UNIVERSITY OF TROMSØ UIT

FACULTY OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

Inexpensive Head Tracking for use with Large High-Resolution Displays

Joakim Simonsson

INF-3981
Master's Thesis in Computer Science
June, 2010

Abstract

This thesis investigates how head tracking can be implemented by using inexpensive off-the-shelf hardware for a 6 × 3 meter high-resolution display wall. The tracking system has been integrated into to an existing event system, Shout, that allows for inter-program communication.

An application called HTSIM has been developed that is used for testing different head tracking configurations in a virtual environment. Developing in a virtual environment does not require access to head tracking hardware. Tracking algorithms developed in the virtual environment can directly be used for head tracking in a physical environment. The tracking system is able to track a user's head with cameras that are placed behind the user.

HTSIM is also used for configuring the head tracking system used in the physical environmet.

Experiments detail out the overall latency in the system and sources of jitter.

Acknowledgements

I would like to thank my supervisors, John Markus Bjørndalen and Daniel Stødle, for guiding me through this project and giving me valuable tips.

A also want to thank Ken-Arne Jensen for constructing much of this project's hardware. This project would have been difficult to complete without his help.

A thank you goes to Tor-Magne Stien Hagen that has contributed with helpful tips.

Contents

List of Figures

Nomenclature

2D	Two-dimensional
3D	Three-dimensional
API	Application Programming Interface
CCD	Charge-coupled device
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DoF	Degrees of Freedom
GB	Gigabyte (1024^3 bytes)
GHz	Gigahertz
IDE	Integrated Development Environment
IR	Infrared
kbps	Kilobit per second
LED	Light Emitting Diode
MAC	Media Access Control
MB	Megabyte (1024^2 bytes)
ms	Millisecond
mW	Milliwatt
nm	Nanometer
RAM	Random Access Memory
SDK	Software Development Kit
USB	Universal Serial Bus
VR	Virtual Reality

Chapter 1

Introduction

One of the important factors in Virtual Reality, VR, is to present computer graphics based on the position of the observer. For desktop VR, it is often assumed that the position of the observer is fixed. This assumption is broken when a user moves in front of a display, resulting in a degraded perception of reality.

To improve the illusion that the user looks into a 3D world, head tracking can be used, informing the system about the position and orientation of the user's head. The infrared (IR) camera in a Nintendo Wii Controller (Wiimote) can be used to accomplish low cost head tracking. This has successfully been done before for TV displays. [8] In that setup, a single Wiimote, directed at the user, is positioned adjacent to a TV set. The user, facing the TV set, wears glasses equipped with two IR emitters placed on each side of the glasses. With this approach, the system is able track the position of the user's head. The IR camera registers side-to-side- and up-and-down-movement of the user. An estimated distance between the TV-set and the user is also calculated.

The display wall at Tromsø Display Wall laboratory uses 28 projectors to create one 6 x 3 meter back projected image. The high resolution image makes it possible for users to move very close up to the display wall. The head tracking implementation in this situation has to be solved in a different way than in [8], since there is no room for positioning a Wiimote in front of the user.

1.1 3D Tracking

By using two or more Wiimotes, it is possible calculate the 3D position of an IR source. [6] and [17] describe how 3D tracking can be accomplished by using two Wiimotes. Here, triangulation is used to find the position of an IR source. Contrary to [8], where only an estimated 3D position is obtained, the triangulation technique makes it possible to accurately calculate the 3D positions of IR sources. Tests in [6] and [17] show that the precision lies in the millimeter range.

1.2 Determining Eye Positions

The term head tracking implies that the position and possibly the orientation of a head is being tracked. To be able to describe the position of a head as a 3D point in space, it has to be decided where on the head this point is located. Defining such a point relative to the head requires a definition of the head and its bounds. For example, if the point that defines the head position is located in the center of the head, the bounds of the head need to be known for calculating the center. A bounds definition must be based on the anatomy of the human body. A consequence is that the anatomy of each user needs to be analyzed, i.e. the user's head needs to be measured based on a head definition.

Fortunately, the purpose of head tracking systems that aim to present a user dependent view is to track the position of the eyes and not the head itself. Since the environment around an observer is interpreted based on where the observer's eyes are located, there is no need for creating a precise definition of the head position.

In optical head tracking systems, markers are typically used. The markers in systems using IR cameras are either IR light sources or an object with a material that reflects IR light. The markers are attached relative to the user's head. The only positional information an optical tracking system has is the position of the markers. If two markers are positioned on each side of an eye, the position centered between these two markers will define the position of that eye. To be able to calculate an eye position when markers are placed on a different part of the head, e.g. the forehead or the back of the head, offsets from the markers to the eyes have to be known. An offset is a 3D vector between the marker and the eye position. The system needs to know the orientation of the head to be able to know what way the offset vector is pointing to.

