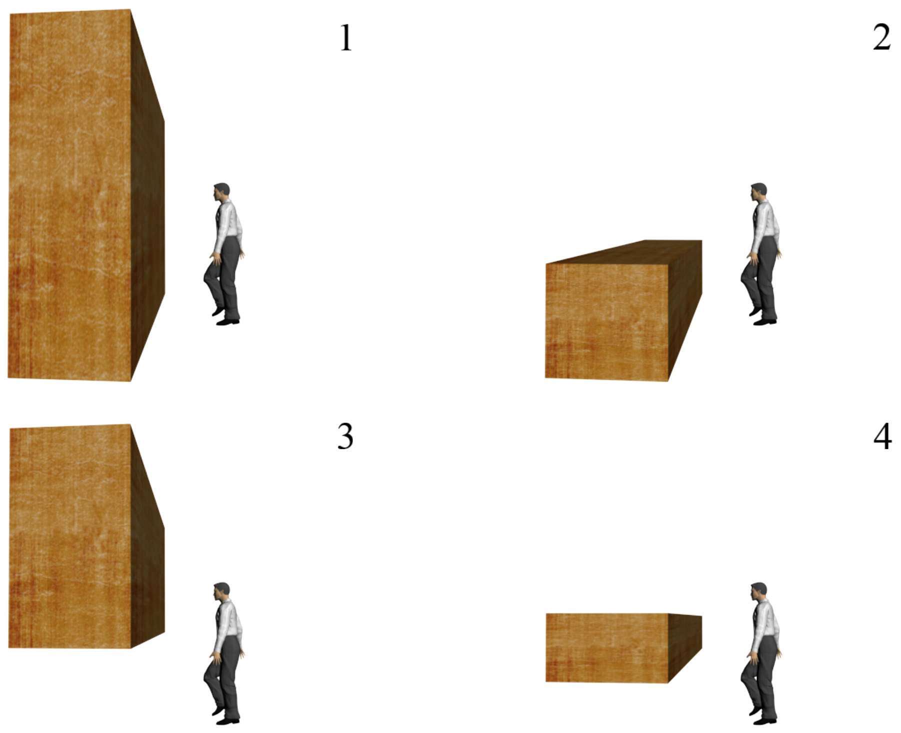


Figure 5.4: Illustration of the 4 ways in which a user can collide with a wall

5.2.2 Overhanging Wall Collision

If a player crouches, moves under an overhanging wall and tries to stand up or they jump into an overhanging platform, the program needs to detect the collision and stop the player from standing up further or jumping higher.

In detecting such a collision, it is noted that a collision will occur if the head position point enters the volume enclosed by the wall. To differentiate between an overhanging wall collision and a sideways wall collision, it is noted that the head position point lies directly below or above the volume of the wall before the collision occurs.

When such a collision is detected, the software notes that the user has hit their head by changing a Boolean variable to 'true.' When this variable is set to true, the software no longer updates the player's height for standing up and, if the player is jumping, sets the jumping velocity to zero resulting in the player dropping back down to ground level.

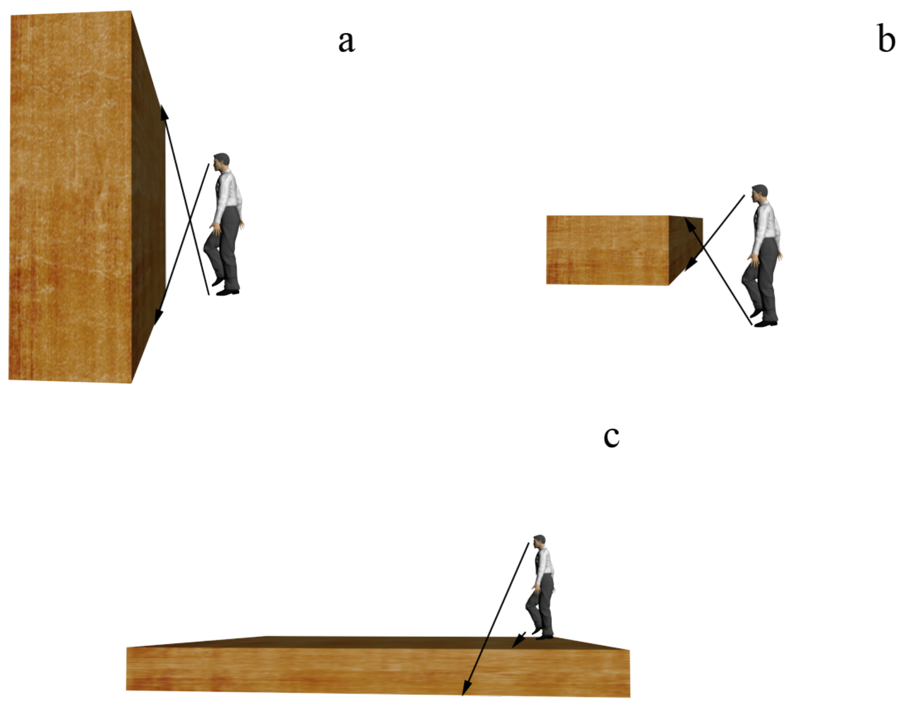


Figure 5.5: Illustrations of collision detection using the 2 collision criteria. Only the second criterion is illustrated in (a) and (b). The first criterion can be seen in (c) as the player's head is above the volume of the wall.

(a) Full body collision detected, (b)Torso collision detected (c) No collision detected as the second criterion does not hold.

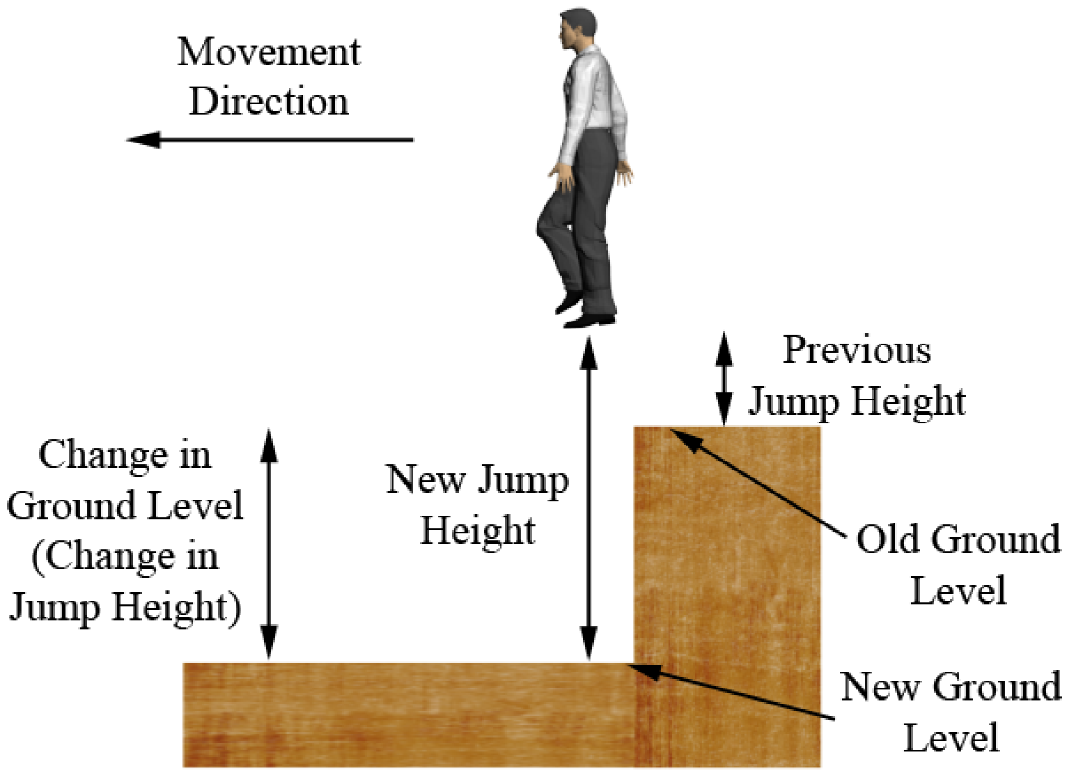


Figure 5.6: The change in the jump height when the ground level is altered

5.2.3 Floor Level Change

When moving around the room, the player will encounter platforms on which they can stand on and holes in which they can fall into. The software needs to recognise when either of these situations occur and respond accordingly.

As noted in Chapter 4, the y value of the head position point is the sum of the ground level, the player height and the jump height. When the player jumps onto a platform or falls into a hole, firstly, the ground level must change and, secondly, the jump height of the player must be altered to correspond to the new ground level.

Every time that the game updates (before rendering a frame) it checks what the current ground level is. This is done by looping through all of the wall y_{max} values and changing the ground level to the biggest y_{max} value which lies under the players feet position point. If the ground level changed, then the jump height is altered accordingly by adding the difference between the new ground level and the old ground level (figure 5.6).

