Headtracking using a Wiimote

Kevin Hejn
khejn@diku.dk

Jens Peter Rosenkvist
jenspr@diku.dk

28th of March, 2008

Graduate project, 7.5 ECTS
Supervisor: Kenny Erleben

Department of Computer Science
University of Copenhagen

Contents

Abstract

This paper presents our adaption of an approach to achieving a virtual reality like experience on a typical monitor using the Nintendo® [19] Wiimote. By using the infrared camera on the Wiimote combined with infrared light emitting diodes positioned on the user's head, we achieve the effect that the content visible on the monitor adapts to the position of the head relative to the monitor. Thereby we achieve the same effect as when looking through a window; what is visible depends on the angle and distance from the window.

1 Introduction

In line with the growth of the game industry, the demands for realism in modern computer games increase as well. Realism is often raised through improved graphics, artificial intelligence, sound effects and similar. The player interaction devices are however, mostly limited to the use of mouse, keyboard and conventional game controllers. The possibilities have improved somewhat by the introduction of Wii[1], Nintendo's® newest game console which was introduced in 2006. Now players can interact more directly and naturally with the games, thus achieving higher realism. This is done through the Wiimote controller which, among other things, features motion-sensing. However, critical areas are still left behind. There is for example poor access to equipment allowing the player to change the view perspective by moving her head. This can indeed already be achieved using available virtual reality (VR) equipment, but for most users it is not within an acceptable price range.

It turns out that equipment allowing this kind of interaction does not necessarily have to be expensive. It can be achieved using an infrared (IR) camera and some light emitting diodes (LEDs). By placing the camera by the monitor and the LEDs by the head, the placement of the LEDs can be found, hereby allowing us to determine the user's movement and changing the field of view according to this. This yields the illusion that the monitor is like a physical window into another room instead of just a static photo. This can increase the realism for especially 3D games dramatically.

Johnny Lee [14] from Carnegie Mellon University has realized the task using the IR camera on the Wiimote to facilitate the job using a simple setup illustrated in figure 1. Normally registration of images is a difficult job and forms

[1] Wii is a registered trademark of Nintendo®

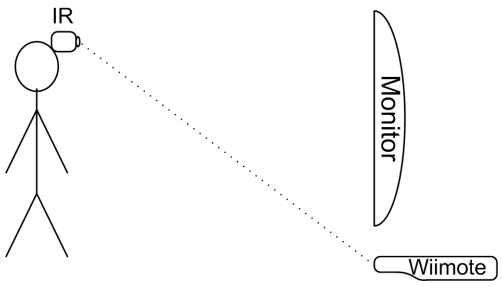

Figure 1: The setup used to perform headtracking with the Wiimote and IR LEDs.

an entire research field, but when taking advantage of the build in IR camera and image processor of the Wiimote, it becomes quite simple to detect and track numerous positions. Furthermore the Bluetooth support and low price of the controller makes it easily accessible. Based on these arguments, we feel that the idea presented by Johnny Lee calls for further investigation. In this paper we will therefore try to mimic his work and determine the field of view in a 3D game like world from the available data. Through a user study we furthermore wish to evaluate whether our solution is suited for interaction in a 3D world.

To keep the focus of the project on the subject at hand, we will assume that the reader is familiar with the process of setting up a camera in a 3D scene. Thus terms like the up vector and projection matrix will not be explained in the paper. A thorough introduction to this subject can be found in [4].

Included with the paper is a CD-ROM containing a digital copy of the paper, the source code, application program interface (API) documentation and a video illustrating our solution in action.

2 Wiimote

The Wiimote is a controller for the Wii. In contrast to most other controllers for game consoles, the input methods are not just buttons and analogue sticks, but an IR camera and motion-sensing. In this section we will describe these features. The technical specification is based on [3, 29, 30].
The appearance of the Wiimote is designed much like a TV remote as seen in Figure 2. It has 12 buttons, where 11 are on the top and one is located

Figure 2: The Wiimote seen from the side, front, top and back.

underneath. The Wiimote has 16 kilobyte EEPROM[2] of which a part is freely accessible and another is reserved. The Wiimote also houses a speaker and the top has holes which the sound can come out of. Four blue LEDs are on the top. If the Wiimote connects to a Wii the LEDs first indicate the battery level and afterwards which number the Wiimote is connected as. A small rotating motor is placed in the Wiimote which can make the Wiimote vibrate. A plug is located in the end of the Wiimote. Through this plug attachments can be connected. A couple of peripherals have been released. However, the most used is the *Nunchuck*. The Nunchuck is a device with two buttons, an analogue stick and motion-sensing as the Wiimote itself.
The Wiimote can detect motion in all 3 dimensions. This is achieved through accelerometers inside. For a thorough discussion of the functionality, see [20].

The front of the Wiimote has a small black area like a normal TV remote. The difference is that a TV remote has an IR LED beneath, where the Wiimote has an IR camera. By placing a so called *sensor bar* (essentially IR LEDs powered by the Wii) below the TV, the Wiimote can be used as a pointing device for the Wii. As can be seen on Figure 3 the LEDs in the sensor bar are located with space between them. This makes it possible to calculate the relative position of the Wiimote with regards to the sensor bar. When using the Wiimote as a pointing device for the Wii, one doesn't actually point at things on the TV, but is pointing relative to the TV.

[2]Electrically Erasable Programmable Read-Only Memory

Figure 3: The sensor bar for the Wiimote seen from the front.

To minimize the amount of data being transferred from the Wiimote to the Wii, it does not actually transfer every image taken by the Wiimote. Instead the Wiimote calculates the position of each point and sends the x- and y-coordinate for each of them together. It can also send other information such as a rough size of the points. It is noted that the LEDs that can be seen of Figure 3 are taken as two separate LEDs and not 10. This is because they are located so close to each other. The Wiimote can register up to four separate points at a time.
The Wiimote does not just take an image, analyze it and send the result of the positions to the Wii, it also keeps track of the points. Every time a point is detected it is assigned a number from one to four. If for instance two points are detected and the one marked as 1 is moved out of range, the other point is still marked as 2. If the point is then moved within range again it is marked as 1, but the Wiimote of course can not tell if it is the same LED as before or a new one.

3 Previous Work

Since the release of the Wiimote several papers have been published, describing how to interact with graphical applications like games in a new way using the Wiimote. In this section we will comment on a selection of the most relevant papers in the field.

The paper [27] discusses how gestures in 3D can be performed with the Wiimote and used to interact with a 3D environment. Through user studies the paper examines which gestures are appropriate to reflect common interactions like waving, with the 3D virtual world, Second Life [13]. The evaluation of the gestures depends solely on data from the accelerometer, thus the IR camera and buttons on the Wiimote are not used. The paper focuses on a human-computer interaction (HCI) aspect of the problem and does not discuss the technical aspect of how the data are interpreted or if they encountered any problems using the Wiimote.

In [23] it is described how the Wiimote is used to interact in a 3D virtual reality theatre. They have adapted the Half-Life 2 [2] game engine to run in the two-walled theatre and to use the Wiimote to look around and aim. Both buttons, accelerometer and IR camera is used to interpret the position of the user and user actions. Since the IR part is essential, the IR camera is described in great detail and several measurements have been performed to determine the range in which the IR works.
Due to the limited IR angles, several sensor bars are used to make sure that at least one IR LED is almost always visible. Because the Wiimote was designed to cope with one to four IR LEDs, they have made several extensions to the default interpretation of the data. Furthermore the problem of lack of visible LEDs for short periods is briefly discussed.

In the paper [25], it is presented how the Wiimote can be used for interaction in three existing applications; two games and a drawing application for children. For the two games, only data from the accelerometer is used. Especially in one of the games, this poses a problem, since it is difficult to estimate the exact position of the controller. This problem is also described in [20].
For the drawing application both accelerometer and IR camera is used to facilitate the painting. Here especially the problem of determining the physical distance between the Wiimote and the sensor bar is described. Since the main focus of the paper is the HCI aspect, there are few details regarding the actual data analysis and implementation.

In [20] they tried to estimate the movement and orientation of the Wiimote based on the date from the accelerometer, while the IR functionality is omitted. The intention was to be able to navigate objects in a 3D environment. In contrast to [27, 23, 25] the approach is significantly more technical. The communication with the Wiimote as a human interface device (HID) on a computer, the theory behind the accelerometer and algorithms for data analysis are described in great detail. Also detailed information about encountered problems related to noise is presented.

In general the previous work within the field has mainly been focusing on the accelerometer in the Wiimote ([27, 23, 25, 20, 17]) while only few have used the IR camera and sensor bar ([23, 25]). In general the buttons have been ignored since these are the same as on any generic game controller. In all the papers mentioned, the user interacts directly with the Wiimote to perform the actions, thus yielding a replacement for mouse, keyboard or game controller. In contrast we want to use the Wiimote as a passive device and

instead let the user move the sensor bar. We have not been able to find any papers describing this type of use. However, Johnny Lee has demonstrated this use in several videos and on his blog [14]. Unfortunately the only documentation on his projects is the source code.

4 Analysis

In this section we will first give a general analysis of the problem we are trying to solve. This is followed by a concrete analysis of the solution of Johnny Lee and the methods used. Finally this will result in a description of the solution we will use to achieve our goal.

4.1 Defining the desired effect

As described briefly in the introduction, our goal is to use headtracking to achieve a VR like effect when looking at a monitor. The effect we wish to obtain can be clarified with the following analogy which is also used in [15]. A traditional monitor can be compared to a photo in a frame. No matter at what distance or angle you watch the motive, you see exactly the same content. This is due to the 2D nature of the photo or image displayed, even though the original content was three dimensional. However, if you remove the photo from the frame, this changes. Now, what you see through the frame depends on the angle and distance that the frame is viewed, exactly like when looking through a window. This effect is what we wish to achieve on the monitor. Depending on the position of the user, the visible content changes to reflect her movement.

4.1.1 Movement scenarios

The movement that the user can make can be separated into two general categories; changes in the angle between the front of the monitor and the user and changes in the distance between the monitor and the user. The change of angle is an effect of the user moving around in a constant distance from the centre of the screen (i.e. the position of the Wiimote). When this occurs, some elements should vanish while others should become visible. Again, using a window as an example; if you stand in front of a window and move to the left, you can see more to the right on the other side of the

window and vice versa. The situation is illustrated from above in Figure 4(a).

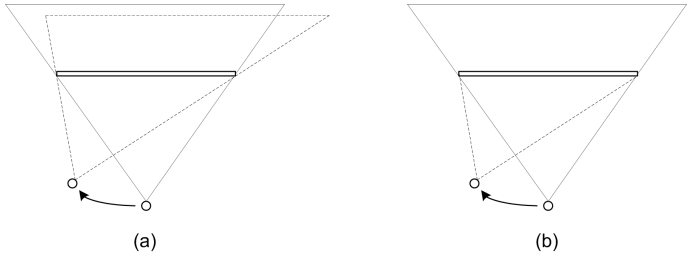

Figure 4: The change in the visible field when the user moves to a different angle relative to (a) a window and (b) a monitor.

As a contrast, the normal scenario when using a monitor is shown in Figure 4(b) where it is obvious that the angle of view has no influence and thus seems unrealistic. The same principles also apply when lowering or raising your head; you can see more of the sky through a window when crouching than when standing.
The other category, change of distance, is a different effect as the distance between the monitor and the user determines how much of an image should be visible overall. Returning to the window example; the closer one is to a window, the more one can see outside the window in all directions. The situation is illustrated in Figure 5(a).

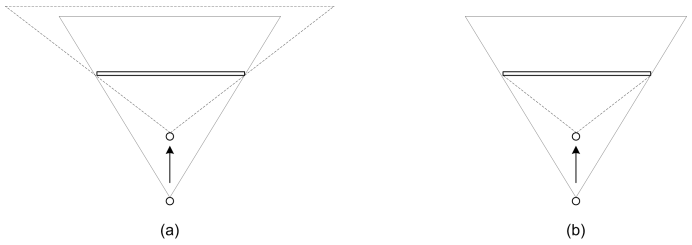

Figure 5: The change in the visible field when the user moves closer to (a) a window and (b) a monitor.

As one moves closer to the window, more of the outside becomes visible. As before, Figure 5(b) is provided as a contrast, showing how distance has no effect when using a traditional monitor.

To enable the kind of interaction described above, the computer needs to know the position of the user's head relative to the monitor. This problem is known as headtracking[3]. As mentioned, headtracking in general is difficult since the process of finding areas of interest and registering several images is complicated. However due to the IR camera and onboard image processor of the Wiimote, the task is significantly simplified. As described in section 5.2.2 the data returned from the Wiimote contains absolute coordinates of up to four points and optionally size estimates. Since little information can be obtained when only one point is used, as the size estimate is too coarse, two points are required. This is also the amount of points recognized when using the sensor bar.
This approach is the foundation in the solution of Johnny Lee as well as in our solution. In the following sections we will discuss these.

4.2 The solution of Johnny Lee

In this section we will give an analysis of how Johnny Lee has constructed his solution and how he has solved certain problems. His solution is very comprehensive which makes it impossible to mention all details, but we will focus on the key elements. As an example, his solution is able to handle a Nunchuck, which is irrelevant to this project. Since our analysis is based solely on his source code [16] and a few comments with it, we can not comment on the reasoning behind his choices, but only the final solution. Lee's implementation is made in C# and can be split in three parts; communication with the Wiimote, interpreting the data and visualization.

All interaction with the Wiimote is done through WiimoteLib [21], which is not developed by Lee. This library contains methods for every kind of interaction with the Wiimote which is publicly known at the moment. The implementation is made quite similar to our approach described later, in section 5.2. The parts we did not mention there are not needed and Lee does not use them in his program. He basically uses the library to retrieve the position of the LEDs and then use these positions.

4.2.1 Interpretation of data

When the position of the points is known, the user's position can be calculated. The distance between the two LEDs is inverse proportional to the

[3]The term headtracking also covers the area of recognizing facial expressions.

distance between the user and the screen. However, instead of just using the distance directly, Lee takes the size of the monitor into consideration. This is done by allowing the user to specify the size of her monitor. If the information is not given, a default value will be used.

As can be seen in Figure 6 the distance from the user to the screen is inverse proportional to the tangent to half the viewing angle measured between the LEDs. To calculate this distance, the distance between the two LEDs is scaled with the size of the monitor and used to derive the relative distance between the user and the monitor. This distance is then used to calculate the position of the user relative to the monitor: By using the sine-function on the angle which indicates the distance between the LEDs on the x-axis, the x-coordinate relative to the camera-space is known. This method assumes, that the user is centred on the x-axis, when directly in front of the Wiimote. The y-coordinate in camera-space is calculated much the same way, but this coordinate is not assumed to be centred directly in front of the Wiimote. To take this into consideration an offset is used. This offset can also be changed by the user to fit the setup used.
In the above calculations we assume the position is centred around the origin. Since the Wiimote returns values ranging from 0 to 1023 and 0 to 767 for the x and y respectively, we must deduct 512 from the x and 384 from y.

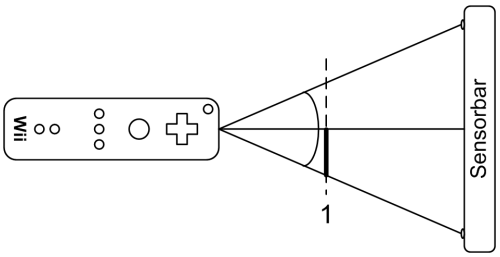

Figure 6: Interpreting the LEDs' position.

Combined, the x- and y-coordinate and distance from user to screen can give the camera position, camera target and camera up vector. From these three vectors the three axis of the coordinate system to be used can be determined:

$$\text{zaxis} = \frac{\text{position} - \text{target}}{||\text{position} - \text{target}||} \tag{1}$$

$$\text{xaxis} = \frac{\text{up vector} \times \text{zaxis}}{||\text{up vector} \times \text{zaxis}||} \tag{2}$$

$$\text{yaxis} = \text{zaxis} \times \text{xaxis} \tag{3}$$

The view-matrix can now be determined from the axis by the following matrix:

$$\begin{bmatrix} & & & 0 \\ \text{xaxis} & \text{yaxis} & \text{zaxis} & 0 \\ & & & 0 \\ -\text{xaxis} \cdot \text{position} & -\text{yaxis} \cdot \text{position} & -\text{zaxis} \cdot \text{position} & 1 \end{bmatrix} \tag{4}$$

The boundaries of the screen are also needed. The x- and y-coordinate gives the offset of the boundaries, while the distance gives the scale. If the offset was not used, objects closest to the user would move around, which must not happen. By using the offset it is ensured, that when the camera moves from side to side and up and down the front plane in the view-frustum remains the same. To keep the view-frustum the same when the camera is moved back and forth the boundaries must also be scaled with the front clipping plane. The boundaries are used to calculate the projection-matrix, which is constructed as follows:

$$\begin{bmatrix} 2 \cdot \frac{\text{front plane}}{\text{right}-\text{left}} & 0 & 0 & 0 \\ 0 & 2 \cdot \frac{\text{front plane}}{\text{top}-\text{bottom}} & 0 & 0 \\ \frac{\text{left}+\text{right}}{\text{right}-\text{left}} & \frac{\text{top}+\text{bottom}}{\text{top}-\text{bottom}} & \frac{\text{back plane}}{\text{front plane}-\text{ back plane}} & -1 \\ 0 & 0 & \frac{\text{front plane} \cdot \text{back plane}}{\text{front plane}-\text{ back plane}} & 0 \end{bmatrix} \tag{5}$$

It is noted, that the frustum given by this matrix is not necessarily the same to the left and right of the camera. However, this is needed because the plane closest to the user has to be locked to the monitor to achieve the effect of looking through a window.
Calculating these two matrices finishes the interpretation of the data, since the world matrix needs not be modified.