Since the display used in this project does not use any stereoscopic techniques[1], it is not relevant to obtain both eye positions of the user. Throughout this text, the term eye position is defined as the location between both eyes of the user, see Figure 1.1.

The head tracking system provided by [11] uses multiple IR sources attached to the user's head. Head tracking implementations such as, [8], [11], [15], and [2] implement head tracking where the camera is placed in front of the user.

1.3 Virtual Head Tracking Environment

The system depends on three components: (i) IR-hardware; (ii) the display wall; and (iii) and the room itself. During the development phase, testing the system implementation requires the developer to have access to the IR-hardware, the display wall and to be physically present in the room. In addition, the developer has to turn on IR hardware between each new build of the program. A person wearing the IR gear is also required when the tests are performed.

[1]Stereoscopic techniques present separate images for each eye, making it possible to create three-dimensional illusions on two-dimensional surfaces.

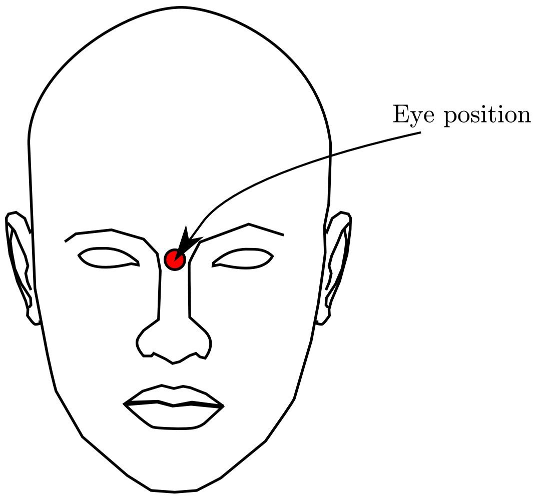


Figure 1.1: The term eye position refers to the location between the eyes.

Software bugs, causing erroneous system behavior, can be difficult to find and reproduce since the system depends on physical objects and their properties, e.g. camera and diode positions.

In this project, an application HTSIM was created that makes it possible to develop and test the system in a virtual environment. The virtual environment removes the some of the time consuming tasks that a physical environment introduces. Developing, testing, and debugging a system that provides states that are identical for each run is a facilitating property for a developer. In a virtual environment it is more convenient to try out different configurations of Wiimote placements. Positioning a Wiimote in reality requires a rig construction that supports the Wiimote. This can be avoided in the virtual environment, making the evaluation of arbitrary position configurations possible.

Algorithms developed in the virtual environment can be used unmodified in the physical implementation of the system.

1.4 Tromsø Display Wall

The Tromsø display wall consists of a 6×3 meter canvas. The canvas is back projected by 28 Dell 4100MP projectors. The projectors are positioned in a 7×4 grid. Each projector has a resolution of 1024×768 pixels. This creates a total resolution of 7168×3072 pixels and a size of 5.2×2.25 meters.

Every projector is connected to a computer that acts as a dedicated image generator. These 28 computers and their corresponding projector are referred to as *tiles* in the rest of this report. One additional computer acts as a front-end for the tiles. All these 29 computers are Dell Precision 370 workstations. They have each an Intel Pentium 4 EM64T CPU running at 3.2 GHz with hyper-threading and 2 GB of RAM. The video cards used are Nvidia Quadro FX 3400 (PCIe x16) cards with 256 MB video memory. The computers communicate