5.2.4 Obstacles

There were 2 obstacles in the game, the spheres in the second corridor and the boxing gloves in the 3rd corridor. Both of the obstacles in the game have a different structure and shape, therefore each object was treated differently.

Spheres

When detecting a collision with one of the spheres in the second corridor, the sphere was treated as a mathematical sphere with a known centre and radius and the player was treated as a cylinder with known lid centre, radius and height.

Since both the radii of the player's cylinder and the sphere were known, then if the distance from the centre of the sphere and the centre of the cylinder was less than the sum of the radii of the two, then a collision had occurred (figure 5.7a).

When such a collision was detected, the following steps were taken:

- The difference of the sum of the radii of the sphere and cylinder and the actual distance between the centre of the sphere and the centre of the cylinder was calculated. This gave the distance the the player would need to be displaced (figure 5.7b).
- The players position was moved by the distance calculated in the previous step so that the sphere and the cylinder lay tangent to one another (figure 5.7c).

Boxing Gloves

For collision detection purposes, the boxing gloves in the third corridor were treated as cuboids (six sided polyhedron with rectangular faces) and the player was treated as a cylinder with a known radius and centre line.

To detect a collision, the distance between the centre of the cylinder and the walls of the cuboid was calculated and if it was less than the radius of the cylinder, then a collision had occurred. Dealing with a collision involved the same steps as with the spheres (Section 5.2.4), except that instead of using the centre of the shape (as with the sphere) to calculate the required player displacement, the perimeter of the cuboid was used and the player was moved to a position such that the cylinder surrounding the player was tangent to the cuboid (figure 5.8).

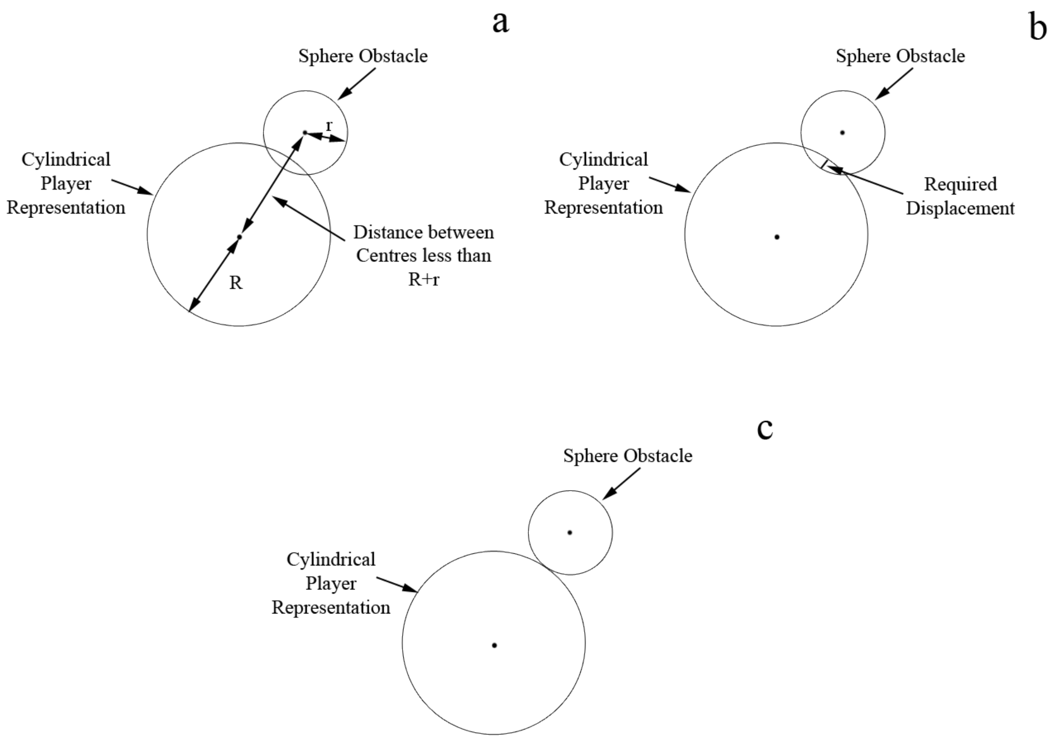

a
Sphere Obstacle
r
Cylindrical Player Representation
Distance between Centres less than R+r
R
b
Sphere Obstacle
Cylindrical Player Representation
Required Displacement
c
Sphere Obstacle
Cylindrical Player Representation

Figure 5.7: Top view representation of a collision of the player with a sphere. (a) Collision detected (b) Required displacement calculated (c) After corrective steps were taken

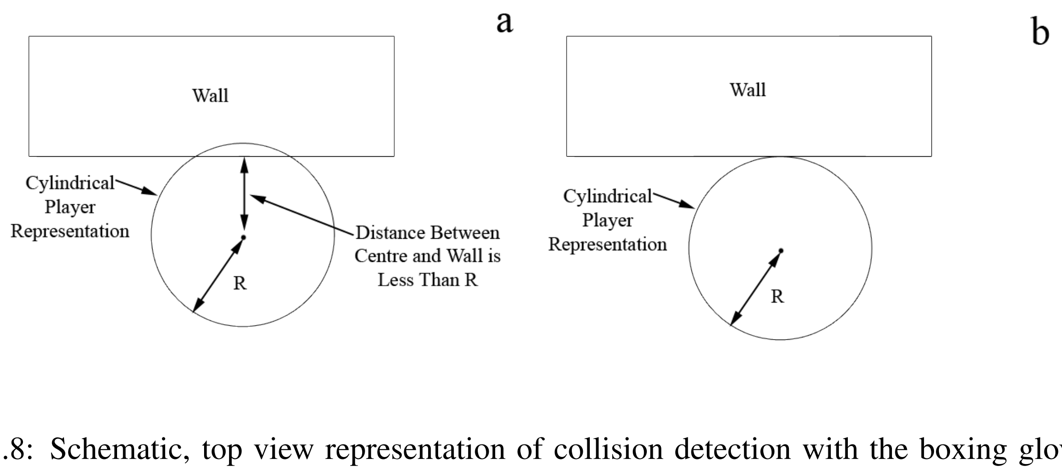

a
Wall
Cylindrical Player Representation
Distance Between Centre and Wall is Less Than R
R
b
Wall
Cylindrical Player Representation
R

Figure 5.8: Schematic, top view representation of collision detection with the boxing gloves. (a) Collision is detected (b) Position of player after processing

5.3 Item Manipulation

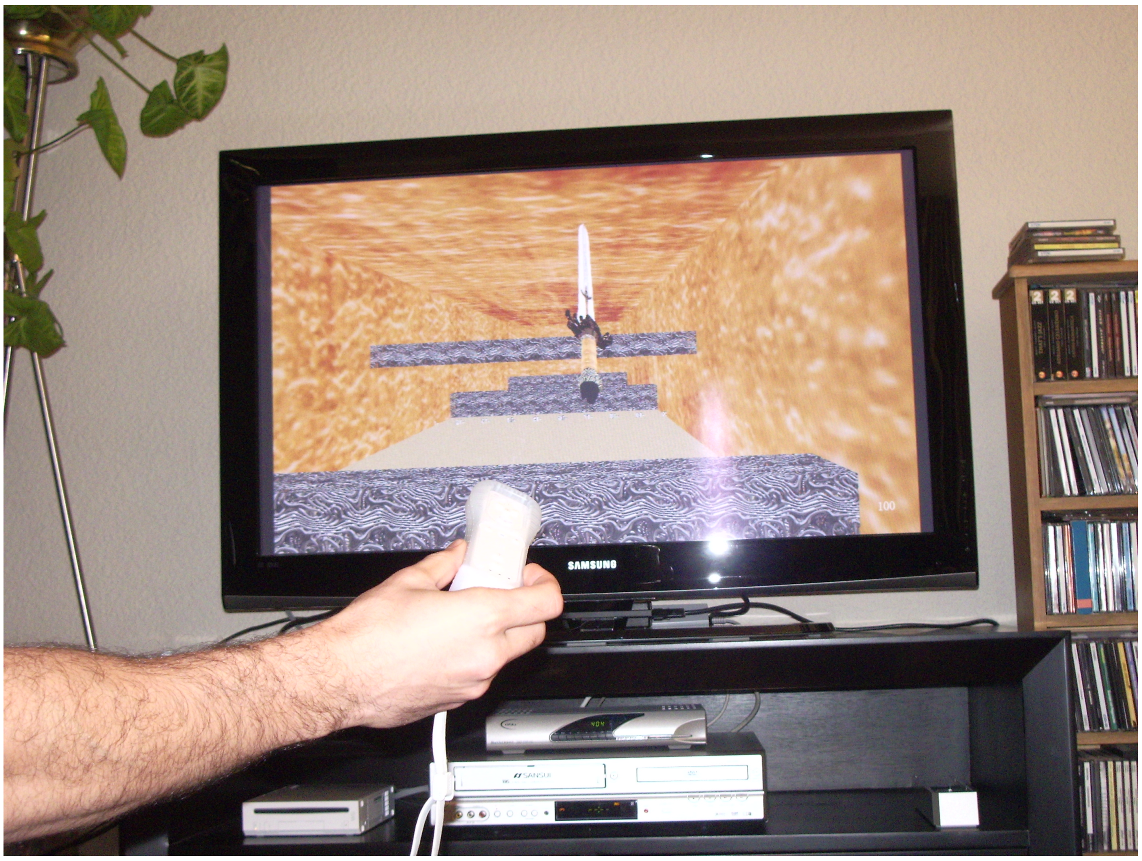
Using the accelerometers on the Wiimote as tilt sensors, the tilt in the *x*,*y* and *z* direction of the Wiimote were measured and used to manipulate the orientation of items in the game.