The visualization is not particularly interesting. After the matrices needed for the transformation are calculated, the graphics are made as in most other 3D applications. This means that objects are shown by transforming them from the 3D space into 2D space by using the mentioned matrices. After this, they are rendered to the monitor.

4.2.2 Robustness

An aspect not taken into consideration in the solution is robustness. A key element in the approach to headtracking is that the Wiimote is able to track the LEDs at all times. However this might not always be the case. One problem is that the sensor bar can be moved outside the field of view of the camera. This happens if the user moves too far away or steps outside the angles of the camera. The other situation is if the line of sight is obstructed, either by the user, another person or by some object.
In the solution of Lee it is detected and displayed whether both LEDs are visible. If not, the perspective is simply not changed and thus, the content on the screen will simply freeze. When both LEDs then become visible again, a new set of matrices are calculated, solely based on the new LED positions. While this indeed works, it can cause the perspective and hence the content of the monitor to change dramatically if the user has moved a lot since both LEDs were last visible. If there are lot of obstructions or the LEDs are not sufficiently bright, this will cause an almost flickering effect due to the sudden, non-smooth changes in the perspective. Possible ways to handle the missing LEDs will be discussed in section 4.3.3.

4.2.3 High frequency noise

In the solution of Lee the movement detected by the Wiimote is used directly in the calculation of the perspective. However, when a person moves from one point to another, it is not only the general motion that is detected, but also a slight jittering due to small movement back and forth. For example this can happen if the user is unable to hold the head still and constantly turns and nods a bit. When movement containing this high frequency noise is mapped directly to the virtual world it can potentially result in perspective changes when standing still.
One could argue that constant small variations in head position are normal and thus it is only realistic to actually map these motions to the monitor. However, when displayed on a monitor, it does not seem realistic. Instead

it will seem like the screen is shaking which makes it an undesired effect. Of cause the magnitude of the effect depends on several things like the user and location. Since the solution of Lee is intended to be used at home, the location has little impact, but imagine that it should be used with a portable device and thus e.g. in a car. Even though this is not the case and for many users it might not be a noticeable problem, we still think the subject should be considered. Therefore we have included a brief discussion of ways it can be handled in section 4.3.4.

4.3 Our solution

We got the initial idea for the project from the work of Johnny Lee. Therefore our solution bears much resemblance to his. Especially how we exactly calculate the matrices used to set up the camera are heavily inspired by his work. Because of the similarities, we will focus on the aspects not mentioned in the previous section.

4.3.1 Position estimation

Since the two IR LEDs are placed centred on the users head, the midpoint between the two detected points can be considered an approximation to the user's head, which is where the camera should be placed. By looking at the x- and y-coordinate of the midpoint, it is possible to determine the angle between the centre of the monitor and the user. However by looking solely at the reported x- and y-coordinate, no information regarding the distance between the monitor and the user is available. Instead the distance can be determined by calculating the distance between the two points. As the distance between the points becomes smaller when the two LEDs are further away from the camera, the distance between monitor and user can be approximated as described in section 4.2.1. Combined these data make it possible to determine the position of the user. However, it should be noted that if the LEDs are not parallel with the monitor, they will appear closer to each other, thus the calculated distance is too large. As long as the user faces the monitor, the error is minimal and therefore there is no need to counter this problem.

4.3.2 Setting up the camera

To set up a camera, three vectors must be known. These are the camera position, camera target and camera up vector. If the distance from the user to the screen is called *distance*, the camera position is the vector $(x, y, distance)$, the camera target is $(x, y, 0)$ and the camera up vector is $(0, 1, 0)$. It should be noted that with this up vector, the screen will remain still if the user tilts his head, but going back to the analogy with the window one can see this is in fact correct.
Besides these three vectors, the camera also needs to know the boundaries of the virtual world to be displayed. Normally when rendering a 3D world the boundaries do not change, but this is needed here because we want to lock the monitor to a certain position in the virtual world. These boundaries can be determined from the x- and y-coordinate and the distance. The x and y determines how much the window must be translated along the two corresponding axis and the distance is the scaling factor of the boundaries.
When all these variables are determined the transformation matrices can be constructed as described in section 4.2. Afterwards the rendering of the scene can be performed. Since this is done quite simple with OpenGL [24], we will not cover this further.

4.3.3 Robustness

As described in section 4.2.2, we cannot assume that both LEDs are visible at all times, might it be due to the user moving to far away from the screen or an object getting between the Wiimote and the LEDs. The first problem encountered is that data is missing at some point and the second problem is what to do when both LEDs become visible again. We will discuss these two problems and possible solutions here.

When the Wiimote is unable to detect a LED it will send a default value. Therefore it is easy to detect when the LEDs are out of sight. The simplest solution is just to use this value as if the data is correct. However, this will of course give a poor result because it will make the screen move a large distance instantly.
Another solution is to pause the game until the LEDs are detected again. This solution will notify the user immediately and make her solve the problem (e.g. move within sight or remove the obstructing object), but will also ruin the flow of the game.
We have chosen a third option where we keep the last correct detected value.

This will stop the screen from making sudden movements and will not ruin the flow of the game. A problem here can be that the user perhaps does not notice that the LEDs are out of sight. To avoid this, some sort of visual indication can be given, for instance a small blinking icon in the corner of the screen.

The second problem is as mentioned what to do when the LEDs become detectable again. No matter which one of the above solutions is chosen it will still be a problem. If someone briefly moves between the Wiimote and the user, the x- and y-coordinate will be fairly similar before and after. However, if the user has moved significantly, the detected coordinates will change accordingly, yielding a large, sudden movement on the screen.
Instead of just using the new value as suggested above, the old value can also be taken into account. For instance the new value can be weighted with 0.7 and the old value with 0.3. Effectively this will result in an interpolation between the two positions, which will make the transition from the old to the new position smooth. Even though the screen for a moment will behave differently than the user, the advantage that no jumping appears outweighs this problem. However, it has to be decided how long this transition must take. The longer it is, the more smooth, but if it is too long, the responsiveness will seem poor. Somewhere around a second will therefore be fitting. Instead of just using some of the old value when the LEDs have been undetected, it can be done always. The advantage is that this is a simple way to reduce the high frequency noise, which we will discuss further in section 4.3.4. The disadvantage is that the overall responsiveness is lowered. This approach is also taken in the driver library cWiiMote [5].

In the above described scenarios it is only considered what happens if both of the LEDs are out of sight. However, it could happen that one LED was detected and the other undetected. This will probably happen less frequently because the two LEDs are relatively close to each other, but it is a problem nonetheless. One solution could be to keep the old value for the undetected and still update the detected one. This is a poor solution since the result quickly becomes completely false. For instance a movement in the direction of the undetected LED would result in a flipping of the two LEDs.
Another solution is to let the two LEDs move together. If the detected LED moves in one direction the old value of the undetected LED is moved the same way. This will give a better result than the first solution, but still not be perfect since for instance moving closer to or farther from the screen will go unnoticed. In both this and the previously mentioned solution the result could be that the screen moves differently from the user.

The simplest solution is to treat it as both LEDs are undetected. The advantage of this solution is that the user becomes aware of the problem and can solve it. Another advantage is that the screen will not move differently from the user. We find this solution the best and have therefore chosen it.

4.3.4 Removing high frequency noise

In section 4.2.3 we described how high frequency noise could occur due to small, unintentional variation in the position of the user. To avoid this problem some sort of data analysis is needed. The simplest solution is to discard all movement below a certain threshold. This solves the problem of the screen shaking when the user is still, but when the user is moving the problem would still occur.

Another solution is to predict where the user will move. This can be done by using a small set of the previously values and then calculate the average velocity and speed of the user. This will reduce the problem much, but it has an undesired side-effect. For instance when the user moves, but makes a sharp turn the prediction is wrong. However, as long as the user moves relatively smoothly the error in the prediction will be small and unnoticeable.

A third option is to perform a filtering of the data. By considering the data as a signal where the relevant data is made up from low frequencies and the noise is made from high frequencies, we can perform a low-pass filtering of the signal to reduce the noise.
Low-pass filtering works by letting the low frequencies pass and reduce the amplitude of the frequencies above some cut-off frequency by smoothing the signal. It is done by performing a convolution of the signal with some filter. The simplest low-pass filter is the box filter which simply cuts off frequencies above the specified threshold. However this can lead to undesired effects because of the discontinuity of the filter. A better option is to use a smooth filter. One commonly used is the Gaussian filter. It is defined by the function

$$G(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}} \tag{6}$$

where σ is the standard deviation of the distribution. Due to the bell like shape of the function, one achieves a much smoother filtering than with a box filter. There exist several other smooth filters that can be used for low-pass

filtering, but we will not discuss them here.

When choosing the cut-off frequency we want a value such that all small variations are removed while actual motion is kept intact. This calls for a trade-off. If the cut-off frequency is low, most or all of the noise is removed and the perspective will remain very calm. However we run the risk of removing frequencies which are actually user movement. This will result in less responsive behaviour. On the other hand, if a high cut-off frequency is used, we might end up allowing much of the noise to pass, yielding the filtering useless. Therefore one must be careful when choosing the cut-off frequency. Another drawback with the method is that it is necessary to delay the signal a little. If the Gaussian kernel used has a size of 7, the reported signal will be three samples behind. This is because the three previous and three following samples must be considered when filtering any given sample. If the filter becomes big, it can therefore result in a noticeable delay from the sensor bar is moved, to the content on the monitor adjusts. To avoid further delays, it must also be guaranteed that the filtering can be performed in real time. If not, increasingly large delays will occur over time.

To determine the optimal method for predicting the user's movement, it would require a large number of experiments. Since this is too time consuming for this project, we have chosen not to use any prediction or filtering. We do however, interpolate between the previous position and current position of the user as mentioned in section 4.3.3. We have done this because it is an extremely easy way of reducing the high frequency noise, but the result is far from as good as the other mentioned solutions.

5 Communication

In this section we will describe how the communication between the Wiimote and the computer or Wii is performed. The general mean of communication is discussed first, followed by specific details for the Wiimote, which are based on [29, 30].

5.1 Bluetooth

Bluetooth [26] is a specification for wireless communication between devices over short distances. A large part of the specification is optional, which al-

lows a minimum set of features to be implemented in a device and exclude those unnecessary.
A device connected with other devices can function as a *master*, a *slave* or both if needed. A master device controls the communication and up to seven slaves can connect to a master. This network of up to eight devices is called a *piconet*. If it is necessary to connect more than eight devices at a time, a slave can switch to being both a slave and a master, hereby creating a new piconet and connecting up to seven more devices.

If a Bluetooth device is queried it will transmit its device name, device class, a list of services and other technical information. When a slave wants to connect to a master it must be *discoverable*. When the master sees a slave it can establish a connection. If both units support it, it is possible to pair two devices with each other. To support this feature the two devices must know a shared passkey. When two devices are paired the connection between them can be reestablished automatically. If they do not have a shared passkey the connection must be set up manually each time.
By pressing a special `sync` button under the battery cover on the Wiimote, it becomes discoverable for 20 seconds. Within this period the user must manually pair the Wiimote with the computer. Alternatively one can hold button 1 and 2 to make the Wiimote discoverable.

When a connection is established the data from the device can be read. To read the data, a Bluetooth stack is needed. However, it is not all implementations that support all devices. The Bluetooth stack in Windows XP does not support the Bluetooth dongle we have used in this project and another one is therefore needed. The Bluetooth stack BlueSoleil [11] supports the dongle and we have therefore used it.
Each device manufactured has a deviceID and vendorID. The deviceID identifies which device it is and the vendorID identifies the manufacturer. For the Wiimote the deviceID is `0x0306` and the vendorID is `0x057e`. This means that all Wiimotes have this deviceID and vendorID, but no other devices have the same combination. This of course makes it possible to read the data from the correct device and not another random device found nearby.

When the connection between the Wiimote and the computer is established through BlueSoleil it is possible to connect to the Wiimote as a HID device. After this connection is established the communication to and from the Wiimote is done by reading and writing to different ports.

5.2 Communicating with the Wiimote

As mentioned in the previous section, communication between the controller and the computer is achieved using the HID standard. This enables one to get an enumeration of the *reports* that the controller understands using a HID descriptor block. *Reports* are conceptually equivalent to network ports assigned to specific services since they specify what kind of data is being transmitted and how it should be parsed. By specifying a report type and some data, *requests* can be sent to the Wiimote. Through requests, different states like reporting mode can be set. Also Wiimote features like LEDs and rumble can be enabled or disabled. However, since communication goes both ways, there are also report types to specify the format of the data which the Wiimote returns. Table 1, based on [29, 30] lists all of the supported report types. It should be noted that data report `0x30` and `0x33` are written explicitly since these are relevant to our project.

Direction	Report ID	Payload	Function
Output	`0x11`	1	LEDs and rumble
Output	`0x12`	2	Data reporting mode
Output	`0x13`	1	IR camera enable 1
Output	`0x14`	1	Speaker enable
Output	`0x15`	1	Controller status
Output	`0x16`	21	Write memory and register data
Output	`0x17`	6	Read memory and register data
Output	`0x18`	21	Speaker data
Output	`0x19`	1	Speaker mute
Output	`0x1a`	1	IR camera enable 2
Input	`0x20`	6	Status information
Input	`0x21`	21	Read memory and register data
Input	`0x22`	4	Write memory and register status
Input	`0x30-0x3f`	2-21	Data report modes
Input	`0x30`	2	Data report mode - Buttons only
Input	`0x33`	17	Data report mode - Buttons, motion-sensing, IR

Table 1: List of report types supported by the Wiimote, the expected size of the payload in bytes and their function. Note that *Output* refers to packages that are sent from the computer to the Wiimote while *Input* refers to packages from the Wiimote to the computer.

Most of the functions are less relevant to our project. However, one relevant report is the one controlling the data reporting mode (`0x12`). Hereby it is possible to control how and what data is being sent to the host.
The Wiimote has two different ways of transmitting information to the host. As default the controller only sends data to the host when a state has changed, like when a button is pressed. However, it can be set to send data

continuous even if nothing has changed since the last report was sent. This is done with a 10 ms interval, which means that 100 samples are sent per second. This is also the upper bound for the non-continuous report mode.
The data reporting mode also controls what data is being sent from the controller. Several modes are available. The simplest mode `0x30` only sends button information while others send information regarding motion-sensing, expansion port and IR. As shown in Table 1, the report mode `0x33` sends button, motion-sensing and IR data. Since we are interested in receiving IR data as often as possible, this mode combined with continuous reporting is used.

Due to the way communication is done, there are a few things one should be aware of. When receiving packages from the Wiimote, they are stored in a buffer. If they are read at the same rate as they are received, this buffer can be ignored since it will have no influence. However, if the packages are read less frequently, the size of the buffer plays an important role. If the buffer size is low, e.g. 1, then all but the most recent package at each sample point will be dropped and hence never read. This causes loss of data. However, a small buffer size ensures that little or no delay is introduced since it is always the most recent package that is read. On the other hand if the buffer size is large, the situation is quite opposite. Little or no data is lost, but large delays might occur.
Since delays would make it hard to navigate precisely in a 3D world, it must be avoided in our application. Therefore we use a small buffer of size 1. This may result in a few lost packages at times, but it is not critical in our application, since all packages contains the absolute position of the IR points as discussed in section 5.2.2.
Another thing that should be noted is that the status of the rumble engine is send as the least significant bit in every request report. Therefore it is important to make sure that this bit is 0 at all times since we never use the rumble feature.

5.2.1 Writing to Registers

As mentioned the Wiimote has some build in memory which can be read from and written to directly. Additionally several peripherals like the IR camera and speaker has a series of registers which are also available for reading and writing. When one wants to read from the memory or registers a report of the format shown in Figure 7(a) is send.

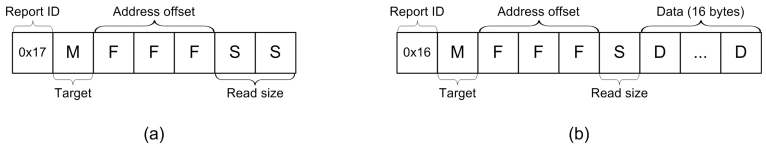

Figure 7: The structure of the (a) read and (b) write request package.

The *target* specifies whether memory or registers should be read, the *address offset* specifies the offset in the address space and *read size* specifies how many bytes one desires to read. The actual data is returned through input port `0x21`. However, we will not go into further details with this since we will not be using the read functionality.
The package format for writing data is very similar as can be seen in Figure 7(b). Besides the report ID, the primary difference is that at most 16 bytes of data can be written at once. These are specified in the 16 last bytes of the package.