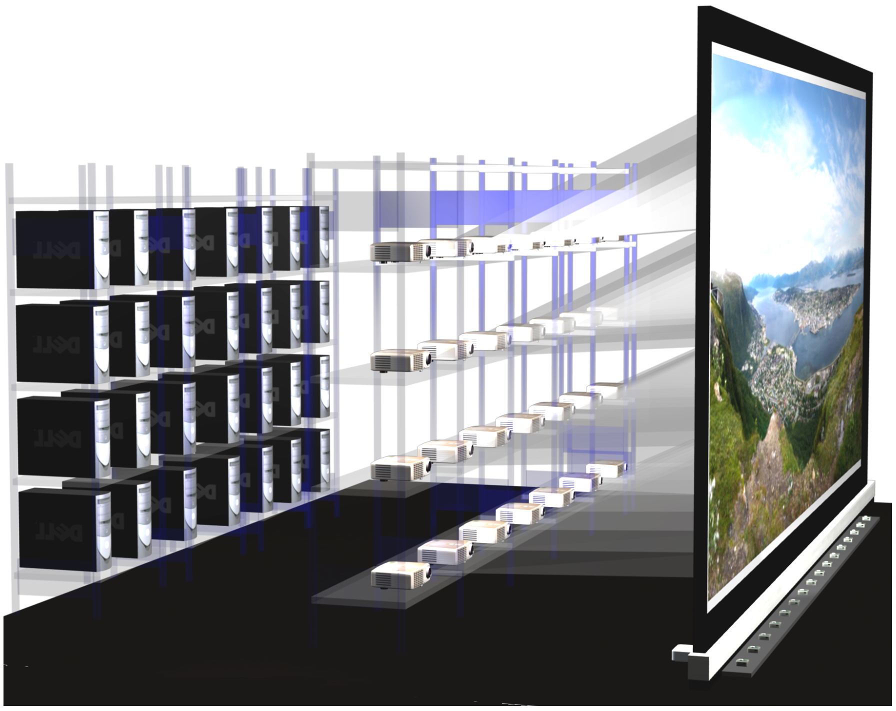


Figure 1.2: Illustration of the display wall (Courtesy of Tor-Magne Stien Hagen).

over a switched gigabit network[21].

Rocks[16], a Linux cluster distribution, is used by all computers. Version 4 of Rocks is the current version installed on the computers. In this version of Rocks, CentOS 4.2 is used as operating system.

16 cameras are located in front of the canvas. These cameras are part of a camera-sense system[21]. This system makes it possible to use the display wall as one large touch display. It is not required to touch the actual canvas. The system registers touch events when an object (in most cases a hand) is close to the canvas. The camera-sense system reports the touch events in pixels. The reported pixels lie within the range of the entire image (7168 $\times$ 3072). Since the size of the wall is known it is possible convert the pixel coordinates into other units, such as meters. An event system called Shout is used for handling passing these events between entities in the system. The Shout event system is described more in detail in Section 4.1.4.

Figure 1.2 illustrates the display wall setup. The 28 tiles are located leftmost in the figure. The 28 projectors, which back-project on the canvas, are located right of the tiles. The cameras are located on the floor, in front of the canvas. Figure 1.3 shows a user interacting with the display wall using the camera-sense system.

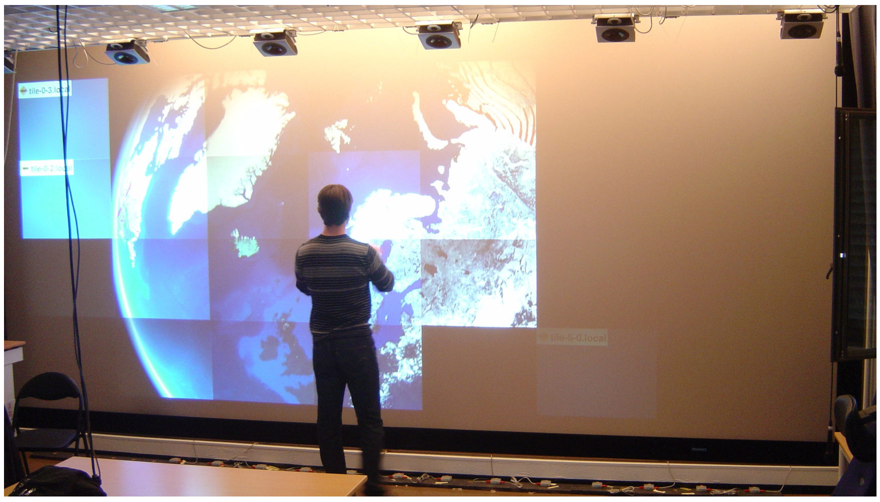


Figure 1.3: A user interacts with the display wall.

1.5 Hardware

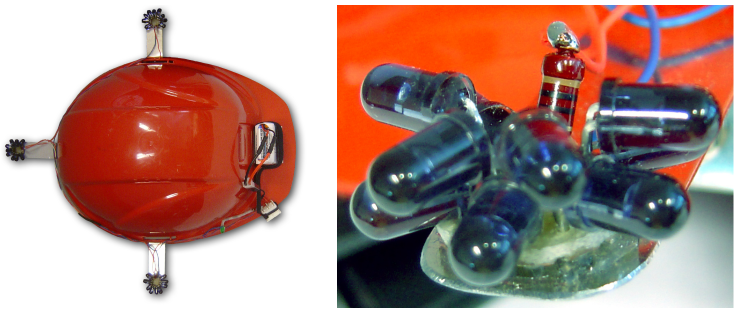
The hardware used in this project is partially off the shelf available and partially custom built. The bought hardware includes two Nintendo Wii Remote Controllers and a Bluetooth interface. All IR source constellations were constructed by Ken-Arne Jensen, senior engineer at the computer science department.