One item, a sword, was implemented to test this feature and it was saved and imported in the same way as the obstacles. When rendered, its position described by a point with *x*,*y* and *z* variables. This point was located in front of the player. Just as the obstacles, a further three rotation variables to define the orientation of the item by rotations around each of the axes. These three values were calculated from the accelerometers.

The accelerometers output the acceleration of the Wiimote in the *x*,*y* and *z* directions. When the Wiimote is stationary or rotating slowly, the accelerometers act as tilt sensors outputting only the acceleration exerted on them by gravity. These accelerations are transmitted to the computer and returned by the API in the form of a decimal number corresponding to the amount of G's experienced by each axis.

When dealing with gravity, this number is in a range of -1 to 1 for each axis. Multiplying the given number by 180° will give the approximate rotation of the Wiimote around each axis. These values are then used to manipulate the orientation of the held object in the game. The result is that the object in the game faces the same direction as the Wiimote in real life (figure 5.9).

Each axis represents either rotation of pitch, roll or yaw. The pitch and roll rotation axes are both perpendicular to the direction of gravity, while that of yaw is co-linear. Since the accelerometers act on gravity, when the Wiimote is rotated about the yaw axis, on which gravity acts, the accelerometers cannot pick up the rotation as there is no change in the direction of gravity, therefore yaw rotation isn't picked up and cannot be implemented with the accelerometers. Figure 5.10 shows the Wiimote with a pitch, roll and yaw rotation and only pitch and roll rotations are picked up while yaw is not.



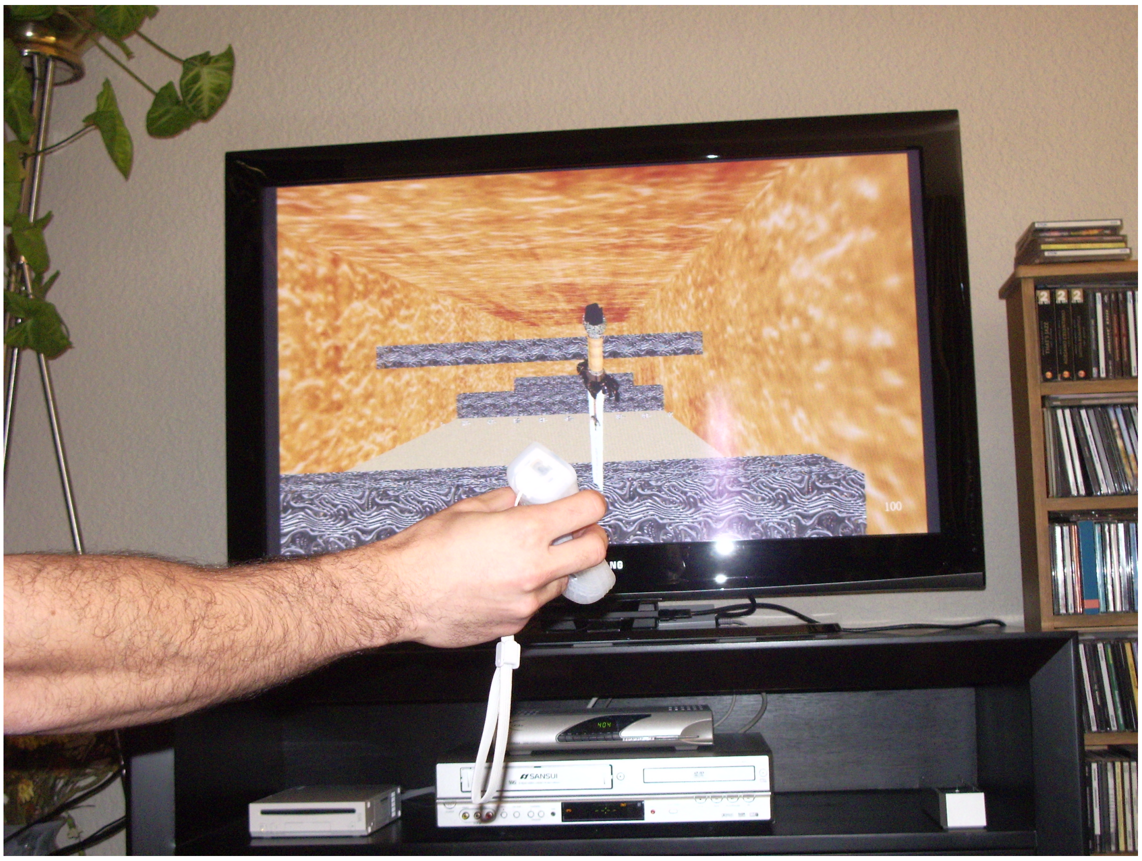


Figure 5.9: The orientation of the object in the game corresponds to the orientation of the Wiimote

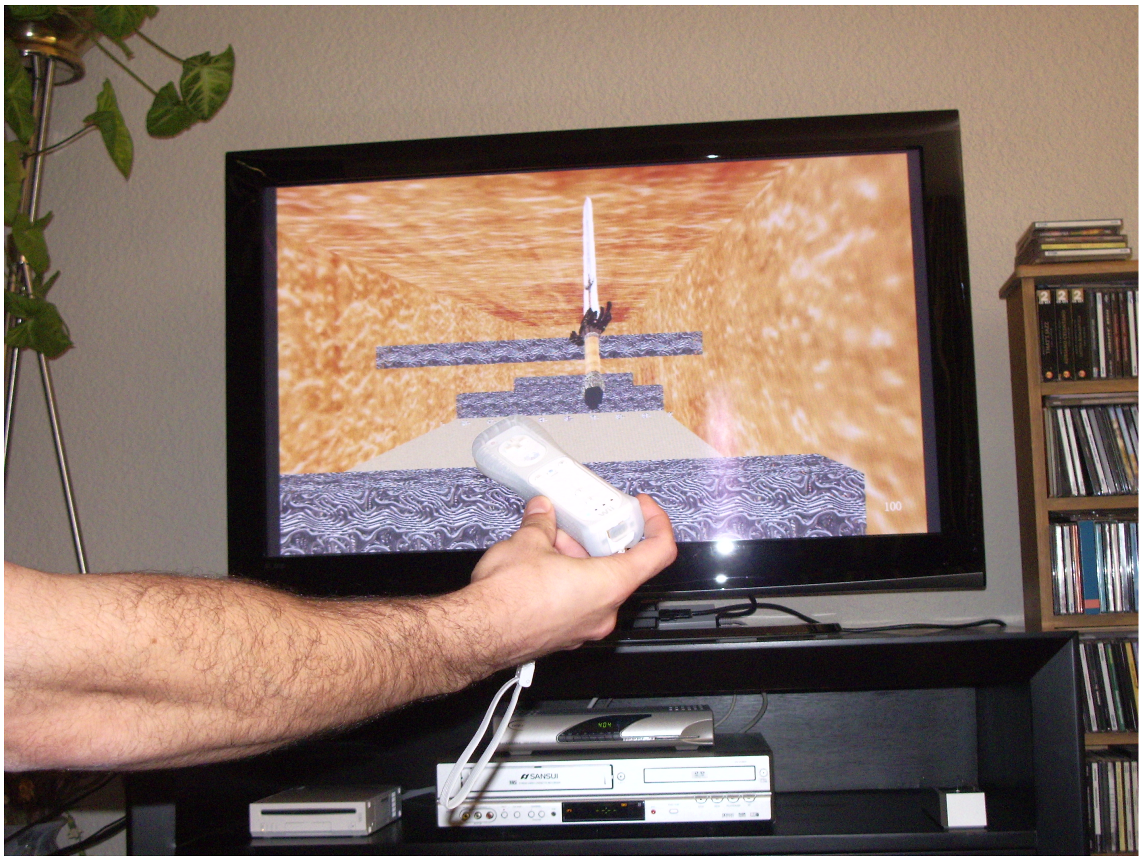


Figure 5.10: Photograph of item manipulation with a yaw rotation which cannot be measured and, hence, isn't depicted in the in-game object

5.4 Results

The methods used to implement the walls and objects proved to be successful. Walls and objects were easy to add or remove and the collision detection was automatically added with each item.