5.2.2 Parsing Returned Data

As mentioned earlier, the amount of data returned depends on the selected data reporting mode. In the simplest case, only button data is returned. As specified in Table 1, this mode has report ID `0x30` and uses two bytes of payload containing information about the button status. To identify itself, the returned package also contains the data report type. The structure of the package can be seen in Figure 8.

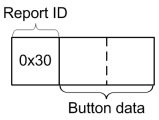

Figure 8: The structure of the data package with data report mode `0x30`.

The first byte contains information about the direction buttons and the + button as seen in Figure 2. The second byte contains information about the rest of the buttons. Since each button is represented by a unique bitmask, a buttons status can be found by making a logical `and` operation between its bitmask and the button data. Since the buttons are of little interest to us, the bitmasks are left out of the report. They are described in detail in

[30, 29, 20].
The other data report mode of interest, i.e. `0x33`, returns a payload of 17 bytes. Besides the button data, this report also contains 3 bytes of data from the accelerometer and 12 bytes from the IR camera. The structure of the package is show in Figure 9.

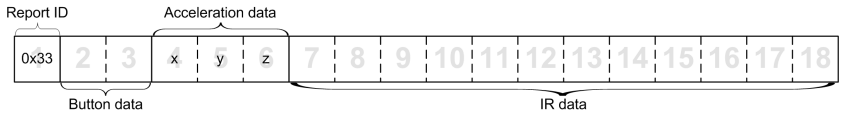

Figure 9: The structure of the data package with data report mode `0x33`.

As one can see, byte 7 - 18 contains all the IR data. However the exact format of the IR data is not directly determined by the data report mode. Instead one of three different IR modes must be selected. These are:

- `Basic mode (0x01)` returns 10 bytes of data, corresponding to the x- and y-coordinates of the four IR points that the camera can detect. These are packed pair wise into groups of 5 bytes as illustrated in Figure 10(a).

- `Extended mode (0x03)` returns 12 bytes of data. Besides the coordinates for each point, a rough size of each point is also sent. In this mode, each point and its size is packed separately into four groups of 3 bytes each as illustrated in Figure 10(b).

- `Full mode (0x05)` returns 36 bytes of data, split up in to separate reports containing 18 bytes each. Two additional elements are appended to each group from the `Extended mode`; a bounding box in pixels of the dot and intensity information about it. This is illustrated in Figure 10(c).

Since it is required that the space for IR data in the report fits the IR mode exactly, `Extended mode` must be used with report type `0x30` described above. To use `Basic` and `Full mode` the data report modes `0x36` and `0x3e` must be used respectively. However, we will not go into details with these two data

report modes since we will be using `Extended mode`. Even though we have no need for the size estimate, we have chosen `Extended mode` since the size of the entire data package is smaller than when using `Basic mode`.
As mentioned, each point is transmitted as an object of 3 bytes when `Extended mode` is used. The structure of the object is illustrated in Figure 10(b).

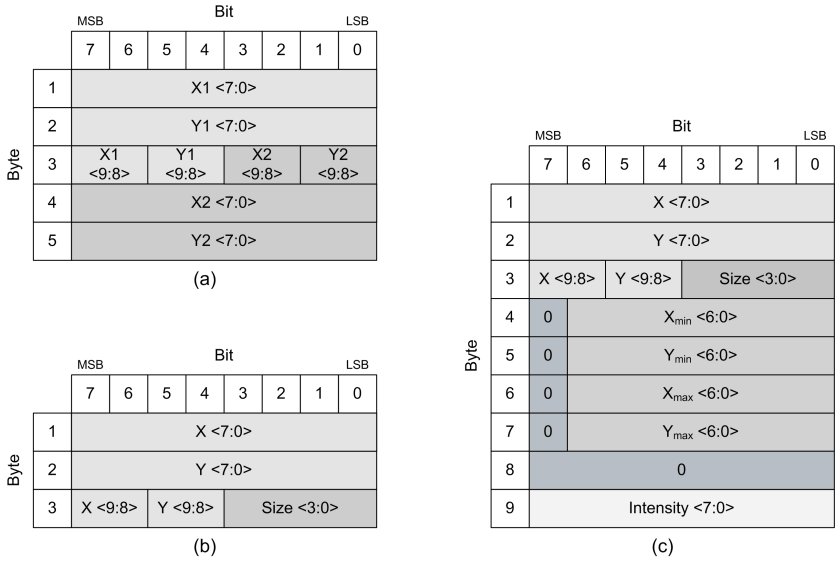

Figure 10: The structure of the object for IR modes (a) `Basic`, (b) `Extended` and (c) `Full`.

The first byte of the object in `Extended mode` contains the 8 least significant bits of the x-coordinate (X<7:0> in the figure) while the 8 least significant bits of the y-coordinate are stored in the second byte. The third byte contains the two most significant bits (bit 9 and 8) for the x- and y-coordinate at bit position 7-6 and 5-4 respectively while bit 3-0 contains the size value of the point. This means that information from the third byte must be appended to the first and second byte in order to get the position of the point. While this mix of information might seem a bit difficult to work with, it packs the bit as tightly as possible and thus is used several places including the bytes containing the accelerometer data.

5.2.3 Initialization of the IR camera

Once a connection is established with the controller, it is possible to initialize the IR camera. First, it is necessary to specify the report mode, that is `0x33`,

and to enable continuous reporting since otherwise IR data is not reported. This is done by sending a report for specifying data reporting mode. The actual content of the package is shown in Figure 11(a).

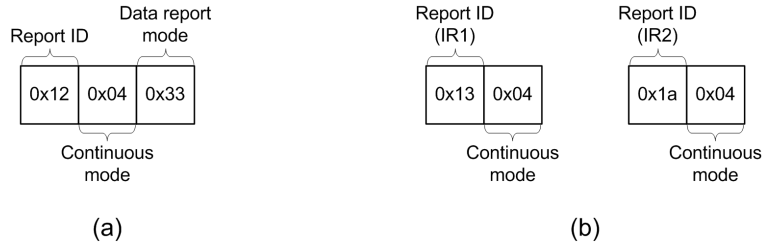

Figure 11: The structure of the package to set report mode (a) and the packages to enable IR sensing (b).

Afterwards the actual sensing of IR is enabled. This is done by sending two packages that specifies continuous reporting for each camera port. These are illustrated in Figure 11(b).
After enabling the camera, it is necessary to specify the IR data mode, that is `Extended mode` in our case, and some sensitivity options that specify how sensitive the camera is to objects. This is done in five steps by writing directly to the registers associated with the camera:

1. Write the value `0x08` to register `0xb00030`.
2. Set the first part of the sensitivity options by writing sensitivity block 1 to the registers at `0xb00000`.
3. Set the second part of the sensitivity options by writing sensitivity block 2 to the registers at `0xb0001a`.
4. Set the IR mode to `Extended mode` by writing the value `0x03` to the register `0xb00033`.
5. Write the value `0x08` to register `0xb00030` once more.

These steps are taken directly from [29]. It is noted that address space `0xb00000 - 0xb00033` is reserved for the IR camera. However there is no information regarding the meaning of each register within the address space

and the value used in step 1 and 5. This is due to the lack of official documentation. As stated, the second and third step specifies the sensitivity of the IR camera. In [29], three different pairs of blocks are suggested. We have chosen to use the `02 00 00 71 01 00 aa 00 64` and `63 03` for block 1 and 2 respectively since we have found at least one other driver implementation using this pair [5]. The only information provided about the blocks is that the last byte in each block determines the required intensity of the IR dots to be accepted.

6 Implementation

In this section we will describe how we have implemented the core features of our program and how it is structured. The source code can be found in appendix C and on the included CD-ROM in the folder `src`. We have also included API documentation of the library, which is available in the folder `docs`.

We have developed a library called WiiIRLib. This handles everything regarding the Wiimote, including communication, using the data, etc. To render the 3D world we have used OpenGL in which we draw the virtual world with simple shapes, such as lines and squares. Since this is not a part of the project, we will not describe it further.

The WiiIRLib is made as a header only implementation. The library makes it possible to use the data from the Wiimote in an easy manner when used in a 3D application. The WiiIRLib consist of three classes, `HIDBridge`, `Wiimote` and `Logger`.

The file `HIDBridge.h` contains the basic functionality which opens and closes the connection and performs writes and reads to the Wiimote. To communicate with the Wiimote as an HID device in C++ in Windows XP the three files `hidsdi.h`, `Setupapi.h` and `windows.h` must be included. `Setupapi.h` and `windows.h` comes for instance with Visual Studio, but `hidsdi.h` must be installed separately and is packed in the Driver Development Kit [1]. `Setupapi.h` is needed because this contains general methods for using a variety of devices like extern hard drives, cameras and HID devices, while `hidsdi.h` manages everything regarding the HID communication. Finally `windows.h` is needed to gain access to general methods for reading and writing data to the device.

The first thing that must be done is to find the Wiimote as a device. In our code this is done with the method `open_device()`. Afterwards the Wiimote must be registered in the program as an HID device. In our code this is done with the method `connect()`. To show the entire code would be too much, but here is an example of the essential elements:

```
1 HANDLE m_handle = NULL;
2
3 unsigned short device_id = 0x0306;
4 unsigned short vendor_id = 0x057E;
5 int device_index = 0;
6
7 for (;;) {
8   HIDD_ATTRIBUTES attrib;
9
10  if (!open_device(device_index) ||
11      !HidD_GetAttributes(m_handle, &attrib)) {
12    break;
13  }
14  if (attrib.ProductID == device_id &&
15      attrib.VendorID == vendor_id) {
16    break;
17  }
18  CloseHandle(m_handle);
19  ++device_index;
20 }
```

As can be seen in the code-example the connection is established by running through all connected devices and connecting to the one that matches the vendorID and deviceID of the Wiimote. Afterwards the Wiimote can be accessed by the variable `m_handle`.
After the connection is established it is possible to write to the Wiimote. In our program this is done in the method `write`. Since it again would be to much to show the entire code, only the essential parts are shown:

```
1 void write(unsigned const char * buffer, int num_bytes) {
2
3   DWORD bytes_written;
4   WriteFile(m_handle, buffer,num_bytes,
5             &bytes_written,&m_overlapped);
6 }
```

Essentially the writing is done by sending a buffer with a handle to the method `WriteFile`, which is part of the underlying API to handle HID devices. Since the operation can fail for a number of reasons our code checks if

anything was written and if the right number of bytes were written. Reading from the Wiimote is done in much the same way.
As mentioned the class `HIDBridge` also contains methods for closing the connection and so forth, but we will not describe them further here.

The second class `Wiimote` includes all the methods that make it possible to change the state (e.g. switching the LEDs on and off) on the Wiimote and get the processed data. All this is done using the methods in the `HIDBridge` class. When using the library, it is therefore only methods from the `Wiimote` class which must be used by the user. The following is an example of how one could turn on the first LED on the Wiimote in the same fashion as is done in the `Wiimote` class.

```
1 static const unsigned char OUTPUT_LED_1       = 0x10
2 static const unsigned char OUTPUT_CHANNEL_LED = 0x11;
3
4 m_output_buffer[0] = OUTPUT_CHANNEL_LED;
5 m_output_buffer[1] = OUTPUT_LED_1;
6 write(m_output_buffer,m_output_buffer_size);
```

The `m_output_buffer` is just a buffer. As can be seen, changing the state of the Wiimote is simply done by writing the report type and the associated action to the Wiimote as described in 5.2.

The most interesting method in `Wiimote.h` is `retrieve_data()`, which reads the latest data from the Wiimote and updates all the variables. It is therefore important to call this method every time the data are used in a new pass, for instance when a new frame is drawn. If this method is called with too short an interval, an error will occur in reading the data. The method therefore has a delay of 1 ms. Besides updating the position of the LEDs, the data needed to set up the camera is also calculated.

The `Logger` class simply serves the purpose of logging calculated data, errors and other convenient messages. The logged information is written to a file specified by the user.

7 Results

This section consists of two main parts. The first part is the verification where we try to verify that the used algorithms produce a reliable result

and in general that our implementation behaves as expected. The second part is the evaluation. Since two of the goals in the project were related to comparing and evaluating our results, we need to do a qualitative evaluation.

7.1 Verification of the program

We have chosen not to construct unit tests for each method in the library since this would yield little information about the actual correctness. Due to the nature of our project, it is also difficult to compare our results with "correct" data. Instead our strategy is to set up a number of situations in controlled environments, thereby allowing us to test separate parts of our implementation without one element influencing another. By examining the precision for example in only one direction at a time, it becomes easier to validate our estimate. Based on this strategy, we have chosen to separate our validation into five different scenarios.

7.1.1 Precision of movement

We need to validate the precision of the measured coordinates. Before we examine the precision during movement, we wish to measure the variation in coordinates when the sensor bar is not moved. This is done to discover any possible lack of precision in the camera. When a possible variation is known, we can examine the precision during movement.
To examine the precision in a single direction, the sensor bar should only be moved in that direction. This allows us to examine whether the coordinates in the other directions are kept constant. Finally by knowing the origin, we can perform a test where we return to check if the measured position for a given sensor bar position is the same after movement.
It should be noted that the measured coordinates are those of the calculated midpoint described in section 4.3.1 and thus not the coordinates for each of the two detected points.
We have made a simplifying assumption in the approach above. We assume that there will be no perspective distortion when the sensor bar is moved away from the middle of camera. This is of course not true. The further away from the centre of the camera, the smaller the distance between the LEDs appear. Ideally we therefore need to move the sensor bar like a pendulum and make sure the LEDs are always facing the camera in the Wiimote. However, this will become very difficult to do precisely. Assuming that we will not move the sensor bar too far away from the centre of the camera, the

distortion will also be minimal.
Based on the discussion above, five separate tests should be made:

- No movement: Theoretically no changes should happen in coordinates, but we expect slight variations due to inaccuracies in the returned data. In the following tests this result is used to define the acceptable threshold when no movement should occur.

- Movement in horizontal direction: Only the x-coordinate should change while the y-coordinate and distance should remain the same.

- Movement in vertical direction: Only the y-coordinate should change and the x-coordinate and distance should be static.

- Movement in depth: The measured distance should change while no change in x- and y-coordinate should occur.

- Coordinate consistency: After a series of random motions with the sensor bar we expect that when we return to the initial position, the coordinates will be the same as before the sensor bar was moved.

To perform the first test, we simply placed the Wiimote and sensor bar on the table, thereby assuring that both units were kept completely still. To perform the three next tests, we made the setup shown at Figure 12.

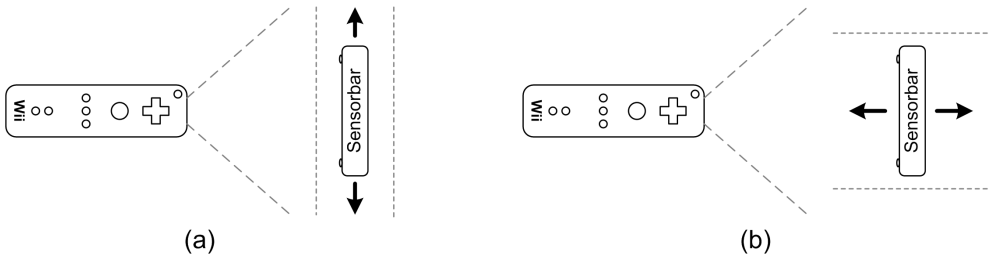

Figure 12: The setup used to validate the precision of the movement in (a) horizontal direction and (b) distance.

As shown in (a), the sensor bar is placed on a table in front of the Wiimote. This makes movement in only one direction easier since no vertical movement can happen. Also, following a measured line helps to insure that no variation happens in distance for the first two tests. To perform the vertical test, we simply place the Wiimote on the side, which allows us to use the same setup. For the distance test shown in (b), we follow two straight lines to ensure that only distance is changed. The setup for the last test is quite similar except that we move the sensor bar freely before returning it to the initial, marked position.

Results

The result of the first test is seen in Figure 13 which shows the reported value of the x- and y-coordinate as well as the distance.The x-axis specifies the samples over time, while the y-axis has no unit since it simply is the data returned from the Wiimote. It should be noted that the distance on all graphs has been scaled to fit within them.
While we expected a little variation in values, it is minimal. In general there are very small variations in the y-coordinate during the whole test. However for the x-coordinate and distance the value is constant with a few exceptions. The largest deviation from the mean value was 0.070%, 0.17% and 0.20% for the x-coordinate, y-coordinate and distance respectively. These values are very small. It is therefore safe to assume that any noteworthy variation in the following experiments is not due to lack of camera precision.

In the following test; movement in the x-direction, the result was overall as expected as seen in Figure 14. There is a fairly constant increase in the x-coordinate, while the y-coordinate and distance are kept fairly constant. The almost linear increase is to be expected since we tried to move the sensor bar with a constant speed.
The small variations are due to our inability to keep the distance completely constant. Despite the effort to follow a straight line, it is difficult to keep the sensor bar perfectly calm during movement.
It is noted that especially the distance has higher values around $200 - 250$ samples. This is due to the perspective distortion discussed earlier. As argued it has little influence on the values, so our assumption about ignoring the distortion was valid.