The collision detection for the walls and spheres worked as expected. However, the boxing glove collision detection was problematic as it used an iterative method for detecting collisions (collisions were checked one by one with each glove). This resulted in some collisions pushing the player into areas which would be in the direct vicinity of other boxing gloves whose collision detection had already been processed and, hence, the player sometimes ended up inside a boxing glove.

Chapter 6

Triangulation and Head Tracking

Triangulation is a method of determining the distance to a point using a known reference point and known angles. Using the IR camera on the Wiimote as the known point and the 3 IR LEDs on the glasses on the user's head as the floating points, we can determine the distance from the stationary Wiimote to the user's head. With some further calculations, it is also possible to determine the location in space of the user's head with respect to the Wiimote. This data would be required to perform accurate head-tracking to implement the enhanced 3D perspective. This chapter deals with achieving accurate triangulation.

6.1 Camera Parameters

In order to implement accurate triangulation, we need to know the intrinsic and extrinsic parameters of the IR camera. The intrinsic parameters are those which are attributed to the camera and do not change as the camera moves in space (the cameras internal parameters). The extrinsic parameters are those which change as the camera moves, such as 3D position and orientation of the camera.

6.1.1 Intrinsic Parameter Calculations

There are 3 important intrinsic parameters which are required for triangulation; the pixel size of the camera, the focal length of the camera and the center of projection of the camera[2]. These parameters exist in a ratio of one to another, so in order to calculate them, we have to set a value for one of them and calculate the other two.

The camera is known to have a resolution of 128x96 pixels. This is interpolated by hardware to 1024x768 resulting in a 1024 by 768 pixel matrix used by the camera to pin-point the co-ordinates of the IR source on the screen. The center of projection of the camera is at the center of the screen, therefore for this camera it will be at the point (511,383), assuming that the x-co-ordinate range is [0:1023] and the y-co-ordinate range is [0:767].

Now all that is left is to find the pixel size of the camera in millimeters and the focal length of the camera.

What is the Focal Length?

The way an average digital camera works is as follows:

- External light is reflected by the objects which are being photographed
- This light passes through the camera's lens
- The light is concentrated, by the lens, onto and image sensor
- The image sensor has many photo diodes which conduct differently depending on how much light they are exposed to
- The amount of current that the photo diode conducts is converted to a digital value and stored in an image file

The image which falls onto the lens is a replica of the final output image, therefore we can view the lens as the image plane. Looking at figure 6.1, the focal length, *t,* is the distance between the lens, *BC,* and the image sensor, *A*. If we view the image sensor as a point behind the lens, then it would be the point at which the light from the lens is concentrated (focal point). So, if we make a photo of 2 objects, *DE*, and we know the distance *u* between them, if we know how far apart they are on the photo, distance *v*, and we know the focal length of the camera, *t*, then we can calculate the distance from the camera to the objects, *w*, by using simple trigonometry.

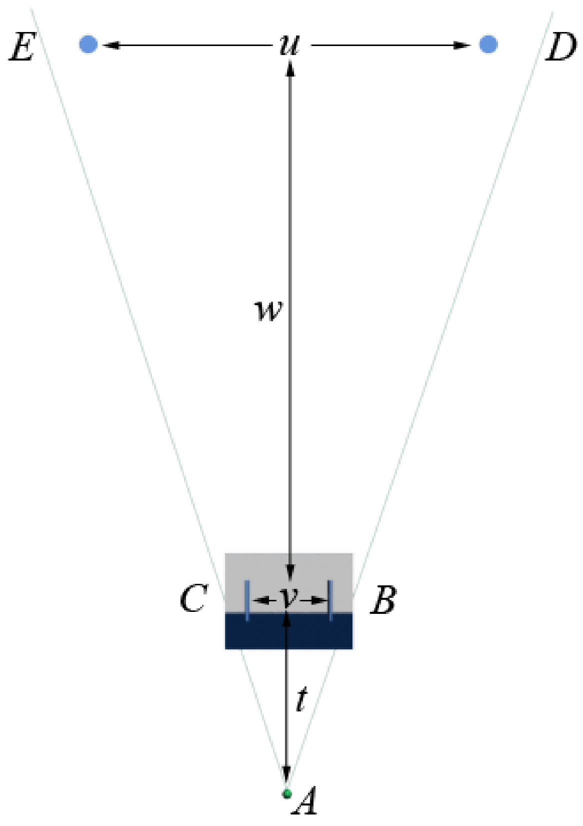


Figure 6.1: Illustration of the relationship between focal length and the distance of 2 objects, of a known distance apart, to the camera

What is Pixel Size and why do we need it?

If we look at a 17” computer screen with a set resolution of 640x480 pixels and we look at an image with a resolution of half the screen size (320x240), then the width of the image on the screen will be half of the width of the screen. Now if we look at the same image on a 19” screen with the same resolution (640x480 pixels), then the same would hold true. However, although the resolution of the screens is the same and the pixel size of the image is the same, the image will appear physically bigger on the 19” screen than on the 17” screen. This means that there is no direct relationship between measuring in pixels and measuring in millimeters.

When we take a photo of 2 objects a known distance *u* apart (as in figure 6.1), and we want to find the distance *w* between them and the camera, we need to know the distance *v* of the objects on the image in millimeters. Since the image produced by the camera is represented in pixels, we need to know what physical size each pixel represents on the image plane.

Calculating the Focal Length and Pixel Size

In order to calculate the pixel size and focal length of the camera, I set up the following experiment:

- Set up 2 IR LEDs 143mm apart
- Place the camera perpendicular to the point between them 500mm away
- Obtained the distance between the points on the captured image (in pixels)
- Moved the camera to a distance of 1000mm(double the original distance) away from the point between the LEDs
- Obtained the new distance between them on the captured image (in pixels)

Following these steps, the values obtained for the distance between the LEDs on the image in pixels was 380pxls when the camera was 500mm away from the LEDs and 192pxls when the camera was 1000mm away.

With this information, it is possible to calculate the pixel size and the focal length using trigonometry. If we look at figure 6.1, we can associate the above acquired values into the figure as follows:

- The fixed distance between the 2 IR LEDs is equal to *u* (*u*=143mm)

- The perpendicular distance between the camera lens (image plane) and the mid-point between the LEDs is w ($w_1 = 500mm$ and $w_2 = 1000mm = 2 \times w_1$)
- The distance between the LEDs in the captured image is equal to v ($v_1 = 380pxls$ and $v_2 = 192pxls$)
- The focal length is equal to t (distance between the image plane and the focal point)

Again, looking at figure 6.1, we can see that triangle ADE is similar to triangle ABC. This means that the sides of the triangles are proportional to each other. The following relationships can therefore be derived:

$$\frac{v}{u} = \frac{t}{t+w} \tag{6.1}$$

$$v(t+w) = ut \tag{6.2}$$

$$w = \frac{ut - vt}{v} \tag{6.3}$$

or

$$t = \frac{wv}{u-v} \tag{6.4}$$

Substituting $w_1, w_2, v_1, v_2$and u into equation 6.3, gives us 2 simultaneous equations with 2 unknowns (distance t and the ratio of millimeters per pixel):

$$w_1 = \frac{ut - v_1 t}{v_1} \tag{6.5}$$

$$w_2 = 2 \times w_1 = \frac{ut - v_2 t}{v_2} \tag{6.6}$$

Substituting equation 6.5 into 6.6 gives us:

$$2 \times \frac{ut - v_1 t}{v_1} = \frac{ut - v_2 t}{v_2} \tag{6.7}$$

$$2 \times \frac{u - v_1}{v_1} = \frac{u - v_2}{v_2} \tag{6.8}$$

$$2.v_2(u - v_1) = v_1(u - v_2) \tag{6.9}$$

$$2.v_2 u - v_1 u = 2v_1 v_2 - v_1 v_2 \tag{6.10}$$

$$u = \frac{v_1 v_2}{2.v_2 - v_1} \tag{6.11}$$

Numerically, this yields:

$$143mm = \frac{380pxls \times 192pxls}{384pxls - 380pxls} \tag{6.12}$$

$$\therefore 143mm = 18240pxls \tag{6.13}$$

$$1mm = 127.55pxls \tag{6.14}$$

From equation 6.14, we can obtain the pixel size of the image on the image plane to be $78.4 \times 10^{-4} mm/pxl$.

Using the point $w_1 = 500mm$, $v_1 = 380pxl/127.55(mm/pxl)$ in equation 6.4, we can calculate the focal length t of the camera:

$$t = \frac{500mm \times (380/127.55)mm}{143mm - (380/127.55)mm} \tag{6.15}$$

$$\therefore t = 10.639mm \tag{6.16}$$

Therefore the focal length of the camera is 10.639mm.

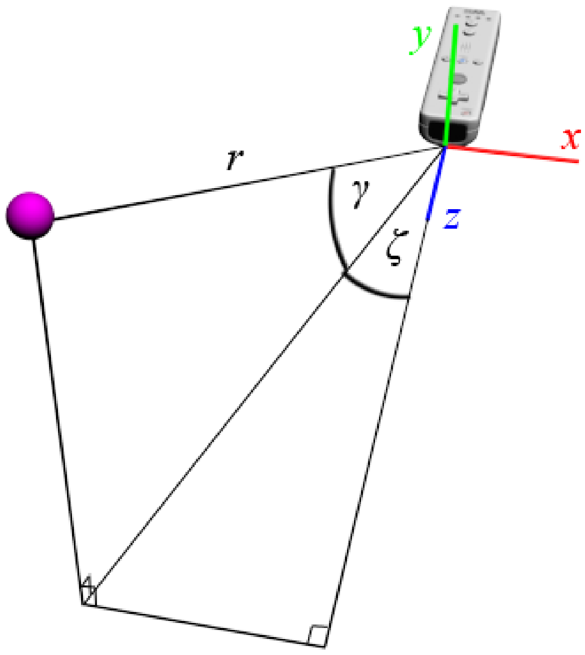


Figure 6.2: Illustration of the required extrinsic parameters

6.1.2 Calculating the Extrinsic Parameters

There are three extrinsic parameters which are required to perform accurate triangulation. Those are the distance, r, of the IR LEDs from the Wiimote, the angle of elevation, γ, of the LEDs with respect to the ground and the angle of rotation, ζ, of the LEDs from the norm of the Wiimote's camera (figure 6.2).

Calculating the distance from the Wiimote to the LEDs

In order to calculate distance accurately, 3 non-co-linear IR LEDs, with known distances between them, are required. In this case, the LEDs are placed on glasses on the user's head. One LED is placed between the user's eyes and the other to are placed next to the user's ears (figure 6.3). If we think of the IR LEDs as the source emitters, the camera lens as the image plane and the cameras internal image processor as the focal point, then we can model the system of the Wiimote camera

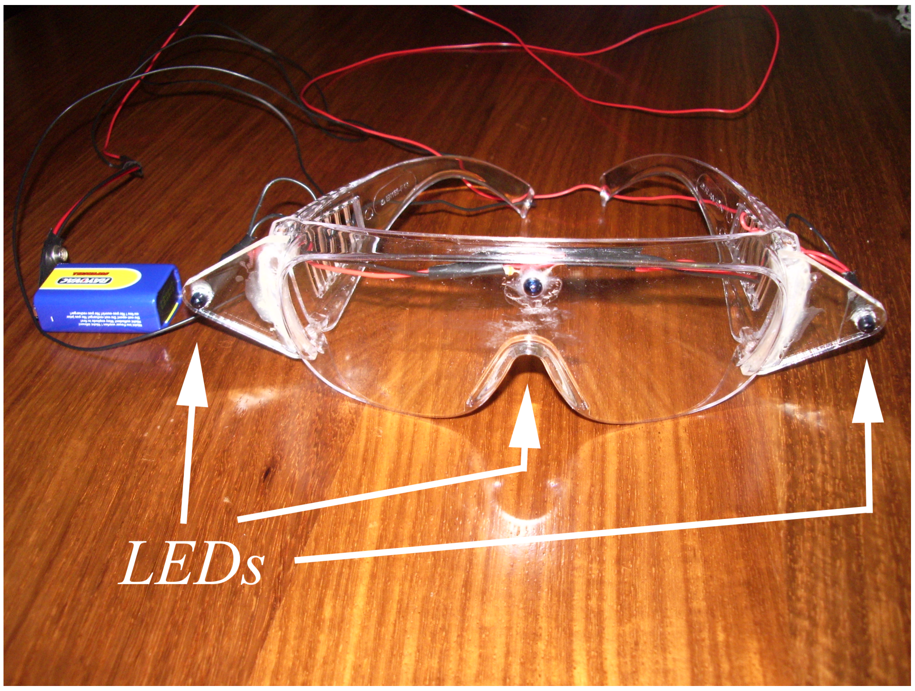


Figure 6.3: The glasses with LEDs used for head tracking

and 3 LEDs as in figure 6.4. This model shows that the each LED light, its corresponding projection on the image plane and the focal point are co-linear.

If we place this model in a x,y,z co-ordinate system with the focal point at the origin and the image plane parallel to the x-y plane, we can assume that there exists a vector for each LED which passes through the projection of the LED on the image plane and extends to the LED itself. These vectors will have a direction defined as follows:

- The x-direction value will be equal to the raw x-co-ordinate received from the IR camera minus 511 (the midpoint of the cameras co-ordinate system becomes the origin)
- Similarly, the y-direction value will be equal to the raw y-co-ordinate received from the IR camera minus 377
- The z-direction value will be the focal length (the distance between the focal point and the image plane)

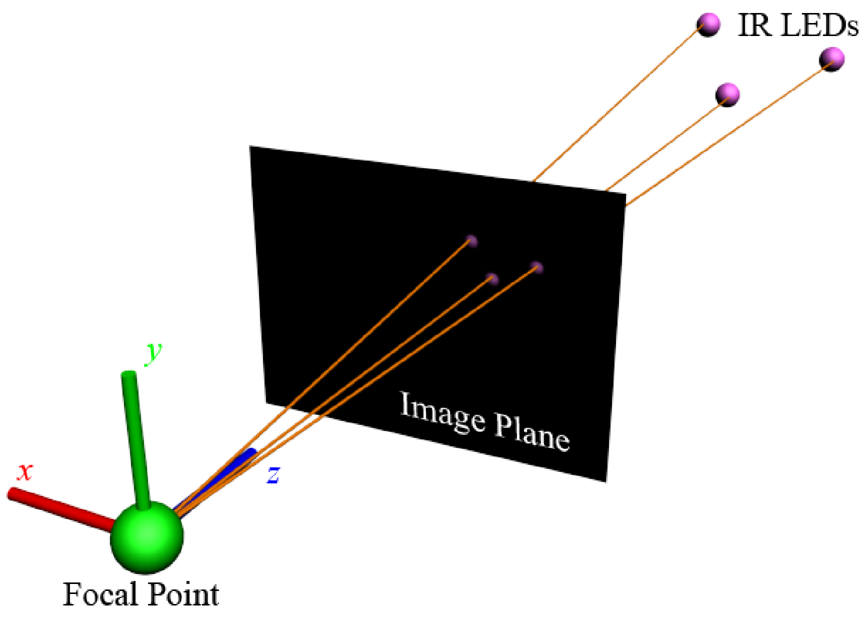


Figure 6.4: The model of the Wiimote-LED system. The orange lines are the vectors running from the focal point to the LEDs

If we do this for each LED, we will end up with a tetrahedron (4 sided shape with each side being a triangle) as shown in figure 6.5. The base of the tetrahedron is the physical positioning of the LEDs and its dimensions can therefore be measured and stay fixed no matter what angle the LEDs are viewed from.

Since each of the remaining sides of the tetrahedron are vectors pointing from the focal point to the LEDs, the angles between them can be worked out using the dot product. Assuming that 2 of the vectors from the focal point to the LEDs are $\overrightarrow{u}$ and $\overrightarrow{v}$, then the dot product of the two vectors yields:

$$\overrightarrow{u}.\overrightarrow{v} = |u| \times |v| \times \cos(\lambda) \tag{6.17}$$

Where $|u|$is the length of vector $\overrightarrow{u}$, $|v|$ is the length of vector $\overrightarrow{v}$ and λ is the angle between the two vectors.