The test for movement in the y-direction show results very similar to those described above as seen in Figure 15. As expected the x-coordinate and

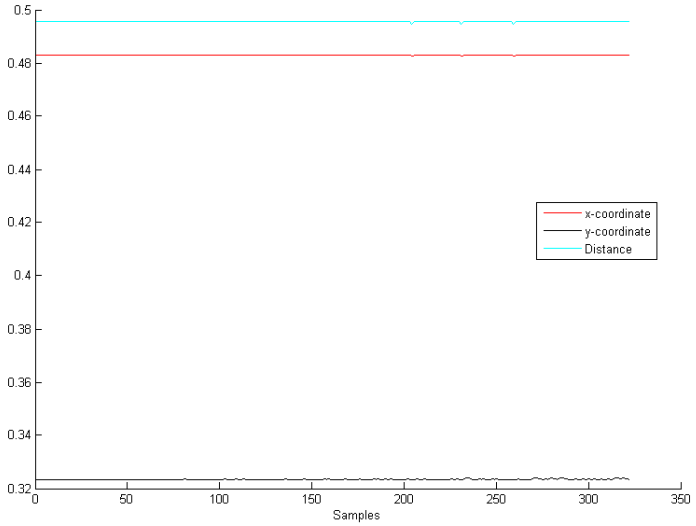

Figure 13: The measured variation in the x-coordinate, y-coordinate and distance.

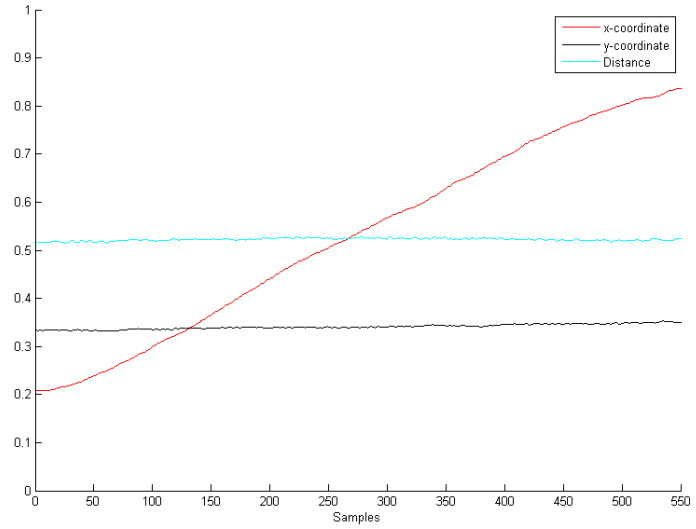

Figure 14: The measured data when only movement in the x-direction occur.

distance are kept fairly constant, while the y-coordinate decreases almost linear. The primary difference is that the distance varied more during this experiment. An explanation to this could be that the vertical resolution of

the camera is lower than the horizontal. This would make variations in the movement appear more clearly.

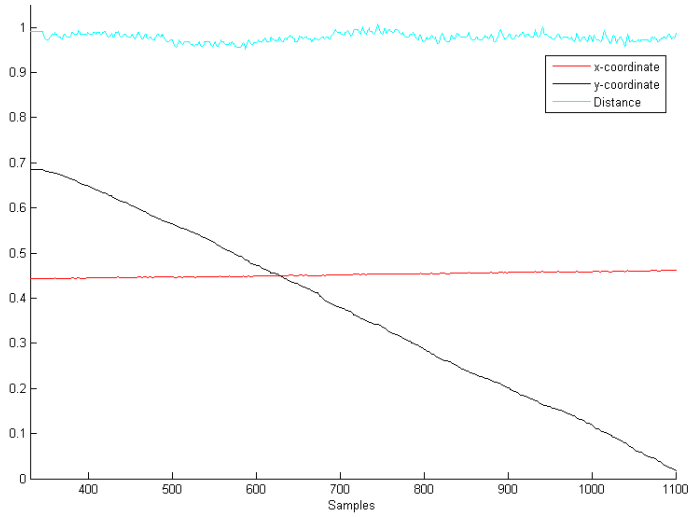

Figure 15: The measured data when only movement in the y-direction occur.

The next test is the distance test. As seen from Figure 16, the result is somewhat different from what we expected. The x-coordinate is close to constant and the distance changes almost linear as anticipated. However the y-coordinate obviously increases over time. After another experiment, we learned that this is due to the fact that the camera is not placed parallel with the front of the Wiimote. Instead it points slightly downwards. This results in an increasing y-coordinate for LEDs placed in the centre of the Wiimote front, when moving the sensor bar backwards.

For the final test, coordinate consistency, the results can be seen in figure 17. As expected the coordinates of the sensor bar are very close before and after the free movement was performed. The measured deviation was 0.095% in the x-coordinate while the y-coordinate was exactly the same. This is absolutely acceptable and we believe the small variations are simply due to us not being able to place the sensor bar on exactly the same spot after the movement.

Overall all five tests returned positive results. There was very little variation when the sensor bar was kept still. Furthermore the values changed as ex-

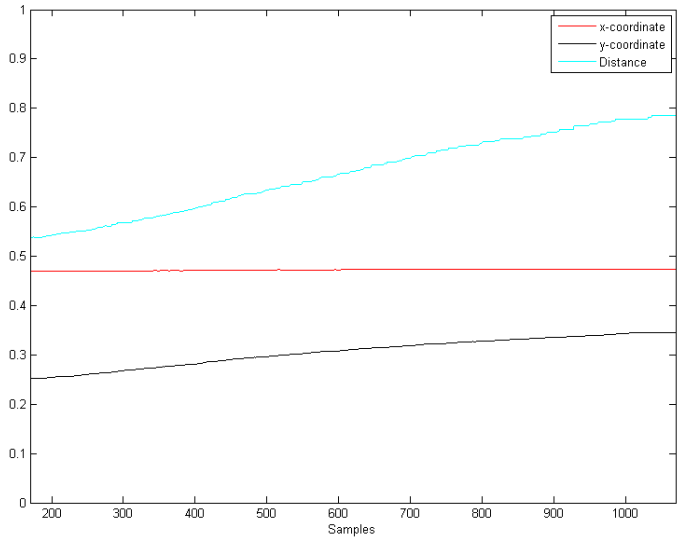

Figure 16: The measured data when only the distance is changed.

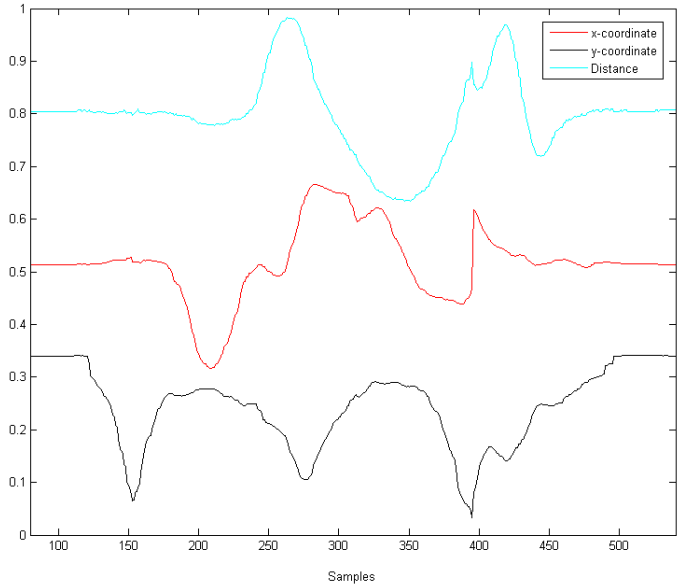

Figure 17: The measured data when the sensor bar is moved freely and returned to the initial position.

pected when performing horizontal and vertical movement. Assuming that we succeeded in moving the sensor bar at a constant speed, the change was also linear. Despite the increasing y-coordinate in the fourth test, we consider it successful as well since the problem was explained. Also, the x-coordinate and distance were as expected. Finally we showed that after free movement, the coordinates were the same as before the sensor bar was moved, when returning to the initial position.

7.1.2 Rotation of the sensor bar

As described in section 4.3.1, we depend on the distance between the two points to estimate the distance of the user. However, this can introduce the problem described in section 4.3.1. If the sensor bar is not facing the Wiimote, the distance between the points becomes smaller. This could happen if the user turns her head away from the monitor. While we speculated that we could ignore this problem as long as the user almost faces the screen, we still need to evaluate the robustness of our solution at thus examine how big a problem it can pose.

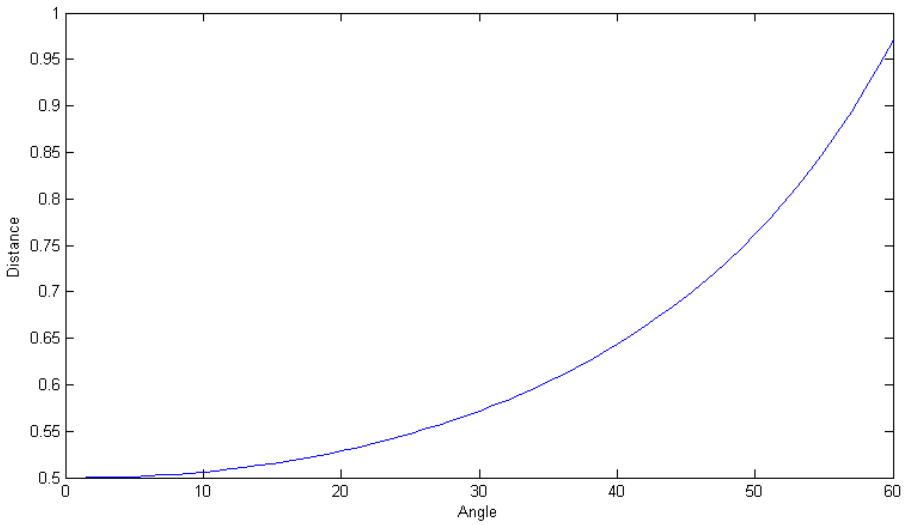

Figure 18: The prediction of the rotation of the sensor bar.

Since we have done nothing to prevent this problem, we expect that the distance should increase quite a lot as the sensor bar is turned. As mentioned earlier the distance is calculated based on the distance between the LEDs.

Since the sensor bar is rotated the function to expect is therefore $\frac{1}{sin(\theta)}$ where θ is the angle from 0 to 60. This function can be seen on Figure 18. Based on the graph, we expect little error in the distance when the sensor bar is close to facing the Wiimote. However, when it is turned more than around 25 degrees, the distance will probably be so erroneous that the estimate is useless.

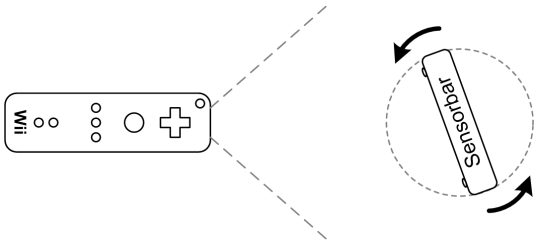

Figure 19: The setup used to examine the error in distance when the sensor bar is rotated.

The setup we used for this experiment is seen in Figure 19. As with the other experiments, it is necessary to isolate the motion we wish to examine. Therefore we must keep a constant distance between the Wiimote and sensor bar. Since we also want to have as little variation as possible in the x- and y-coordinates, we place the sensor bar and Wiimote on a table and rotate the sensor bar around its own centre.

Results

The result of the experiment can be seen on Figure 20. The first thing to note is, that the x- and y-coordinates are almost completely still. The variation on the x and y is 1.4% and 2.5% respectively which is acceptable since we were unable to keep the centre perfectly still while rotating. The measured distance rises as expected and looks like Figure 18. When the angle became above 65-70 degrees the Wiimote was unable to detect both LEDs and the distance was therefore completely wrong. We have chosen not to show this on the figure since the large value would make it impossible to see anything. Overall the result of the test can be seen as a verification of our expectations regarding rotation of the sensor bar.

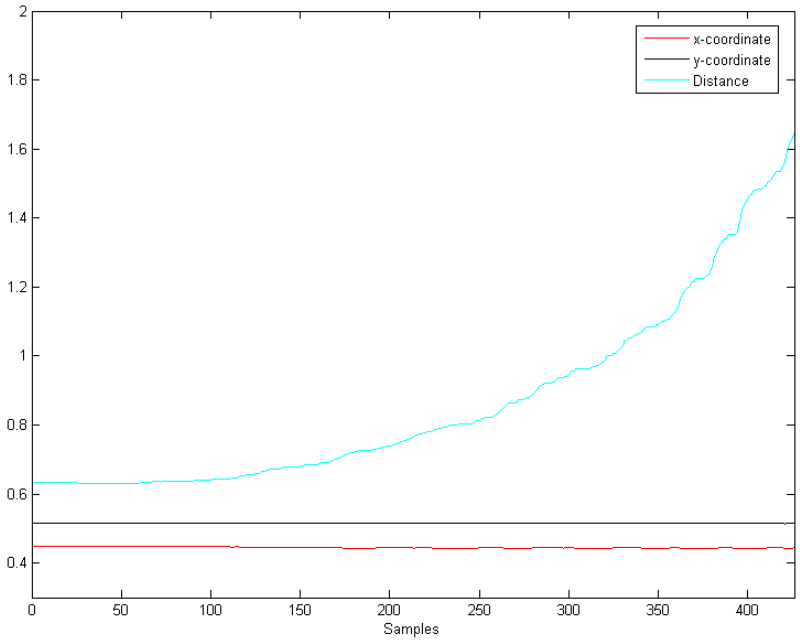

Figure 20: The result of the rotation of the sensor bar.

7.1.3 Robustness of tracking

In section 4.3.3 some of the problems regarding the use of IR were discussed. The problem is what to do if the sensor bar is moved outside the visibility range of the camera or the line of sight is obstructed. In both cases we have tried to handle the situation by keeping the last known valid position until reliable data is available again, as described in section 4.3.3. We then try to make a smooth transition between the two positions.

To verify the robustness of our approach, we have set up two test cases. Opposite to the two previous scenarios, we are not interested in isolating a specific motion. Instead we are interested in the robustness in a general usage scenario. Therefore we have chosen a setup as it is intended to be used. The Wiimote is placed beneath the monitor and the sensor bar is moved freely around, following the user. The setup can be seen in Figure 21.

In the first case, we move the sensor bar outside the visibility field of the Wiimote as shown in Figure 21(a). When entering the field again, we vary

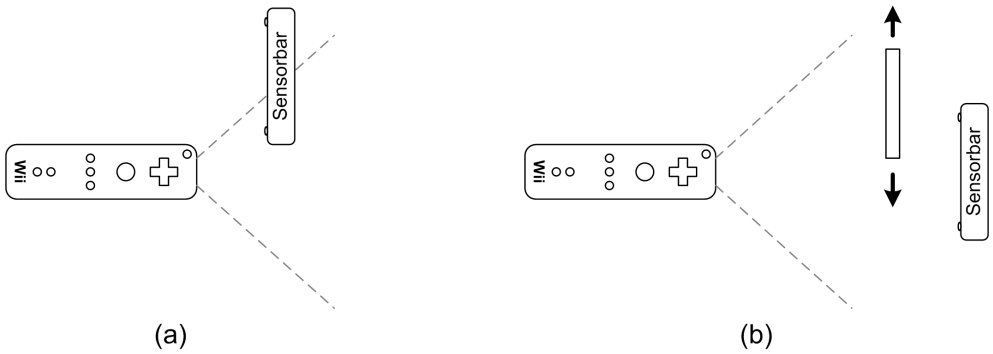

Figure 21: The setup used to examine the robustness when a direct line between the camera and the IR LEDs are lost (a) because the Wiimote is outside the visible field and (b) because of an obstruction.

the entry position so that it differs significantly from the exit point at times. For the second case, we obstruct the line of sight simply by putting a solid object in front of the sensor bar as in Figure 21(b). We then move the sensor bar in different directions before removing the occlusion. By this we simulate minor occlusions like a person passing between the user and the camera and more permanent occlusions like furniture and similar.
In both cases we expect our solution to be robust to the extent that we simply freeze the movement while no new information is available and resume as soon as valid data arrives. Again it is a great advantage that we receive absolute coordinates, as it allows us to quickly change the view according to the position of the user.

Results

In the first experiment where the sensor bar was moved outside of the visibility field, the screen froze as expected. When the sensor bar was moved within sight again, the screen changed perspective to the new position of the user. In section 4.2.2 we said that the transition between the old and new perspective should take around one second. We found, that a time around half a second was more fitting, because the interaction seemed more responsive, but without the perspective jumping from one position to another.
In the second experiment where something passed by the LEDs shortly the result was also as expected. The screen froze for a moment and then continued afterwards. However, a very small jittering could be seen. We found that the reason for this was, that when the object moving in front of the LEDs only covered for instance half of one LED, the detected position of the

LED was moved slightly. Overall this was almost unnoticeable if one was not looking for it.
For both tests the results of the experiments were positive and as expected, which means that the measures we have taken to make our application more robust is working as intended.