Unfolding the tetrahedron into its 2D representation, we can separate the tetrahedron into its 4 separate sides (figure 6.6). The dimensions (A, B and C) of the base of the tetrahedron are fixed, so they do not need to be calculated, however, calculating the dimensions of the other 3 triangles (a, b

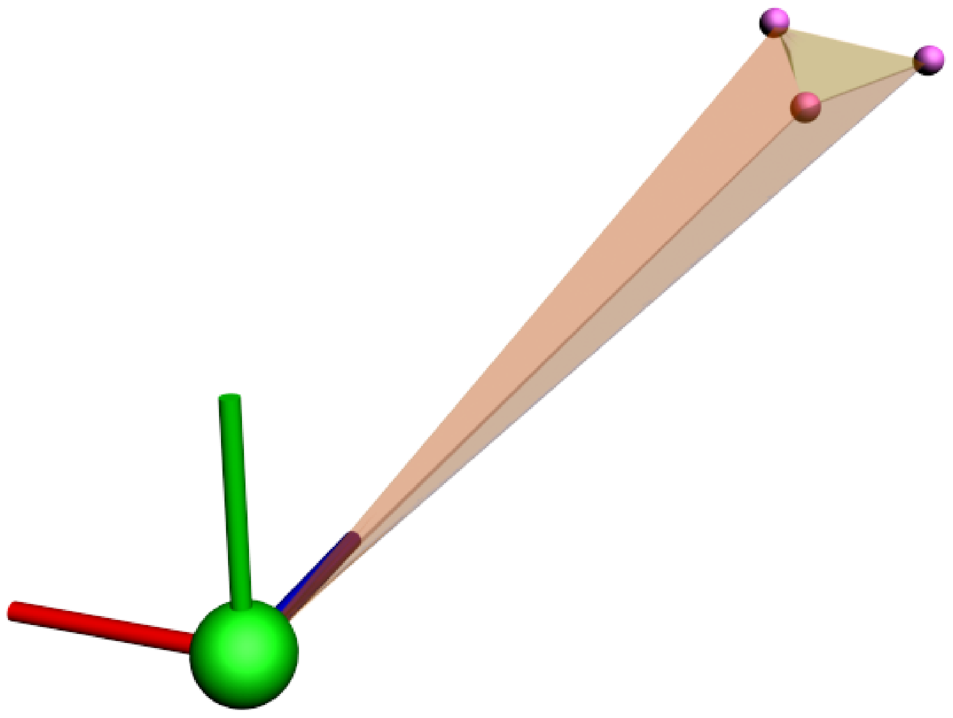


Figure 6.5: Joining each LED to the focal point will result in a tetrahedron

and c in figure 6.6), will result in the distance from the focal point to each of the LEDs. The angles (α, β and Θ in figure 6.6) between these sides are those obtained from taking the dot product of the vectors before unfolding the tetrahedron.

Using the known, fixed dimensions of the base A, B and C to find the distances a,b and c, using the cos-rule 3 times will result in 3 simultaneous equations with 3 unknowns:

$$A^2 = b^2 + c^2 - 2bc.\cos\alpha$$

$$0 = b^2 - b(2c.\cos\alpha) + c^2 - A^2$$

$$b = \frac{(2c.\cos\alpha) \pm \sqrt{(2c\cos\alpha)^2 - 4(c^2 - A^2)}}{2}$$

$$b = (c.\cos\alpha) \pm \sqrt{(c\cos\alpha)^2 - (c^2 - A^2)} \tag{6.18}$$

Similarly:

$$a = (c.\cos\beta) \pm \sqrt{(c\cos\beta)^2 - (c^2 - B^2)} \tag{6.19}$$

$$C^2 = a^2 + b^2 - 2ab.\cos\theta \tag{6.20}$$

Substituting equations 6.19 and 6.18 into equation 6.20 would yield a complex trigonometric equation. In order to solve it, I implemented recursive loop (a loop which repeats itself until, in this case, the error between the approximation and the actual value is less than a specified value) in the software which makes use of the bisection method. Solving this equation allows us to obtain the distance from the focal point to the LED positioned between the user's eyes.

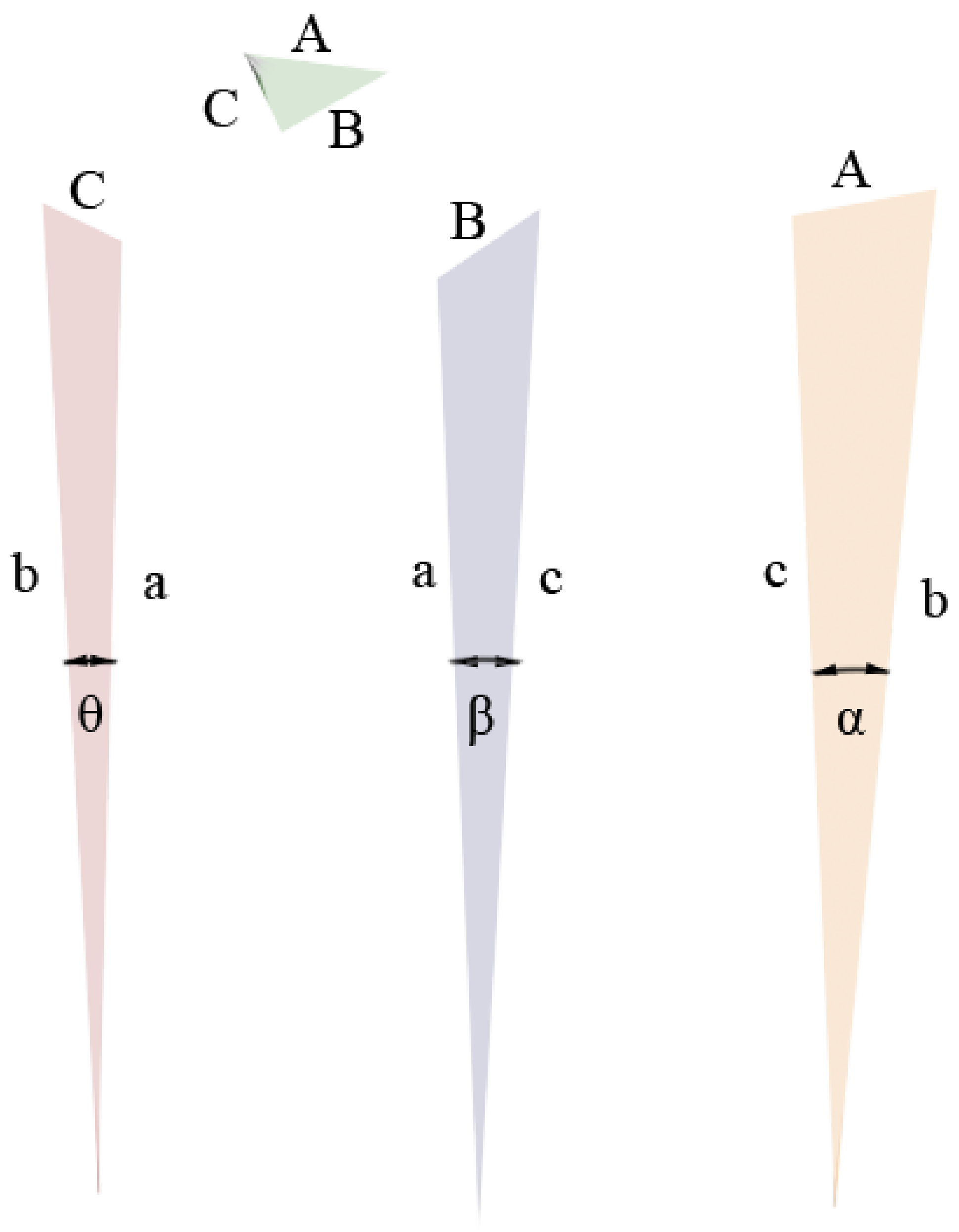


Figure 6.6: The tetrahedron formed by the focal points and LEDs unfolded

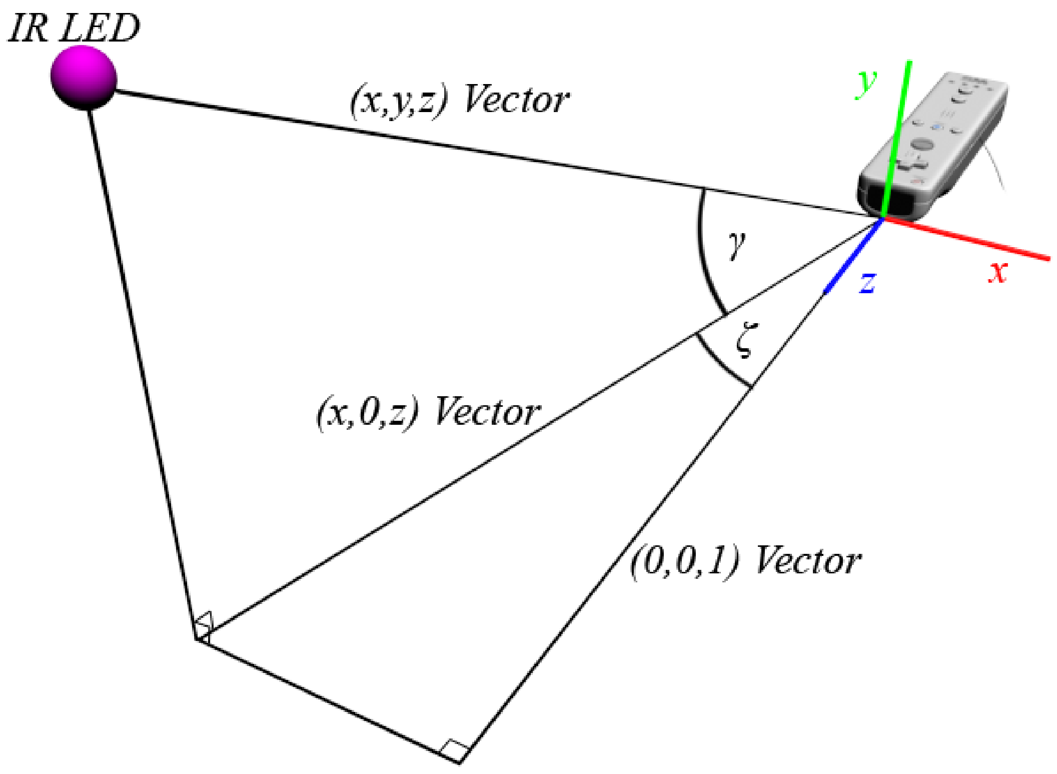


Figure 6.7: Illustration of the vectors used to calculate the angles of elevation and rotation

Calculating the angle of rotation and elevation of the LEDs with respect to the Wiimote

Modeling the focal point as the origin in the system with the image plane parallel to the x-y plane once again (as in figure 6.4), we can create a vector normal to the image plane by giving it a direction of (0,0,1). Creating a vector from the focal point to the projection of an LED on the image plane, we can obtain a vector which passes through the focal point, the projected image of the LED on the image plane and the LED itself as in the previous section (figure 6.4).