7.1.4 Responsiveness

Since our solution should be used for real time interaction, it is important that the response time from an action is performed to the result is shown on the monitor is low. Furthermore one of the primary arguments for doing this kind of interaction was realism, and this will disappear if the response time becomes noticeable.
It is difficult to measure the response time, assuming that it is fractions of a second. If we simply try to measure it with a stopwatch, the result will be inaccurate due to our own reaction time. Therefore we must use a different approach. A possibility is to use a camera recorder. By filming both the sensor bar and the content on the screen, we can capture both elements at once. We can then analyse the captured movie and count the frames from the sensor bar is moved to the content of the screen is changed. With a known frame rate of the movie, we can then calculate the delay in milliseconds. While it will work in theory, it requires a video camera with a high time and spatial resolution to obtain a high precision. If the resolution is too low, it becomes difficult to determine when the content on the monitor changes.
In real time interaction, a low response time is key to ensure smooth operation. Therefore the response time should be below 100 milliseconds. Since our code is fairly light weighted, we expect to be able to do this. To investigate this further, we also include responsiveness as one of the rating criteria during the user evaluation, as described in section 7.2.2.
It should be noted that due to the intentional 1 millisecond delay described in section 6 a slight delay must necessarily occur. However a single millisecond will not be noticeable during interaction.

Results
We made the experiment, where we recorded our movement of the sensor bar and the screen. The camera we used has a frame rate of 15 which means that we can at most get a precision of 7 milliseconds. The result after several experiments is that the response time of our application is 400 milliseconds. Unfortunately this is far too high. We have tried to examine our code to

find what is causing the delay. At first we removed all unnecessary elements. This did not result in a lower response time. Unfortunately we have not been able to locate the exact source of the delay.
Overall we must conclude that the response time is far too high and must be improved before the application can be used properly.

7.1.5 High frequency noise

The purpose of the final scenario is to examine how sensitive our solution is to high frequency noise. We can test the sensitiveness to this problem simply by holding the sensor bar and on purpose make a series of small movements that ideally should be ignored and therefore have no influence on the perspective in the 3D world. However we might end up exaggerating when using this approach. Therefore we will also try simply to put the sensor bar on the head and stand still as one would do in a natural situation. That way we can examine if the noise poses an actual problem during normal usage.
As described in section 4.3.4 we only implemented the simple noise reduction method where we interpolate between the new position of the user and the previous position. Therefore we expect that most of the noise is visible and probably intolerable in the first of the two experiments. However, when placing the sensor bar on the head, we expect less significant problems, since the head can be kept relatively calm in most cases.

Results
In the first test the result was as expected. The jittering of the screen was rather obvious and far too big of a problem to simply overlook. The noise was such a big problem, that the application actually became unusable.
The result for the second experiment was not quite as expected. The noise was less noticeable than in the first test, but it was still visible. The noise does not pose such a big problem, that the application becomes unusable because of it, but it is annoying to look at when the screen is jittering a tiny bit almost all the time.
Overall the result of these two tests showed, that noise is quite a big problem if nothing is done to prevent it.

7.2 Evaluation

As mentioned, we will do a qualitative evaluation of our result. This serves two goals: To evaluate whether we have been able to imitate Johnny Lee's results and to examine if our solution is suitable for navigation in a 3D world. In the first part we ourselves will compare the result demonstrated by Lee in the video [15] with our solution and comment on similarities and differences. We will also give a general evaluation of the result. This is described in section 7.2.1. In the second part, found in section 7.2.2 we will have users evaluate our solution and comment on it.
To illustrate our solution, a video demonstrating the solution in action, has been included on the CD-ROM in the folder `video`. The video was recorded by placing a camera recorder on top of the sensor bar and move it around as a person would.

7.2.1 Comparison with the solution of Johnny Lee

As mentioned above, we will compare our solution to the video [15]. While this will not allow us to do an exact comparison between the solution of Johnny Lee and our own, we have chosen this approach to limit the extent of the project. Since our goal is to examine whether we are able to imitate his solution, there is no need to do a user experiment using his solution, only to identify the features illustrated in the video and to compare them to our solution.
This approach raises a question. To use the video for comparison, it is important that we can trust the content of it. As the video is made by Johnny Lee himself, we can rule out the possibility that some random person has used his program and then tampered with the results. However there is no guarantee that the result presented by Johnny Lee is correct. To investigate this we ran the application and mimicked the movement that he does in the video to assure that we saw the same results as presented in the video. As this turned out to be the case for everything demonstrated in the video, we see no problem in accepting it as trustworthy and thus useable for our comparison.

In general, the video demonstrates the features described in section 4.1, that is adaption of the visible field according to distance and angle. Besides just changing the visible field, these adaptations allow him to look behind objects and even go behind some of them. Also, another interesting feature is pointed out in the video. At times, the foremost target seems to be floating

in front of the screen, giving the illusion that it reaches out into the real world.

To evaluate whether we have been able to achieve these features, we have constructed a 3D scenario much similar to the one used by Johnny Lee. It features a number of floating targets contained within a three dimensional room. A screenshot of the scenario can be seen in Figure 22. As mentioned in section 6 it is rendered using OpenGL.

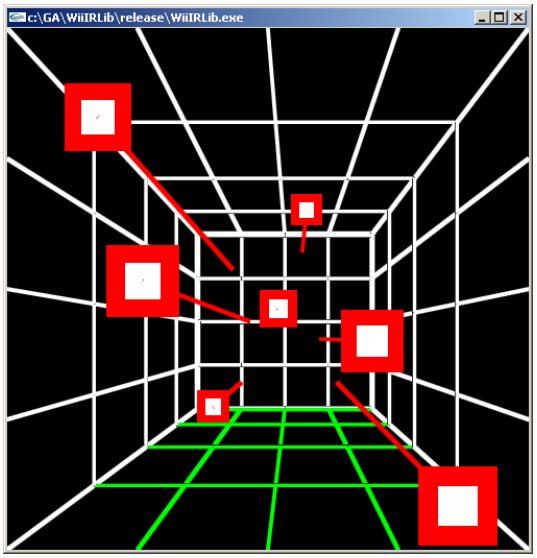

Figure 22: The 3D scenario used in the evaluation of our solution against Johnny Lee's solution.

Results

Overall we found that the solution of Lee and ours had many similarities. This is the case because we originally were inspired be his solution to make this project. However, when comparing his solution to ours we found some elements which were different. In this section we will describe the differences along with a problem we found in his and our solution.

In section 7.1.4 we found that the response time of our solution was 400 milliseconds which was too high. In the solution of Lee the response to the users movement was almost instantly and the delay actually unnoticeable. This is of course far better than our solution. However, the low response time is not just because the work of Lee, but because the library he uses for reading

data from the Wiimote is much faster than our method.

When moving around with the sensor bar the solution of Lee is affected by high frequency noise as ours also is, but in a smaller manner. Lee does nothing to prevent this in his code, but the fact that there is no delay also means that he can update the position of the user more often and thereby reduce the problem of the high frequency noise.

When one or both of the LEDs are undetected the solution of Lee freezes the perspective to the last known. When both LEDs are detected again the perspective is immediately updated to the new position, which often results in the screen making a rather large jump. As we have mentioned several times our solution makes a transition between them. Regarding this, we think our solution is better than Lees.

A problem which is inherent to the approach of making the monitor like a window is that when the user moves closer to the screen more content becomes visible as illustrated in Figure 5, section 4.1.1. This is the case with a normal window and also the case with both the solution of Lee and ours. The problem is that the size of the monitor is fixed and when more content has to be shown, each object has to fill less on the monitor, the closer the user gets to the screen. Because this is a problem inherent when trying to use the monitor as a window, there is no obvious solution.

Regarding some elements the solution of Lee is better than ours, while with other elements it is the other way around. However, overall we must conclude that the solution of Lee performs better than ours. This is especially the case because the delay in our solution is far too high and the high frequency noise is a bigger problem.

7.2.2 User evaluation

In order to get others to evaluate if our solution is suitable for 3D navigation, we have to set up a scenario that they should navigate through. However, we can not just ask participants to look around and tell how they feel since this would yield little insight. Therefore we will design a task that should be solved. This will make the participants engage actively in the navigation and not just "play around". Afterwards we will ask them to evaluate the navigation based on 5 different criteria. These are ease of use, fun to use, intuitiveness, level of realism and responsiveness. They will also be asked to

comment on the experience. The questionnaire used can be seen in appendix A.

The task scenario is set in a simple 3D world very similar to the one used in the first part of the evaluation. A screenshot can be seen in Figure 23. The participants must then look around and find a number of hidden squares behind the targets. Since they are behind the targets, they are not visible from the initial point of view. The user must therefore look around and move to find the squares.

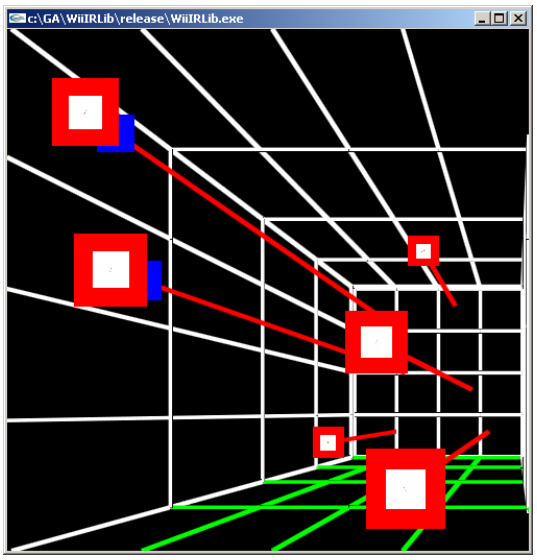

Figure 23: The 3D scenario used in the evaluation of our solution using participants.

In general many considerations should be done when designing a complete experiment and thus it easily becomes an entire project. Since planning of the experiment is not the primary focus in this project, we will only briefly describe the considerations done during the construction of the experiment. This limitation also means that we are not considering several design issues like methods for selecting users and making sure that they have the same preferences.
As mentioned, we have chosen to let the participants perform a task instead of just playing around. This is done since it is known that participants tend to loose interest very fast when they are not performing specific tasks [22].
The rating is done using a questionnaire based on a 6-point explicit numeric scale as described in [22]. We have chosen scales since it makes interpreta-

tion of the data easy. It also helps avoid bias due to leading questions. The reason for choosing a 6-point scale is to avoid users giving completely neutral answers.
Finally it should be noted that we try to be as neutral as possible during the experiment and thus interfere as little as possible. This is done to avoid bias due to us affecting the users. Because of that, we only interfere if the participants are having significant problems understanding the navigation or finding a certain mark.
We conducted our experiment with three participants. Two of them were females aged 20 and 57 and the male was 20. All three were regular users of computers and different kinds of interfaces, but none of them had previous experience with VR like interfaces.

Results
The answers to the questionnaires can be seen in appendix B. Since the interviews were performed in Danish, the text answers are our notes about what the users said.
A summery of the scores from the rating based part is shown in Table 2. In general the score is above average, with four out of five statements receiving an average score around 4. The only statement getting a higher score is the intuitiveness, with an average of 5.0. Overall there are quite some variations in the score given between the users, which makes it difficult to conclude much. However, an interesting observation is that the average score on responsiveness is 4. One would have expected worse ratings considering the results found during the verification. However, it seems to be acceptable according to the users.

Statement	User 1	User 2	User 3	Average
The interface was easy to use	4	2	6	4
The interface was fun to use	2	5	6	4.3
The interface was intuitive	5	6	4	5
The perspective was like looking through a window	4	3	6	4.3
The program responded immediately	4	3	5	4

Table 2: The scores given during the rating of the five statements. Included is also the average score for each statement.

When taking the answers to the three questions into account, a much more general opinion about our solution is seen: The idea of using the head to

change the perspective is good. It can add realism and fun to the application and would probably work well in a game. One user even speculated that this approach could improve working methods and help avoid problems like mouse-related injuries. The users also found the form of interaction intuitive which also can be seen from the ratings.

Despite the potential there were a number of serious issues, stopping it from being really useable, which also shows by the fact that two of the users preferred a conventional mouse. One of the reasons for this was the delays experienced when moving fast. Another problem was the high sensitivity. Even very small motions were transferred to the screen. This conforms very well with the observations about the high frequency noise problems discussed during the verification.
Another major problem mentioned was the instability. Because it is fairly easy to move the sensor bar outside the range of the Wiimote, the application easily locks. The primary reason for this is the relatively small field of view of the camera, especially when standing close to the Wiimote. This problem made one of the users report that the application did not run smoothly and he thought this was due to a low frame rate.
Finally a common problem was that there is little freedom when looking and moving around. Two of the users felt locked and unable to move as they would like at times. While one user was unable to move the desired amount sideways, another felt constrained in up- and downward movement.
It is noteworthy that none of the participants commented on the problem with the perspective described in section 7.2.1 when moving closer and farther from the screen.

8 Reflections

During the project, an interesting announcement on the game related news site Joystiq [12] came to our attention. At the Game Developers Conference (GDC) 2008 [6], it was revealed that the game Boom Blox™ [7] for the Wii console will feature the possibility to control the camera using headtracking in the same way as we have done, by using a Wiimote and some IR LEDs. However it was pointed out that the feature should be considered an Easter egg due to the requirement of a "do-it-yourself IR LED headset" and thus is not an official game feature. Despite that it is not official, it shows that the game industry has also taken interest in the headtracking possibilities of the Wiimote and that the feature could become part of games at some point.

Several other interesting things have happened in the area of VR like interaction in games during the project. An interesting piece of hardware also shown at GDC 2008 was the 3DV Systems ZCam™ [28]. Basically it is a camera with a specialized chip, capable of determining the depth of objects in real time using infrared technology, which is not just capable of detecting IR LEDs, but IR light in general. This means that it can extract the shape of a person standing in front of the camera and map her movement directly to a game. At GDC this was illustrated using a boxing game where the player simply punches into the air and this action is mapped directly into the game where the character then punches. To avoid a punch from the opponent, one simply dodges in the real world and the game character then follows. From a video of the demonstration [9], it seems very convincing and is a huge leap towards VR like game experiences. Although this does not directly aim at headtracking, it is not unlikely that the technology can be adapted to achieve results similar to the ones we have presented due to the high resolution depth map it can produce.
Finally another noteworthy piece of hardware announced during our project is the Neural Impulse Actuator (NIA) [10]. It is a device that allows you to control e.g. games using only your mind. It works by measuring several elements from a headband, including activities of the brain, nervous system and muscles [18]. These data are then sent to the PC for interpretation. While this seemingly has little with games to do, it has been demonstrated how the game Unreal Tournament III™ [8] can be controlled using the device [18]. While it might not directly provide a VR like game experience, it opens for a whole new way of game interaction. One could also imagine that it could enable the user to change the visible area in a 3D world according to her position like we do, but without using actual headtracking.

The ideas behind the above mentioned products are not new and similar products have been shown previously. The interesting aspect is that these solutions are at a much lower price than those previously shown. The price actually is so low that we can expect to see them in the mass market in the near future. Therefore it is necessary to discuss if our solution has the potential to be used on the mass market.
The setup needed to use this headtracking is a Wiimote and some form of IR LEDs. As we already mentioned in section in the introduction, the price of the Wiimote is so low that is can be used on a mass market which it already is. In our setup we used the sensor bar from the Wii as IR LEDs, but this is not a good solution since the user has to hold it on her head. Instead the LEDs should be mounted on the side of a pair of glasses. This way the user

can move freely around. We tried to construct a small device with the LEDs on. It was rather easy to construct and only cost a few dollars. The problem with this is off course that the user does not want to do this and the tracking was not as robust as when using the sensor bar. It did however show that it is possible to produce a solution at a very low price.
Overall we learned that using the Wiimote to perform headtracking is possible and could be used on the mass market. The user evaluation showed that the users found the interface fun to use. However, it also showed that our solution still have some issues which should be addressed and further development is needed before the solution is final.

8.1 Other applications

During the project we have focused on using the headtracking results for VR like interaction in 3D worlds. However, there exist other applications of this type of headtracking. In [17] they compare different techniques for object selection in augmented reality. Here one of the problems that should be overcome is when objects are outside the field of view of the user. Different approaches are suggested, but a possibility not suggested would be to adapt our solution so that when the user turns her head, other objects become visible. Since a Wiimote is already used in one of the techniques described, the necessary type of equipment is already in use.

While augmented reality is still primarily of interest in research, there exist other, more widely used applications. Headtracking does not only have to be used in 3D applications. Some of the most common content in programs is 2D, like text and images. Many of these programs contain more information than is visible on a single screen. This means that a lot of scrolling or zooming is often necessary. Just take a 20 page text document or architectural drawings as examples. Instead of having to scroll using a mouse, our solution could be used to move the visible area. That way the user gets more freedom since this task is moved from the mouse which is not always active, to the head. An architect would for instance be able to move to the left part of the blueprint, simply by turning her head to the left.
While our current approach is not optimal for this kind of usage, it could be made so with only a few changes. By expecting the user to be in a fixed position we could interpret changes in coordinates as rotation of the head instead of movement of the user. That way the user could sit in front of the screen and turn her head instead of moving around.