Referring to figure 6.7, assume that the vector pointing from the focal point to the LED has co-ordinates (x,y,z). By setting y to zero, we get the projection of this vector onto the x-z plane with direction (x,0,z). We can now calculate the angle of rotation, ζ, of the LED with respect to the normal of the Wiimote camera by taking the dot product of the (0,0,1) and (x,0,z) vectors (Equation 6.17).

To calculate the angle of elevation, γ, we need the (x,0,z) vector once again. This is the projection of the (x,y,z) vector onto the x-z plane. Taking the dot product of the (x,y,z) and (x,0,z) vectors will give us the angle of elevation.

6.2 Head Tracking

When looking at a 3D game on a normal monitor a user gets a limited sense of 3D depth (Objects in the distance appear smaller and moving around an object gives the user different isometric views of the object). However, if the user physically moves their head to a different angle to the screen, the image is not affected, somewhat like a painting. Using head tracking, we can locate the position and orientation of the user's head by triangulation and adjust the image on the screen accordingly. This gives an added perception of perspective to the game and the screen no longer resembles a flat painting in a frame, but rather a 3D environment viewed through a window.

6.2.1 Implementation of Head Tracking for Enhanced Perspective

When applying head tracking, we need to define how the software will respond to the movement of the user's head with regards to the virtual world. If we assume that the user is standing directly in front of the Wiimote at a standing position, then this would be equivalent to not using head tracking. If the user moves to the left, then we want the point on which they are focusing to stay the same, but their position to shift to the left.

In Chapter 4, the user's position in space was defined using two vectors. The first vector described the position of the player and the second described the looking direction. When implementing head tracking, we define a further vector and a point. Firstly, the vector points from the position of the user's head, if no head tracking was implemented, to the actual position of the user's head after head tracking manipulation. Secondly, the new point lies on the original looking direction vector, and is the focal point of the user as they move their head around (figure 6.8).

When navigating through the virtual world, the original two vectors were modified by the input of the user through the keyboard or the Wiimote. With head tracking, those same vectors are still used for the same function and are not affected by head tracking. The new point is also not affected by head tracking since it lies a pre-defined distance on the looking direction vector. Head tracking only affects the new vector. Using the triangulation methods described above, when the user moves to the side, the new vector is moved in the same direction relative to the player position vector. This gives the illusion that the player in the game is moving in the same way as the player in the real world.

In real life, when the player moves to the side, they keep their focus on the screen. To simulate this, when the player moves, the point on which they focus on in the virtual world is kept constant at the newly defined focus point.

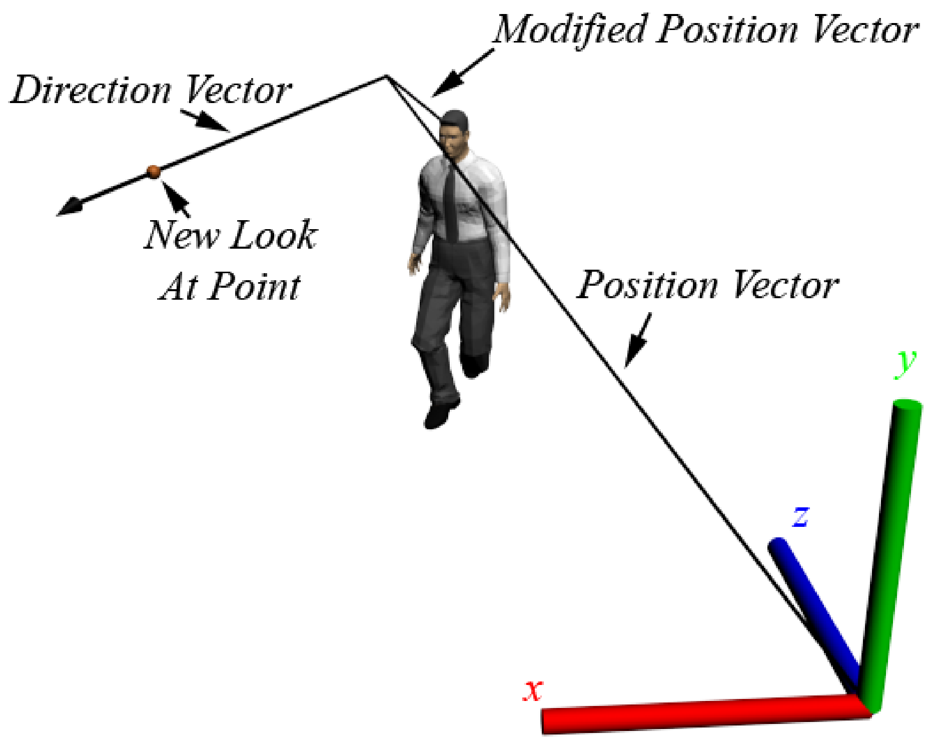


Figure 6.8: Illustration of the new vector and point used for head tracking.

Actual Distance	Calculated Triangulation Distance
100cm	103cm
150cm	155cm
200cm	207cm
250cm	256cm
300cm	311cm

Table 6.1: Comparison of the actual distance of the IR sources from the Wiimote and the distance calculated using triangulation

6.3 Results

Using the above method of triangulation, it was possible to get an adequately accurate measurement of distance of the glasses to the Wiimote. On average, the distance was out by 6.4cm over the measurements as per table 6.1. This is an adequate approximation of the distance for the head tracking application.

When the angles of elevation and rotation were calculated, it was discovered that they were very sensitive to change and had a much larger resolution than was expected. Using the calculated distance and these two angles to calculate the height of the glasses, it was discovered that a change of 1 pixel in the position of one of the sources, as seen by the camera, while the other 2 remained in one position, resulted in a change of height in over 10cm at a distance of 150cm from the Wiimote (change was higher at farther distances). This resulted in a very noisy height measurement which was not appropriate for use in head tracking.

It was attempted to improve the resolution of this measurement by averaging a range of calculations for the angles of elevation and rotation. This improved the resolution from over 10cm to just under 3cm and also improved the total accuracy of the angles of elevation and rotation, allowing for a much better measurement for head tracking.

Chapter 7

Study of the System

This chapter deals with the analysis of the Wii-integrated system and its comparison to the traditional keyboard/mouse interface. The draw-backs of the system are also discussed.

7.1 User Area Limits for Wiimote Head Tracking

Because of the allowable 45° view angle of the Wiimote camera and the 30° emitting direction of the IR LEDs, there existed some limits to the area and positions in which the user could stand.

If the user was directly facing the Wiimote's camera and in the 45° viewing area, then the range of distances that worked for successful tracking was from 55cm to 5.3m. If they stood closer than 55cm, then the camera would see double the amount of actual IR sources or none at all. Standing further than 5.3m resulted in the camera seeing one source instead of 3, so triangulation became impossible.

When the user was in the allowable area, they had to face the Wiimote almost directly because of the 30° emitting angle of the LEDs. This resulted in the image jumping around on the screen since the Wiimote sometimes captured only 1 or 2 of the sources on the glasses instead of all 3. This was a great limitation for the design of the glasses, so a second revision of the glasses used 3 LEDs per source. This increase the emitting area and improved the overall triangulation stability.

7.2 Performance

The overall performance of the game was acceptable. With collision detection and head tracking implemented the game ran at an average of 82 frames per second which was enough for smooth rendering.

The head tracking section of the software performed well. It did not noticeably slow the game down and the Wiimote camera responded with an adequate speed to update the position of the user's head for smooth head tracking. The system responded quickly to sudden movements such as dodging with no noticeable lag.

7.3 Ease of Use of Head Tracking System

The head tracking system added an extra sense of perspective which a person would have in real life when looking through a window. Since people are accustomed to such an action in real life, getting used to the head tracking interface was intuitive. Initially, users experienced difficulty with the change from the traditional keyboard and mouse input method, but adaptation to the head tracking interface was fast.