9 Conclusion

The purpose of this project was to mimic the work of Johnny Lee and implement a solution for performing headtracking using the Wiimote. The goal of the headtracking was to allow users to interact with a 3D world and give much of the same feeling as when looking through a window.
In this paper we described the Wiimote and the how communication with it is performed as well as how the returned data can be used to navigate in a 3D world. We also made an analysis of the problem as well as of the solution of Johnny Lee. Here we discussed his approach and aspects that could be improved; namely robustness and noise reduction. Based on the analysis, we described our solution and made suggestions to improve the robustness and noise problems.
Afterwards we performed verification and an evaluation of our solution. The verification generally went as expected, showing the precision and robustness we had foreseen. However, it turned out that high frequency noise is a larger problem than expected. Also the responsiveness was significantly worse than anticipated.
In the evaluation, we compared our solution to the one of Johnny Lee. Generally the two solutions acted similar, but our solution used interpolation to improve the robustness which the solution of Lee did not. Overall though, his solution performed better than ours, especially regarding responsiveness and tolerance to high frequency noise.
The user study showed that the presented idea works well and could potentially be suitable for interaction in a 3D world. The users found the navigation intuitive and it has high potential. Despite the high potential our solution suffers from a few problems such as high response time and high frequency noise. Before our solution can be used for actual interaction further work is therefore needed.

10 Future work

There are some relevant areas of this project that we have not had time to explore thoroughly. One of these is the implementation of high frequency noise reduction discussed in section 4.3.4. As revealed by the verification experiments, our solution is relatively sensitive to high frequency noise, especially if the head is not kept calm. Therefore it would be interesting to implement both the discussed prediction of the user's movement and the Gaussian smoothening of the samples. That would allow us to examine which approach

is the most appropriate and whether the noise can be removed without sacrificing responsiveness.

Another interesting aspect would be to add two-player support on e.g. a split screen. This is relevant since many games for consoles like the Wii have multiplayer support on the same monitor.
The most direct approach is to connect two different Wiimotes and let each track a sensor bar. However this would require some way to determine which sensor bar belongs to which Wiimote. While this can be specified by the user before entering a game, it becomes more complicated if both sensor bars disappear from the visible field and then returns. Another drawback is the requirement of two Wiimotes which means that the solution becomes somewhat more expensive.
Instead one can take advantage of the ability of the Wiimote to track four points at once. That will require only an extra sensor bar which is somewhat cheaper. However the problem of tracking pairs of points, especially if they are not visible for periods becomes more complex. Besides being more difficult to determine which points are associated to which user, one must expect that obstruction will happen more frequently since users might get in each others way. This implies that just locking the view in case of obstructions might not be acceptable and a better solution is required.

Finally it would be interesting to do a more thorough user evaluation. As mentioned in section 7.2.2, we have ignored several aspects in empirical research to limit the extent of the project. Performing a more thorough evaluation would make it easier for others to compare results. Also an obvious possibility is to include other interaction methods in the evaluation. This would allow us to examine if our solution actually provides a better form of interaction than other devices like game controllers, mice and similar regarding both qualitative and quantitative aspects.
Another interesting aspect would be to examine how suited our solution is for several hours of interaction. Since it is not uncommon to e.g. play a game for two or three hours in a row, this is relevant. While our solution might be realistic, one can imagine that it has higher physical requirements of the user since she must constantly be moving around. This is not the case for conventional controllers and thus is worth examining.

References

[1] Microsoft Corporation. Windows Server 2003 SP1 Driver Development Kit. www.microsoft.com/whdc/devtools/ddk/default.mspx.

[2] Valve Corporation. Half-Life 2, 2004. orange.half-life2.com/hl2.html.

[3] Spark Fun Electronics. Wii-mote Guts, 2006. http://www.sparkfun.com/commerce/present.php?p=Wii-Internals.

[4] James D. Foley, Andries van Dam, Steven K. Feiner, and John F.Hughes. *Computer Graphics - Principles And Practice*. Addison Wesley, 2nd edition edition, 1996.

[5] Kevin Forbes and Giancarlo Todone. cWiiMote 0.3. www.ecando.it/faceplate.html.

[6] Game Developers Conference 2008, 2008. www.gdconf.com.

[7] Electronic Arts Inc. Boom Blox, 2008. www.ea.com/boomblox/.

[8] Epic Games Inc. Unreal Tournament, 2007. www.unrealtournament3.com.

[9] IGN Entertainment Inc. The Wii Killer, 2008. blogs.ign.com/Blogs/Comments.aspx?blog=Matt-IGN&entryid=82107.

[10] OCZ Technology Inc. Neural Impulse Actuator, 2008. www.ocztechnology.com/aboutocz/press/2008/274.

[11] IVT. BlueSoleil. www.ivtcorporation.com/products/bluesuit/index.php.

[12] Joystiq. GDC08: Boom Blox to include head tracking. Seriously., 2008. www.joystiq.com/2008/02/21/gdc08-boom-blox-to-include-head-tracking-seriously/.

[13] Linden Lab. Second Life. secondlife.com.

[14] Johnny Lee. www.johnnylee.net.

[15] Johnny Lee. Head Tracking for Desktop VR Displays using the WiiRemote, 2007. www.youtube.com/watch?v=Jd3-eiid-Uw.

[16] Johnny Lee. WiiDesktopVR, 2007. www.cs.cmu.edu/ johnny/projects/wii/.

[17] Julian Looser, Mark Billinghurst, Raphaël Grasset, and Andy Cockburn. An evaluation of virtual lenses for object selection in augmented reality. In *GRAPHITE '07: Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*, pages 203–210. ACM, 2007.

[18] Overclock3D Ltd. OCZ prepares Neural Impulse Actuator for shipping next week, 2008. www.overclock3d.net/news.php?/input_devices/ocz_prepares_neural_impulse_actuator_for_shipping_next_week/1.

[19] Nintendo®. www.nintendo.com.

[20] Lasse Jon Fuglsang Pedersen and Anders Sabinsky Tøgern. Styring af objekter i 3D vha. Wii-controller, 2007. Under graduate report at DIKU.

[21] Brian Peek. WiimoteLib. www.codeplex.com/WiimoteLib.

[22] Ralph L. Rosnow and Robert Rosenthal. *Beginning Behavioral Research - A Conceptual Primer*. Prentice Hall, 5th edition edition, 2005.

[23] Torben Schou and Henry J. Gardner. A wii remote, a game engine, five sensor bars and a virtual reality theatre. In *OZCHI '07: Proceedings of the 2007 conference of the computer-human interaction special interest group (CHISIG) of Australia on Computer-human interaction: design: activities, artifacts and environments*, pages 231–234. ACM, 2007.

[24] SGI. OpenGL. www.opengl.org.

[25] Akihiko Shirai, Erik Geslin, and Simon Richir. 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*, pages 133–140. ACM, 2007.

[26] Bluetooth SIG. Bluetooth. www.bluetooth.org.

[27] Sreeram Sreedharan, Edmund S. Zurita, and Beryl Plimmer. 3D input for 3D worlds. In *OZCHI '07: Proceedings of the 2007 conference of the computer-human interaction special interest group (CHISIG) of Australia on Computer-human interaction: design: activities, artifacts and environments*, pages 227–230. ACM, 2007.

[28] 3DV Systems. ZCam, 2008. www.3dvsystems.com/technology/product.html.

[29] WiiBrew.org. www.wiibrew.org.

[30] WiiLi.org. www.wiili.org.

A Questionnaire

Please rate the experience with the interface by the following criteria on a scale from 1 to 6 by circling the appropriate value. 1 is strongly disagree and 6 is strongly agree.

I found that the interface was easy to use:

Strongly disagree 1 2 3 4 5 6 Strongly agree

I found that the interface was fun to use:

Strongly disagree 1 2 3 4 5 6 Strongly agree

I found that the interface was intuitive:

Strongly disagree 1 2 3 4 5 6 Strongly agree

I found that the perspective was like looking through a window:

Strongly disagree 1 2 3 4 5 6 Strongly agree

I found that the interface responded immediately:

Strongly disagree 1 2 3 4 5 6 Strongly agree

Do you prefer this over a normal mouse-interface? Why /why not?

What problems did you notice?

Further comments:

B Questionnaire Results

User 1

Please rate the experience with the interface by the following criteria on a scale from 1 to 6 by circling the appropriate value. 1 is strongly disagree and 6 is strongly agree.

I found that the interface was easy to use:

Strongly disagree 1 2 3 4 5 6 Strongly agree

I found that the interface was fun to use:

Strongly disagree 1 2 3 4 5 6 Strongly agree

I found that the interface was intuitive:

Strongly disagree 1 2 3 4 5 6 Strongly agree

I found that the perspective was like looking through a window:

Strongly disagree 1 2 3 4 5 6 Strongly agree

I found that the interface responded immediately:

Strongly disagree 1 2 3 4 5 6 Strongly agree

Do you prefer this over a normal mouse-interface? Why /why not?

This interaction is not preferred. This due to the delay and the fact that it does not always reaction as expected.

What problems did you notice?

It does not always move in the desired directions as expected. The boxes are not always following when moving in distance.

Further comments:

User 2

Please rate the experience with the interface by the following criteria on a scale from 1 to 6 by circling the appropriate value. 1 is strongly disagree and 6 is strongly agree.

I found that the interface was easy to use:

Strongly disagree 1 2 3 4 5 6 Strongly agree

I found that the interface was fun to use:

Strongly disagree 1 2 3 4 5 6 Strongly agree

I found that the interface was intuitive:

Strongly disagree 1 2 3 4 5 6 Strongly agree

I found that the perspective was like looking through a window:

Strongly disagree 1 2 3 4 5 6 Strongly agree

I found that the interface responded immediately:

Strongly disagree 1 2 3 4 5 6 Strongly agree

Do you prefer this over a normal mouse-interface? Why /why not?

Mouse interaction is preferred.

What problems did you notice?

The sensitivity was too high. The possibilities to move sideways are limited.

It seemed not to run smooth, due to a low frame rate. It is unhandy with the sensor bar on the head.

Further comments:

The responsiveness rating applies when the perspective does not lock.

There was some delay.

User 3

Please rate the experience with the interface by the following criteria on a scale from 1 to 6 by circling the appropriate value. 1 is strongly disagree and 6 is strongly agree.

I found that the interface was easy to use:

Strongly disagree 1 2 3 4 5 6 Strongly agree

I found that the interface was fun to use:

Strongly disagree 1 2 3 4 5 6 Strongly agree

I found that the interface was intuitive:

Strongly disagree 1 2 3 4 5 6 Strongly agree

I found that the perspective was like looking through a window:

Strongly disagree 1 2 3 4 5 6 Strongly agree

I found that the interface responded immediately:

Strongly disagree 1 2 3 4 5 6 Strongly agree

Do you prefer this over a normal mouse-interface? Why /why not?

This interaction is better than with a mouse, but it will probably require some practice.

What problems did you notice?

It was annoying that it was not possible to see the hidden squares furthest away.

Further comments:

Perhaps it will improve the methods for working over a mouse. You could possibly avoid mouse-related injuries. In games one will probably be able to react faster since the coordination between eye and hand is avoided.

C Source code

C.1 HIDBridge.h

```
1  #ifndef WIIIRLIB_HIDBRIDGE_H
2  #define WIIIRLIB_HIDBRIDGE_H
3
4  #include <windows.h>
5
6  extern "C" {
7    #include <hidsdi.h>
8    #include <Setupapi.h>
9  }
10
11 #pragma comment(lib, "setupapi.lib")
12 #pragma comment(lib, "hid.lib")
13
14
15 namespace WiiIRLib {
16
17   /**
18    * HIDBridge manages the connection to a HID device like the Wiimote.
19    * The class is based on cWiiMote 0.3 which is available at
20    * http://www.ecando.it/cwmdown.html
21    */
22   class HIDBridge {
23
24
25   private:
26     HANDLE m_handle;
27     HANDLE m_event;
28     OVERLAPPED m_overlapped;
29     bool m_connected;
30
31
32   public:
33     HIDBridge() : m_connected(false), m_handle(NULL), m_event(NULL) {};
34
35
36
37     /**
38      * Upon destruction, the HIDBridge closes the connection.
39      */
40     ~HIDBridge() {
41
42       if (m_connected) {
43         disconnect();
44       }
45     }
46
47
48
49     /**
50      * disconnect closes the connection to the HID device.
51      * @return - True if the connection was closed, false otherwise.
52      */
53     bool disconnect() {
54
55       bool success = false;
56       if (m_connected) {
```

```
57          success = (CloseHandle(m_handle) && CloseHandle(m_event));
58          m_connected = false;
59        }
60        return success;
61      }
62
63
64
65      /**
66       * connect esabilishes a connection to a HID device with the specified
67       * device and vendor ID. If nothing is specified, the IDs for the
68       * Wiimote will be used.
69       * @param device_id - The ID of the device.
70       * @param vendor_id - The ID of the vendor.
71       * @param devices - Specifies the number of devices to be found.
72       * @return - True if a connection was estabilished, false otherwise.
73       */
74      bool connect(unsigned short device_id = 0x0306,
75                   unsigned short vendor_id = 0x057e, int devices = 0) {
76
77        if (m_connected) {
78          if (!disconnect()) {
79            return false;
80          }
81        }
82
83        bool success = false;
84        int device_index = 0;
85        int matching_devices_found = 0;
86
87        for (;;) {
88          // Go through all HID devices until a matching device is found.
89          HIDD_ATTRIBUTES attrib;
90
91          if (!open_device(device_index) ||
92              !HidD_GetAttributes (m_handle, &attrib)) {
93            break;
94          }
95          if (attrib.ProductID == device_id && attrib.VendorID == vendor_id) {
96            // The desired device has been found.
97            if (matching_devices_found == devices) {
98              m_connected = true;
99              break;
100           }
101           matching_devices_found++;
102         }
103         CloseHandle(m_handle);
104         device_index++;
105       }
106       return m_connected;
107     }
108
109
110
111     /**
112      * is_connected returns the connection status.
113      * @return - True if connected, false otherwise.
114      */
115     bool is_connected() const {
116
117       return m_connected;
118     }
```

```
119
120
121
122     /**
123      * write writes the specified data to the HID device.
124      * @param buffer - The data to be written.
125      * @param num_bytes - The number of bytes to be written.
126      * @return - True if all bytes were written, false otherwise.
127      */
128     bool write(unsigned const char * buffer, int num_bytes) {
129
130       bool success = false;
131       if (m_connected) {
132         DWORD bytes_written;
133         success = WriteFile(m_handle,buffer,num_bytes,
134                             &bytes_written,&m_overlapped) == TRUE;
135         success = success && bytes_written == num_bytes;
136       }
137       return success;
138     }
139
140
141
142     /**
143      * read reads the data returned from the HID device.
144      * @param buffer - The target for the data.
145      * @param max_bytes - The maximum number of bytes that should be read.
146      * @param bytes_read - Specifies the number of bytes read on return.
147      * @param timeout - The number of ms. before the function times out.
148      * @return - True if the data was read, false otherwise.
149      */
150     bool read(unsigned const char * buffer, int max_bytes,
151                                         int & bytes_read, int timeout) {
152
153       bool success = false;
154       if (m_connected) {
155         ReadFile(m_handle,(LPVOID)buffer,max_bytes,
156                           (LPDWORD)&bytes_read,&m_overlapped);
157         DWORD Result = WaitForSingleObject(m_event, timeout);
158         if (Result == WAIT_OBJECT_0) {
159           success = true;
160         } else {
161           CancelIo(m_handle);
162         }
163         ResetEvent(m_event);
164       }
165       return success;
166     }
167
168
169
170   private:
171
172     /**
173      * open_device opens the HID device with the specified index.
174      * @param index - The index of the device.
175      * @return - True if the device was opened, false otherwise.
176      */
177     bool open_device(int index) {
178
179       bool success = false;
180       GUID guid;
```

```
181         HidD_GetHidGuid (&guid);
182         HDEVINFO devinfo = SetupDiGetClassDevs(&guid,NULL,NULL,
183                                         DIGCF_PRESENT | DIGCF_DEVICEINTERFACE);
184
185         SP_DEVICE_INTERFACE_DATA dev_int_data;
186         dev_int_data.cbSize = sizeof(SP_DEVICE_INTERFACE_DATA);
187
188         if (SetupDiEnumDeviceInterfaces(devinfo,NULL,&guid,
189                                             index,&dev_int_data) == TRUE) {
190           DWORD size;
191           SetupDiGetDeviceInterfaceDetail(devinfo,&dev_int_data,
192                                                       NULL,0,&size,0);
193
194           PSP_INTERFACE_DEVICE_DETAIL_DATA detail =
195                               (PSP_INTERFACE_DEVICE_DETAIL_DATA)malloc(size);
196           detail->cbSize = sizeof(SP_INTERFACE_DEVICE_DETAIL_DATA);
197
198           if (SetupDiGetDeviceInterfaceDetail(devinfo,&dev_int_data,
199                                                   detail,size,NULL,NULL)) {
200             m_handle = CreateFile (detail->DevicePath,
201                                    GENERIC_READ | GENERIC_WRITE,
202                                    FILE_SHARE_READ | FILE_SHARE_WRITE,
203                                    NULL, OPEN_EXISTING,
204                                    FILE_FLAG_OVERLAPPED, NULL);
205             // Set the buffer size to 1
206             HidD_SetNumInputBuffers(m_handle, 1);
207             m_event = CreateEvent(NULL, TRUE, TRUE, "");
208             m_overlapped.Offset = 0;
209             m_overlapped.OffsetHigh = 0;
210             m_overlapped.hEvent = m_event;
211             success = true;
212           }
213           free(detail);
214         }
215         SetupDiDestroyDeviceInfoList (devinfo);
216         return success;
217       }
218
219     };
220   } // namespace WiiIRLib
221   // WIIIRLIB_HIDBRIDGE_H
222   #endif
```