7.4 Comparison of Wii-integrated System and Keyboard/Mouse System

A number of people tested the software using both the keyboard/mouse input method and the Wii input method. Only one Wiimote was available for tests, so when the Wii input method was tested in two ways. First, the user used the head tracking input to perform jumps, ducks and dodges, while a second user performed horizontal movement using the keyboard. Second, the user used the Wiimote to move around the world without head tracking.

It was found that people who play computer games as a hobby preferred the keyboard/mouse input as they found it much easier to perform precise movements. People who did not play games often preferred the Wii input more as they found it more interactive, however, they also believed that the keyboard/mouse input was easier to achieve greater accuracy with.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

Since the first 3D first-person shooter was developed over 3 decades ago, 3D gaming has evolved drastically, however, the user-software interaction hasn't changed from the traditional keyboard and mouse combination. My research concentrated on justifying the Wiimote as a new potential input device to enhance 3D gaming. It focused on creating a 3D game based on the traditional controls and creating a head-tracking user interface and integrating it into the game.

The WiiUseJ API was used as an interface to the Wiimote controls. It had the capability of accessing the IR camera, accelerometers and buttons of the Wiimote and the accelerometer, joystick and buttons of the Nunchuk and it was compatible with the given hardware and software. The API was used to implement object manipulation, virtual navigation and head-tracking.

Object manipulation was implemented using the accelerometers on the Wiimote as tilt sensors. This added a new method of in-game object manipulation for computer games which is used in games for the Wii console.

The navigation of the virtual world was implemented using the Wiimote and Nunchuk. User tests showed that this interface had adequate performance and accuracy but was not as accurate for in-game manoeuvrability as the keyboard-mouse user interface.

Head-tracking was implemented with triangulation using the IR camera on the Wiimote and safety glasses with LEDs integrated into them as IR sources. The final head-tracking system performed adequately and used a small enough amount of processing to leave the performance of the game unaltered. User tests showed that users found the game more entertaining with head-tracking as it was more interactive.

The completed project performed effectively and was met with enthusiasm by the game testers, proving that the Wiimote is feasible as an input device to enhance the perception of in-game 3D perspective. Therefore, it is possible to conclude that the Wiimote has the potential to be integrated into future first-person games to improve the user experience.

8.2 Future Work

The following are some areas which can be further explored to test the Wii-integrated user interface and which this thesis did not have a large enough time and/or budget scope for.

8.2.1 Glasses

The IR LEDs used in the glasses had an emitting range of 30°. This resulted in jerky performance of the head tracking system. Adding an extra 2 LEDs per source improved the overall performance, but there were still a few jerky moments when the player positioned their head in a specific way.

Although they are more expensive, using LEDs with a higher emitting range could decrease this problem and improve smoothness of the game.

8.2.2 Head Orientation

The current method of triangulation uses multiple approximations of complex trigonometric functions to calculate distance. It is an effective method of triangulation requiring a little processing power, however, it does not calculate the orientation of the user's head. Using an orientation calculation method such as the Euler's method, the orientation of the user's head can be calculated and used to manipulate the orientation of the player's head in the game adding an extra enhancement in the perspective user interface.

8.2.3 Tests With Two Wiimotes

In order to test the fully Wii integrated system, two Wiimotes are required. One for head tracking and another for navigation. Future tests with two Wiimotes would result in more accurate results to the feasibility of this system as it's full functionality can be tested.

8.2.4 Enhance Interaction With Items Using the Wiimote

The accelerometer on the Wiimote was used to manipulate items in a game. This allowed the user to rotate an object which they had picked up. Using the IR camera on the Wiimote and a stationary stand with IR LEDs as a base point, triangulation with orientation can be used to overcome the problem with detecting changes in yaw rotation and also to allow the user to move the object around in front of them.

8.2.5 Other Game Types

This project is focused on testing the Wii as an input device only for a first person action game. Testing the Wiimote in games of different genres would be desirable to get a more objective view of the Wiimote as an input device for gaming.

8.2.6 Other Tracking Methods

Other methods for tracking a users head can be explored. One option would be to use facial recognition techniques to calculate the position of the user's head. In this way, the camera would not require any extra sources for tracking and, therefore, the glasses would not be required and many faces can be tracked simultaneously.

Part III

Appendix

Appendix A

Wii API Official Websites

A.1 Java APIs

Java is a platform-independent programming language. This means that all of the following APIs will work in Windows, Linux, Mac Os and Unix:

- WiiUseJ - This API is built on top of WiiUse (See 3.4.2 below). It is compatible with the accelerometer and IR sensors and the 11 buttons on the Wiimote and it has multiple connected Wiimote support. It also has support for the nunchuk's joystick, buttons and accelerometer. It does not support the speaker.
  http://code.google.com/p/wiiusej/

- Wiimote Simple - This API is compatible with the Wiimote's accelerometer and IR sensors and buttons. It has no support for the nunchuk or the Wiimote's speaker and can only support one Wiimote at a time.
  http://code.google.com/p/wiimote-simple/

- WiiremoteJ - Unlike WiiUseJ and Wiimote Simple, this API is not open source but it is well documented. It has full support for the Wiimote and the nunchuk. This is also the only API which supports connections to the Wiimote without the external Bluetooth stack application. However, this API was not compatible with the Toshiba Bluetooth stack.
  http://www.wiili.org/index.php/WiiremoteJ

A.2 C APIs

C is not a platform-independent language which means that a program written in one operating system will not function in another unless extra support is added. The following are the available C APIs for their respective platforms:

Linux:

- LibWiimote's - This API has full support for the nunchuk and Wiimote (including the speaker).
  http://libwiimote.sourceforge.net/

Windows:

- Wiim - This API supports the Wiimote's buttons and accelerometer sensor, however, it has no support for the IR sensor, speaker or the nunckuk. This API project is no longer being maintained.
  http://digitalretrograde.com/projects/wiim/
- Wiimote-Lib - This is the original API used by Johnny Lee in his head tracking demonstration. It has full support for the nunchuk and other extensions and can use all of the Wiimotes features except for the speaker.
  http://blogs.msdn.com/coding4fun/archive/2007/03/14/1879033.aspx
- Wiimote-api - Supports all the Wiimote functions except the speaker. Does not support the nunchuk.
  http://code.google.com/p/wiimote-api/
- WiiYourself! - Supports all nunchuk and Wiimote functionality (Speaker has experimental support).
  http://wiiyourself.gl.tter.org/

Multi-Platform:

- WiiUse - This API has full support for the Wiimote, except the speaker. It also fully supports the nunchuk and various other extensions. Multiple Wiimote connections are also possible.
  http://wiiuse.net/?nav=docs

Appendix B

Read Me File on CD

##A project report submitted to the Department of Electrical Engineering, ##University of Cape Town, in partial fulfilment of the requirements for ##the degree of Bachelor of Science in Engineering.

##Author: Victor Kirov ##Supervisor: Prof. M. Inggs

Implementation of a 3D Game Using the Wii-Remote for Enhanced Perspective Simulation

This readme file describes the information that can be found on the CD accompanying this project. The following is the directory structure of the CD and the contents of each directory:

/Root/Thesis.PDF //This is a digital copy of the project write up

/Root/Movies/HeadTracking.mpg //This is a video of the head tracking in action

/Root/Movies/Traditional.mpg //This is a video of the game without head tracking

/Root/Movies/ItemManipulation.mpg //this is a demonstration of item manipulation using the Wiimote

/Root/Software/ /Java SDK/ //the Java software development kit used to compile the software

/Root/Software/Netbeans/ //The Java compiler used in creating the software

/Root/Software/Adobe Acrobat/ //A PDF reader to open the digital copy of the thesis document

/Root/Wiigame/ //This is the game used to test the head-tracking in the project saved as a NetBeans project

/Root/Wiigame src/ //This is the game used to test the head-tracking in the project saved in a generic method which can be opened by any java compiler

/Root/Wiigame src/WiiGame/ WiiGame.java //The source file containing the main method to run the game

/Root/Wiigame src/ObjLoader/ //The source files for the classes which load the 3D models used in the game

/Root/Wiigame src/images/ //The textures used in the game

/Root/Wiigame src/models/ //The 3D models (.obj files), their material maps(.mtl files) and their textures

Bibliography

[1] Andrew Davison. *Pro Java 6 3D Game Development*. Apress, 2007. ISBN 1590598172.

[2] Oliver Kreylos. Wiimote hacking. page Technology, 19 October 2008
URL `http://idav.ucdavis.edu/~okreylos/ResDev/Wiimote/Technology.html`.

[3] Jack B. Kuipers. *Quaternions and Rotation Sequences*. Princeton University Press, 41 William Street, Princeton, New Jersey, 1999. ISBN 0-691-05872-5.

[4] Johnny Chung Lee. Head tracking for desktop vr displays using the wiimote. 19 October 2008
URL `http://www.cs.cmu.edu/~johnny/projects/wii/`.