C.2 Logger.h

```
1   #ifndef WIIIRLIB_LOGGER_H
2   #define WIIIRLIB_LOGGER_H
3
4   #include <iostream>
5   #include <fstream>
6
7   namespace WiiIRLib {
8
9     /**
10     * Logger is a class that logs program information, errors and
11     * IR point data.
12     */
13    class Logger {
```

```
14
15    private:
16      std::ofstream m_target_file;
17      bool m_log_messages;
18      bool m_log_errors;
19      bool m_log_positions;
20      bool m_matlab_mode;
21      bool m_logging_enabled;
22
23    public:
24
25      Logger() {
26
27        m_logging_enabled = false;
28        m_log_messages = false;
29        m_log_errors = false;
30        m_log_positions = false;
31        m_matlab_mode = false;
32      }
33
34
35
36      ~Logger() {
37
38        if (m_matlab_mode) {
39          m_target_file << "];";
40        }
41        m_target_file.close();
42      }
43
44
45
46      /**
47       * init sets up the logger. As default only positions are logged.
48       * @param file - The path and file name of the target file.
49       * @param matlab_mode - True if positions should be logged in Matlab
50       * format for graph drawing. If enabled, only positions will be logged.
51       * @param messages - True if messages should be logged, false otherwise.
52       * @param errors - True if errors should be logged, false otherwise.
53       * @param positions - True if positions should be logged,
54       * false otherwise.
55       */
56      void init(std::string file, bool matlab_mode, bool messages = false,
57                bool errors = false, bool positions = true) {
58
59        m_target_file.open(file.c_str());
60        m_matlab_mode = matlab_mode;
61        m_log_messages = messages && !m_matlab_mode;
62        m_log_errors = errors && !m_matlab_mode;
63        m_log_positions = positions || m_matlab_mode;
64        if (m_matlab_mode) {
65          m_target_file << "A = [";
66        }
67      }
68
69
70
71      /**
72       * toggle_logging enables or disables logging to a file.
73       * @return - True if logging is enabled, false otherwise.
74       */
75      bool toggle_logging() {
```

```
76
77        return m_logging_enabled = !m_logging_enabled;
78      }
79
80
81
82      /**
83       * log_message logs the specified message if message logging is enabled.
84       * @param message - The message to log.
85       */
86      void log_message(std::string message) {
87
88        if (m_log_messages && m_logging_enabled) {
89          m_target_file << "Message: " << message.c_str() << std::endl;
90
91        }
92      }
93
94
95
96      /**
97       * log_error logs the specified error if error
98       * logging is enabled.
99       * @param error - The error to log.
100      */
101     void log_error(std::string error) {
102
103       if (m_log_errors && m_logging_enabled) {
104         m_target_file << "Error: " << error.c_str() << std::endl;
105       }
106     }
107
108
109
110     /**
111      * log_position logs the specified position if position
112      * logging is enabled.
113      * @param x - The x coordinat to log.
114      * @param y - The y coordinat to log.
115      * @param z - The z coordinat to log.
116      */
117     void log_position(float x, float y, float z) {
118
119       if (m_log_positions && m_matlab_mode && m_logging_enabled) {
120         m_target_file << "[" << x << ";" << y  << ";" << z << "],";
121       } else if (m_log_positions && m_logging_enabled) {
122         m_target_file << "Position: (" << x << "," << y << "," << z
123                       << ")" << std::endl;
124       }
125     }
126
127   };
128 } // namespace WiiIRLib
129 // WIIIRLIB_LOGGER_H
130 #endif
```

C.3 Wiimote.h

```
1  #ifndef WIIIRLIB_WIIMOTE_H
2  #define WIIIRLIB_WIIMOTE_H
3
4  #define M_PI       3.14159265358979323846f
5
6  #include <cmath>
7  #include "HIDBridge.h"
8  #include "Logger.h"
9
10 /**
11  * Point represents a detected IR point by the Wiimote.
12  */
13 typedef struct point {
14   float x, y, z, size;
15   void init(float x_ = 1, float y_ = 1, float z_ = 0, float size_ = 15) {
16     x = x_;
17     y = y_;
18     size = size_;
19     z = z_;
20   }
21 } point;
22
23 namespace WiiIRLib {
24
25   /**
26    * Wiimote is a class handeling all communication with a Wiimote.
27    * It is dependent on HIDBridge to perform low level communication.
28    */
29   class Wiimote {
30
31   private:
32
33     // Define button values
34     static const unsigned short BUTTON_All = 0x9F1F;
35
36     // Define led and vibration values
37     static const unsigned char OUTPUT_LED_1      = 0x10;
38     static const unsigned char OUTPUT_LED_2      = 0x20;
39     static const unsigned char OUTPUT_LED_3      = 0x40;
40     static const unsigned char OUTPUT_LED_4      = 0x80;
41     static const unsigned char OUTPUT_LED_ALL    = 0xF0;
42     static const unsigned char OUTPUT_VIBRATION = 0x01;
43     static const unsigned char OUTPUT_ALL        = 0xF1;
44     static const unsigned char OUTPUT_NONE       = 0x00;
45
46     // Define output channels
47     static const unsigned char OUTPUT_CHANNEL_FORCE_FEEDBACK = 0x13;
48     static const unsigned char OUTPUT_CHANNEL_LED = 0x11;
49     static const unsigned char OUTPUT_CHANNEL_REPORT = 0x12;
50     static const unsigned char OUTPUT_READ_MEMORY = 0x17;
51     static const unsigned char OUTPUT_WRITE_MEMORY = 0x16;
52     static const unsigned char OUTPUT_ENABLE_IR = 0x13;
53     static const unsigned char OUTPUT_ENABLE_IR2 = 0x1a;
54
55     // Define report request types
56     static const unsigned char REQUEST_CONTINUOUS_REPORTS = 0x04;
57     static const unsigned char REQUEST_SINGLE_REPORTS = 0x00;
58
59     // Define input channels
60     static const unsigned char INPUT_CHANNEL_BUTTONS_ONLY = 0x30;
61     static const unsigned char INPUT_CHANNEL_MOTION_IR = 0x33;
62
```

```
63      // Define IR related constants
64      static const unsigned long IR_REG_1 = 0x04b00030;
65      static const unsigned long IR_REG_2 = 0x04b00033;
66      static const unsigned long IR_SENS_ADDR_1 = 0x04b00000;
67      static const unsigned long IR_SENS_ADDR_2 = 0x04b0001a;
68
69      static const unsigned char IR_MODE_OFF = 0;
70      static const unsigned char IR_MODE_STD = 1;
71      static const unsigned char IR_MODE_EXP = 3;
72      static const unsigned char IR_MODE_FULL = 5;
73
74      static const int m_output_buffer_size = 22;
75      unsigned char m_output_buffer[m_output_buffer_size];
76
77      static const int m_input_buffer_size = 22;
78      unsigned char m_input_buffer[m_input_buffer_size];
79
80      point point1, point2, eye_position, old_eye_position;
81      point target_vector, up_vector;
82
83      float m_left, m_right, m_bottom, m_top;
84
85      float dot_distance_in_mm;
86      float screen_height_in_mm;
87      float radians_per_pixel;
88
89      float camera_verticale_angle;
90      float relative_vertical_angle;
91
92      float m_near_plane;
93      float m_far_plane;
94
95      float screenAspect;
96
97      WiiIRLib::HIDBridge hid_device;
98      WiiIRLib::Logger m_logger;
99
100   private:
101
102     /**
103      * empty_output_buffer clears the output and input buffers.
104      */
105     void empty_output_buffer() {
106
107       memset(m_output_buffer,0,m_output_buffer_size);
108       memset(m_input_buffer,0,m_input_buffer_size);
109     }
110
111
112
113     /**
114      * calculate_eye_point calculates the position of the user's eye
115      * in world coordinates.
116      */
117     void calculate_eye_position() {
118
119       eye_position.x = (point1.x+point2.x)*0.5f;
120       eye_position.y = (point1.y+point2.y)*0.5f;
121       eye_position.z = sqrt(pow(point1.x-point2.x,2)+
122                             pow(point1.y-point2.y,2));
123     }
124
```

```
125
126   public:
127
128     Wiimote() {
129
130       empty_output_buffer();
131
132       dot_distance_in_mm = 30.0f;
133       //dot_distance_in_mm = 5 ;
134       screen_height_in_mm = 192.0;
135       radians_per_pixel = M_PI / 4.0f / 1024.0f;
136
137       camera_verticale_angle = 0;
138       relative_vertical_angle = 0;
139
140       m_near_plane = 0.05f;
141       m_far_plane  = 100;
142
143       screenAspect = 1;
144     };
145
146
147
148     ~Wiimote() {}
149
150
151
152     /**
153      * toggle_logging enables or disables logging to a file.
154      * @return - True if logging is enabled, false otherwise.
155      */
156     bool toggle_logging() {
157
158       return m_logger.toggle_logging();
159     }
160
161
162
163     /**
164      * connect opens a connection to the Wiimote and turns on LED 1
165      * as the only light if successful.
166      * @return - True if successful, false otherwise.
167      */
168     bool connect() {
169
170       bool success = hid_device.connect();
171       if (success) {
172         set_led_status(OUTPUT_LED_1);
173         m_logger.log_message("Connection_estabilished");
174       }
175       return success;
176     }
177
178
179
180     /**
181      * Disconnect closes the connection to the Wiimote and turns on
182      * LED 3 as the only light.
183      * @return - True if successful, false otherwise.
184      */
185     bool disconnect() {
186
```

```
187         set_led_status(OUTPUT_LED_3);
188         Sleep(10);
189         return hid_device.disconnect();
190       }
191
192
193
194       /**
195        * set_led_status turns on/off the specified LEDs.
196        * @param leds - A bitmask specifying which LEDs that should
197        * be turned on/off.
198        * @return - True if the LEDs were set, false otherwise.
199        */
200       bool set_led_status(unsigned char leds) {
201
202         Sleep(10);
203         empty_output_buffer();
204         m_output_buffer[0] = OUTPUT_CHANNEL_LED;
205         m_output_buffer[1] = leds;
206         bool status = hid_device.write(m_output_buffer,m_output_buffer_size);
207         return status;
208       }
209
210
211       /**
212        * is_connected returns whether a Wiimote is connected.
213        * @return - True if connected, false otherwise.
214        */
215       bool is_connected() {
216
217         return hid_device.is_connected();
218       }
219
220
221       /**
222        * enable_ir sets the reporting mode to continuous, enables IR 1 and 2
223        * and sets the data mode to extended. If successful, LEDs 1 and 2
224        * are turned on.
225        * @return - True if IR was enabled, false otherwise.
226        */
227       bool enable_ir() {
228
229         const unsigned char IR_SENS_MIDRANGE_PART1[] =
230                       {0x02, 0x00, 0x00, 0x71, 0x01, 0x00, 0xaa, 0x00, 0x64};
231         const unsigned char IR_SENS_MIDRANGE_PART2[] = {0x63, 0x03};
232
233         bool success = false;
234
235         point1.init();
236         point2.init();
237         eye_position.init();
238         target_vector.init(0.5,0.5,0.0);
239         up_vector.init(0.0,1.0,0.0);
240         old_eye_position.init();
241
242         // Set continuous reporting mode.
243         empty_output_buffer();
244         m_output_buffer[0] = OUTPUT_CHANNEL_REPORT;
245         m_output_buffer[1] = REQUEST_CONTINUOUS_REPORTS;
246         m_output_buffer[2] = INPUT_CHANNEL_MOTION_IR;
247         success = hid_device.write(m_output_buffer,m_output_buffer_size);
248
```

```
249         Sleep(10);
250
251         // Enable IR 1.
252         empty_output_buffer();
253         m_output_buffer[0] = OUTPUT_ENABLE_IR;
254         m_output_buffer[1] = REQUEST_CONTINUOUS_REPORTS;
255         success = success &&
256                 hid_device.write(m_output_buffer, m_output_buffer_size);
257
258         Sleep(10);
259
260         // Enable IR 2.
261         empty_output_buffer();
262         m_output_buffer[0] = OUTPUT_ENABLE_IR2;
263         m_output_buffer[1] = REQUEST_CONTINUOUS_REPORTS;
264         success = success &&
265                 hid_device.write(m_output_buffer, m_output_buffer_size);
266
267         Sleep(10);
268
269         // Write sensitivity data to the Wiimote and set IR mode to Extended.
270         if (success) {
271           unsigned char val = 0x1;
272           success = write_memory(IR_REG_1,1,&val);
273           Sleep(10);
274         }
275         if (success)  {
276           success = write_memory(IR_SENS_ADDR_1,9,IR_SENS_MIDRANGE_PART1);
277           Sleep(10);
278         }
279         if (success) {
280           success = write_memory(IR_SENS_ADDR_2,2,IR_SENS_MIDRANGE_PART2);
281           Sleep(10);
282         }
283         if (success)  {
284           success = write_memory(IR_REG_2,1,&IR_MODE_EXP);
285           Sleep(10);
286         }
287         if (success)  {
288           unsigned char val = 0x8;
289           success = write_memory(IR_REG_1,1,&val);
290           Sleep(10);
291         }
292
293         // Turn on LEDs 1 and 2.
294         if (success) {
295           set_led_status(OUTPUT_LED_1 | OUTPUT_LED_2);
296           m_logger.log_message("IR_enabled");
297         }
298         return success;
299       }
300
301
302
303       /**
304        * disable_ir disables both IR1 and 2 and sets the reporting mode
305        * to buttons only mode to avoid continuous reporting. If successful
306        * LED 1 is turned on.
307        * @return - True if successful, false otherwise.
308        */
309       bool disable_ir() {
310
```

```
311         bool success = false;
312
313         // Set reporting mode to on button press only.
314         empty_output_buffer();
315         m_output_buffer[0] = OUTPUT_CHANNEL_REPORT;
316         m_output_buffer[1] = REQUEST_SINGLE_REPORTS;
317         m_output_buffer[2] = INPUT_CHANNEL_BUTTONS_ONLY;
318         success = hid_device.write(m_output_buffer,m_output_buffer_size);
319
320         Sleep(10);
321
322         // Disable IR 1.
323         empty_output_buffer();
324         m_output_buffer[0] = OUTPUT_ENABLE_IR;
325         m_output_buffer[1] = 0x00;
326         success = hid_device.write(m_output_buffer,m_output_buffer_size);
327
328         Sleep(10);
329
330         // Disable IR 2.
331         empty_output_buffer();
332         m_output_buffer[0] = OUTPUT_ENABLE_IR2;
333         m_output_buffer[1] = 0x00;
334         success = success &&
335               hid_device.write(m_output_buffer,m_output_buffer_size);
336
337         Sleep(10);
338
339         // Turn on LED 1.
340         if (success) {
341           set_led_status(OUTPUT_LED_1);
342           m_logger.log_message("IR_disabled");
343         }
344         return success;
345       }
346
347
348
349       /**
350        * retrieve_data is a heart beat function that retrieves the latest
351        * data from the Wiimote and stores it. The user's position in the
352        * world and the boundaries of the view frustum are also calculated.
353        * To reduce high frequency noise, the value is also smoothed.
354        * @param timeout - The timeout for the function.
355        * @return - True if the data was retrieved successful, false otherwise.
356        */
357       bool retrieve_data(int timeout = 1) {
358
359         // Sleep 1 ms to avoid too frequent reads.
360         Sleep(1);
361
362         bool success = false;
363         int bytes_read = 0;
364
365         // Read the data.
366         success = hid_device.read(m_input_buffer,
367                             m_input_buffer_size,bytes_read,timeout);
368
369         if (success && bytes_read > 0) {
370           if (m_input_buffer[0] == INPUT_CHANNEL_MOTION_IR) {
371             // The package is of the expected format, containing motion
372             // and IR data. Shifting is due to the extended IR mode. See
```

```
373             // http://wiibrew.org/index.php?title=Wiimote#IR_Camera
374             // for details.
375
376
377             // Get point position and size. Then calculate the users position.
378             point1.x = (float)(m_input_buffer[6+0] |
379                                (m_input_buffer[6+2]&0x30) << 4);
380             point1.y = (float)(m_input_buffer[6+1] |
381                                (m_input_buffer[6+2]&0xC0) << 2);
382             point2.x = (float)((m_input_buffer[6+3] |
383                                (m_input_buffer[6+5]&0x30) << 4)+0.01);
384             point2.y = (float)(m_input_buffer[6+4] |
385                                (m_input_buffer[6+5]&0xC0) << 2);
386             point1.size = (float)(m_input_buffer[6+2] & 0xF);
387             point2.size = (float)(m_input_buffer[6+5] & 0xF);
388
389
390             // Only calculate a new head position if the points are detected.
391             if (point1.y < 769.0f && point2.y < 769.0f) {
392
393               point1.x = 1023 - point1.x;
394               point2.x = 1023 - point2.x;
395
396               // Calculate initial estimate of eye position.
397               calculate_eye_position();
398
399               m_logger.log_position(eye_position.x/1023.0f,
400                                     eye_position.y/767.0f, eye_position.z);
401
402               float offset = ((eye_position.x-512.0f)/1023.0f)*0.0776f;
403               float angle = radians_per_pixel * eye_position.z / 2.0f;
404
405               // Calculate the eye position, considering perspective,
406               // screen size and LED distance. Apporach based on the
407               // solution of Johnny Lee.
408               eye_position.z = dot_distance_in_mm / 2.0f /
409                                        tan(angle) / screen_height_in_mm;
410               eye_position.x = sin(radians_per_pixel * (eye_position.x - 512))
411                                * eye_position.z;
412               relative_vertical_angle = (eye_position.y - 384) *
413                                                    radians_per_pixel;
414               eye_position.y =
415                     sin(relative_vertical_angle + camera_verticale_angle)
416                     * eye_position.z;
417
418               // Weight the previous calculated position by 0.3 to smooth
419               // the movement of reduce the high frequency noise.
420               eye_position.x = eye_position.x*0.7f+old_eye_position.x*0.3f;
421               eye_position.y = eye_position.y*0.7f+old_eye_position.y*0.3f;
422               eye_position.z = eye_position.z*0.7f+old_eye_position.z*0.3f;
423
424               old_eye_position.x = eye_position.x;
425               old_eye_position.y = eye_position.y;
426               old_eye_position.z = eye_position.z;
427
428               // Calculate the boundaries for the view frustum.
429               // Apporach based on the solution of Johnny Lee.
430               m_left = m_near_plane*(-0.5f * screenAspect + eye_position.x)/
431                        eye_position.z - offset;
432               m_right = m_near_plane*( 0.5f * screenAspect + eye_position.x)/
433                         eye_position.z - offset;
434               m_bottom = m_near_plane*(-0.5f - eye_position.y)/eye_position.z;
```

```
435             m_top = m_near_plane*( 0.5f - eye_position.y)/eye_position.z;
436           }
437         } else {
438           m_logger.log_error("Unexpected_package");
439         }
440       }
441     return success;
442     }
443
444
445
446     /**
447      * init_logger sets up the logger. As default only positions are logged.
448      * @param file - The path and file name of the target file.
449      * @param matlab_mode - True if positions should be logged in Matlab
450      * format for graph drawing. If enabled, only positions will be logged.
451      * @param messages - True if messages should be logged, false otherwise.
452      * @param errors - True if errors should be logged, false otherwise.
453      * @param positions - True if positions should be logged,
454      * false otherwise.
455      */
456     void init_logger(std::string file, bool matlab_mode,
457         bool messages = false, bool errors = false, bool positions = true) {
458
459       m_logger.init(file,matlab_mode,messages,errors,positions);
460     }
461
462
463
464     /**
465      * get_point1 returns the first point.
466      * @return - The latest point.
467      */
468     const point get_point1() {
469
470       return point1;
471     }
472
473
474
475     /**
476      * get_point2 returns the second point.
477      * @return - The latest point.
478      */
479     const point get_point2() {
480
481       return point2;
482     }
483
484
485
486     /**
487      * get_eye_position returns the position of the user's head.
488      * @return - The latest position.
489      */
490     const point get_eye_position() {
491
492       return eye_position;
493     }
494
495
496
```

```
497     /**
498      * get_target returns the target of the camera.
499      * @return - The target of the camera.
500      */
501     const point get_target() {
502
503       return target_vector;
504     }
505
506
507
508     /**
509      * get_up returns the up vector in the world.
510      * @return - The up vector.
511      */
512     const point get_up() {
513
514       return up_vector;
515     }
516
517
518
519     /**
520      * get_left returns the left boundary of the view.
521      * @return - The left boundary.
522      */
523     const float get_left() {
524
525       return m_left;
526     }
527
528
529
530     /**
531      * get_right returns the right boundary of the view.
532      * @return - The right boundary.
533      */
534     const float get_right() {
535
536       return m_right;
537     }
538
539
540
541     /**
542      * get_bottom returns the bottom boundary of the view.
543      * @return - The bottom boundary.
544      */
545     const float get_bottom() {
546
547       return m_bottom;
548     }
549
550
551
552     /**
553      * get_top returns the top boundary of the view.
554      * @return - The top boundary.
555      */
556     const float get_top() {
557
558       return m_top;
```

```
559     }
560
561
562     /**
563      * get_near_plane returns the near clipping plane.
564      * @return - The near clipping plane.
565      */
566     const float get_near_plane() {
567
568       return m_near_plane;
569     }
570
571
572     /**
573      * get_far_plane returns the far clipping plane.
574      * @return - The far clipping plane.
575      */
576     const float get_far_plane() {
577
578       return m_far_plane;
579     }
580
581
582
583     /**
584      * write_memory writes up to 16 bytes of data to the specified memory
585      * area in the Wiimote. Function is based on the corresponding function
586      * in cWiiMote 0.3, available at http://www.ecando.it/cwmdown.html.
587      * @param address - The address to write to.
588      * @param size - The size of the data to be written in bytes.
589      * @param buffer - The data to be written.
590      * @return - True if all data was written, false otherwise.
591      */
592     bool write_memory(unsigned int address, unsigned char size,
593                                             const unsigned char * buffer) {
594
595       bool retval = false;
596       if (size <= 16)   {
597         empty_output_buffer();
598         m_output_buffer[0] = OUTPUT_WRITE_MEMORY;
599         m_output_buffer[1] = (address & 0xff000000) >> 24;
600         m_output_buffer[2] = (address & 0x00ff0000) >> 16;
601         m_output_buffer[3] = (address & 0x0000ff00) >> 8;
602         m_output_buffer[4] = (address & 0xff);
603         m_output_buffer[5] = size;
604         memcpy(&m_output_buffer[6], buffer, size);
605         retval = hid_device.write(m_output_buffer, m_output_buffer_size);
606       }
607       return retval;
608     }
609
610
611
612     /**
613      * print prints out the position and size of the first two points.
614      */
615     void print() {
616
617       std::cout << "(x1,y1,s1): " << point1.x << ", " <<
618                                 point1.y << ", " << point1.size << std::endl;
619       std::cout << "(x2,y2,s2): " << point2.x << ", " <<
620                                 point2.y << ", " << point2.size << std::endl;
```

```
621       }
622
623     };
624   } // WiiIRLib namespace
625   #endif //WIIIRLIB_WIIMOTE_H
```

C.4 WiiIRLib.cpp

```
1   #include <iostream>
2   #include "Wiimote.h"
3   #include "GL/glew.h"
4   #include "GL/glut.h"
5
6   /**
7    * TargetRoom renders a cube containing a number of targets using OpenGL.
8    * The sensor bar and Wiimote can then be used to move forward and backward
9    * as well as look around in the room.
10   */
11  WiiIRLib::Wiimote *m_wiimote = NULL;
12  GLfloat cubedepth = 3.0;
13
14  // Define the cube
15  const int no_vertices = 5;
16  GLfloat left[no_vertices*no_vertices*3];
17  GLfloat right[no_vertices*no_vertices*3];
18  GLfloat top[no_vertices*no_vertices*3];
19  GLfloat bottom[no_vertices*no_vertices*3];
20  GLfloat back[no_vertices*no_vertices*3];
21  GLint cube_indices[] = {0,1,2,3,4,
22                          9,8,7,6,5,
23                          10,11,12,13,14,
24                          19,18,17,16,15,
25                          20,21,22,23,24,
26                          19,14,9,4,3,
27                          8,13,18,23,22,
28                          17,12,7,2,1,
29                          6,11,16,21,20,
30                          15,10,5,0};
31
32
33
34  /**
35   * place_target places a target in the cube.
36   * @param x - The x-coordinate of the target.
37   * @param y - The y-coordinate of the target.
38   * @param z - The z-coordinate of the target.
39   */
40  void place_target(float x, float y, float z) {
41
42    float size = 0.06;
43    float hidden_size = size*0.6;
44    float hidden_depth = 0.1;
45    // Define the targets and some "hidden" targets behind the main targets.
46    GLfloat target[] = {x-size, y+size, z,
47                        x+size, y+size, z,
48                        x+size, y-size, z,
49                        x-size, y-size, z,
50                        x, y, z,
```

```
51                          x, y, -cubedepth,
52                          x-size*0.5, y+size*0.5, z,
53                          x+size*0.5, y+size*0.5, z,
54                          x+size*0.5, y-size*0.5, z,
55                          x-size*0.5, y-size*0.5, z,
56
57                          x-hidden_size, y+hidden_size, z-hidden_depth,
58                          x+hidden_size, y+hidden_size, z-hidden_depth,
59                          x+hidden_size, y-hidden_size, z-hidden_depth,
60                          x-hidden_size, y-hidden_size, z-hidden_depth};
61
62    GLint indices[] = {0, 1, 2, 3};
63    GLint line[] = {4,5};
64    GLint box[] = {6, 7, 8, 9};
65    GLint hidden[] = {10, 11, 12, 13};
66
67    glVertexPointer(3, GL_FLOAT, 0, target);
68
69    // Draw the targets
70    glColor3f(1.0,1.0,1.0);
71    glDrawElements(GL_POLYGON,4,GL_UNSIGNED_INT,box);
72
73    glColor3f(1.0,0.0,0.0);
74    glDrawElements(GL_LINES,2,GL_UNSIGNED_INT,line);
75    glDrawElements(GL_POLYGON,4,GL_UNSIGNED_INT,indices);
76
77    glColor3f(0.0,0.0,1.0);
78    glDrawElements(GL_POLYGON,4,GL_UNSIGNED_INT,hidden);
79  }
80
81
82
83  /**
84   * init sets up the OpenGL states and calculates the positions of the
85   * vertices that makes up the cube.
86   */
87  void init(void) {
88
89    // Set up OpenGL states
90    glClearColor(0.0, 0.0, 0.0, 0.0);
91    glShadeModel(GL_FLAT);
92    glEnable(GL_DEPTH_TEST);
93    glEnable(GL_LINE_SMOOTH);
94    glEnable(GL_BLEND);
95    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
96    glHint(GL_LINE_SMOOTH_HINT, GL_DONT_CARE);
97    glLineWidth(3.5);
98
99    // Calculate the vertices of the cube.
100   float a, b;
101   for (int i = 0; i < no_vertices; ++i) {
102     for (int j = 0; j < no_vertices; ++j) {
103       a = j/(float)(no_vertices-1)-0.5;
104       b = -i/(float)(no_vertices-1)*cubedepth;
105
106       left[(i*no_vertices+j)*3] = -0.5;
107       left[(i*no_vertices+j)*3+1] = a;
108       left[(i*no_vertices+j)*3+2] = b+0.1;
109
110       right[(i*no_vertices+j)*3] = 0.5;
111       right[(i*no_vertices+j)*3+1] = a;
112       right[(i*no_vertices+j)*3+2] = b+0.1;
```

```
113
114         top[(i*no_vertices+j)*3] = a;
115         top[(i*no_vertices+j)*3+1] = 0.5;
116         top[(i*no_vertices+j)*3+2] = b+0.1;
117
118         bottom[(i*no_vertices+j)*3] = a;
119         bottom[(i*no_vertices+j)*3+1] = -0.5;
120         bottom[(i*no_vertices+j)*3+2] = b+0.1;
121
122         back[(i*no_vertices+j)*3] = a;
123         back[(i*no_vertices+j)*3+1] = i/(float)(no_vertices-1)-0.5;
124         back[(i*no_vertices+j)*3+2] = -cubedepth;
125       }
126     }
127 }
128
129
130
131 /**
132  * display renders the cube and targets.
133  */
134 void display(void) {
135
136   glClear(GL_COLOR_BUFFER_BIT);
137   glColor3f(1.0, 1.0, 1.0);
138   glLoadIdentity();
139   point p = m_wiimote->get_eye_position();
140
141   // Set up the view matrix.
142   gluLookAt(p.x, p.y, p.z,
143             p.x, p.y, 0.0,
144             0.0, 1.0, 0.0);
145
146   // Draw the cube.
147   glColor3f(1.0,1.0,1.0);
148   glEnableClientState(GL_VERTEX_ARRAY);
149
150   glVertexPointer(3, GL_FLOAT, 0, back);
151   glDrawElements(GL_LINE_LOOP,45,GL_UNSIGNED_INT,cube_indices);
152
153   glVertexPointer(3, GL_FLOAT, 0, left);
154   glDrawElements(GL_LINE_LOOP,45,GL_UNSIGNED_INT,cube_indices);
155
156   glVertexPointer(3, GL_FLOAT, 0, right);
157   glDrawElements(GL_LINE_LOOP,45,GL_UNSIGNED_INT,cube_indices);
158
159   glVertexPointer(3, GL_FLOAT, 0, top);
160   glDrawElements(GL_LINE_LOOP,45,GL_UNSIGNED_INT,cube_indices);
161
162   glColor3f(0.0,1.0,0.0);
163   glVertexPointer(3, GL_FLOAT, 0, bottom);
164   glDrawElements(GL_LINE_LOOP,45,GL_UNSIGNED_INT,cube_indices);
165
166   // Draw the targets.
167   place_target(0.0,0.0,-1.0);
168   place_target(0.2,-0.1,0.0);
169   place_target(-0.3,0.3,0.1);
170   place_target(0.1,0.4,-1.5);
171   place_target(-0.2,0.0,0.2);
172   place_target(-0.25,-0.35,-1.4);
173   place_target(0.3,-0.35,0.3);
174
```

```
175     glFlush();
176     glClear(GL_DEPTH_BUFFER_BIT);
177   }
178
179
180
181   /**
182    * reshape sets up the projection matrix and viewport.
183    * @param w - The width of the window.
184    * @param h - The height of the window.
185    */
186   void reshape (int w, int h) {
187
188     // Set up the viewport.
189     glViewport(0, 0, (GLsizei) w, (GLsizei) h);
190
191     // Set up the projection matrix.
192     glMatrixMode(GL_PROJECTION);
193     glLoadIdentity();
194     glFrustum (m_wiimote->get_left(), m_wiimote->get_right(),
195                m_wiimote->get_bottom(), m_wiimote->get_top(),
196                m_wiimote->get_near_plane(), m_wiimote->get_far_plane());
197     glMatrixMode (GL_MODELVIEW);
198   }
199
200
201
202   /**
203    * idle is the idle function to perform when there is nothing to do. It
204    * updates the view frustum and reads the newest data from the Wiimote.
205    */
206   void idle() {
207
208     // Set up the projection matrix.
209     glMatrixMode(GL_PROJECTION);
210     glLoadIdentity();
211     glFrustum (m_wiimote->get_left(), m_wiimote->get_right(),
212                m_wiimote->get_bottom(), m_wiimote->get_top(),
213                m_wiimote->get_near_plane(), m_wiimote->get_far_plane());
214     glMatrixMode (GL_MODELVIEW);
215
216     // Get the newest data from the Wiimote.
217     m_wiimote->retrieve_data();
218     glutPostRedisplay();
219   }
220
221
222
223   /**
224    * keyboard detects inputs and allows one to terminate the program
225    * by pressing the escape key.
226    * @param key - The keyboard key.
227    * @param x - The x position of the mouse.
228    * @param x - The y position of the mouse.
229    */
230   void keyboard(unsigned char key, int x, int y) {
231
232     if (key == 27) {
233       // The pressed key is the escape key.
234       std::cout << "IR_activated:_" << !m_wiimote->disable_ir() << std::endl;
235       Sleep(1000);
236       m_wiimote->disconnect();
```

```
237       exit(0);
238     }
239   }
240
241
242
243   /**
244    * main estabilishes the connection to the Wiimote, initializes the
245    * OpenGL window and starts the rendering.
246    * @param argc - The number of arguments.
247    * @param argv - The arguments.
248    */
249   int main(int argc, char** argv) {
250
251     // Connect to the Wiimote
252     m_wiimote = new WiiIRLib::Wiimote();
253     std::cout << "Is connected: " << m_wiimote->is_connected() << std::endl;
254     m_wiimote->connect();
255     std::cout << "Is connected: " << m_wiimote->is_connected() << std::endl;
256     Sleep(1000);
257     std::cout << "IR activated: " << m_wiimote->enable_ir() << std::endl;
258
259     m_wiimote->retrieve_data();
260
261     // Set up the OpenGL window using GLUT.
262     glutInit(&argc, argv);
263     glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
264     glutInitWindowSize (500, 500);
265     glutInitWindowPosition (100, 100);
266     glutCreateWindow (argv[0]);
267     init ();
268
269     // Set up the display, reshape, idle and keyboard function.
270     glutDisplayFunc(display);
271     glutReshapeFunc(reshape);
272     glutIdleFunc(idle);
273     glutKeyboardFunc(keyboard);
274     glutMainLoop();
275     return 0;
276   }
```