As specified in [17], [6], and [22], IR LEDs that have 940 nm wavelength are found to give good results with the Wiimote. [22] states that the Wiimote detects IR sources with 940 nm wavelength with the twice intensity than 850 nm sources. This project uses 950 nm wavelength IR LEDs. The intensity of these LEDs is 18 mW and their view angle is $\pm 25\,^{\circ}$.

1.5.1 Head Mounted IR sources

The IR LEDs are attached to a plastic helmet shown in Figure 1.4(a). Three clusters of infrared LEDs is placed on the helmet, one cluster on each side, and one in the back. Each cluster consists of nine LEDs as shown in 1.4(b). The way the clusters are constructed allows for a viewing angle of more than $180\,^{\circ}$. The helmet has a rechargeable battery pack that can be replaced easily. In addition there is an on/off switch, a power indicator LED and various resistors.

Alternative head worn devices, such as caps and sweat-bands, were considered. These devices tend to be of an elastic nature, which results in giving the LED clusters varying positions. The helmet assures that the LED clusters always have the same relative position.



(a) Top view. (b) IR LED Cluster.

Figure 1.4: Head mounted IR sources.

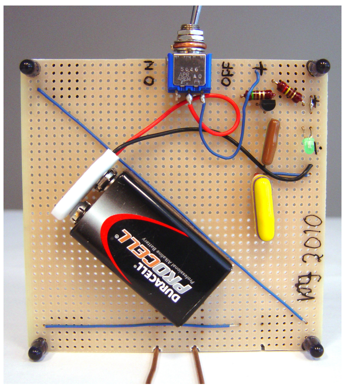


Figure 1.5: Calibration rig.

1.5.2 Calibration Rig

A rig with four infrared LEDs was built. The purpose of the rig is to calibrate cameras. The rig is powered by a nine-volt battery. It has an on/off switch and a power indicator. As seen in Figure 1.5, the LEDs are positioned at each corner of the rig. The spacing between two adjacent corners is 91.44 mm. Camera calibration is described in Section 3.3.

1.5.3 Nintendo Wii Remote Controller

The controllers used with the Nintendo Wii video game console are called Wiimotes. Various Internet communities have reversed engineered much of how the controller works. The WiiBrew Wiki [22] provides the very thorough information about the Wii console. Here, detailed specifications for the controller's different parts are freely available.

The Wiimotes use Bluetooth to communicate. In theory this makes it possible to make the controllers communicate with any Bluetooth interface. However, experiences such as in [2], show that some Bluetooth interfaces are incompatible with the Wiimote.

The Wiimote consists of several input and output devices. It has buttons, an infrared camera, and an accelerometer. A speaker, four LEDs, and a rumble motor, are its output devices.

According to [22], the camera of the Wiimote has a resolution of 128 x 96 pixels. It is a monochrome camera located behind an infrared pass filter. Instead of sending the entire camera image over the Bluetooth link, only the four brightest infrared sources are reported as x y coordinates. This saves bandwidth and enables higher framerates to be achieved. Based on the monochrome image, the Wiimote extracts the detected infrared sources on a subpixel level. The coordinates sent from the Wiimote therefore have a higher resolution than what the camera can provide. The coordinates are reported in the range [0-1023, 0-767]. The Wiimote also sends the approximate size of its detected IR sources. Sizes are reported either in the discrete range [0-15] or with bounding box coordinates. According to [8] and [2], the Wiimote transmits IR data at 100 Hz.

Several open source software libraries have been made based on the reversed engineered information. For example, [19], [4], and [23].

1.6 Alternatives

Implementing low cost head tracking can be achieved using other techniques than tracking infrared diodes. Ordinary webcameras are used in systems like [15]. By using face detection algorithms, that system tracks the 3D position and orientation of the user's head. Webcamera-based face detection systems can handle different lighting conditions, but require a minimum level of lighting to be operational.

Webcameras are available in varying frame rates, where 30 fps is typical for modern webcameras. Poor lighting conditions often result in longer exposure times. Blurred images and decreased framerate are possible consequences of increased exposure time. It is therefore important to have good lighting conditions in the room to achieve good frame rates. However, the issues with dark rooms can be solved by using infrared cameras for face detection.

Display wall environments add extra requirements for head tracking systems. In contrary to desktop environments, where the user is bound to a desk, display wall environments make it possible for a user to walk around freely. The distance an object moves between each frame, becomes greater when the object's speed increases or the frame rate decreases. Therefore, too low frame rates can become an issue when tracked objects are moving quickly.

Using a face detection system for head tracking in a display wall environment would require multiple cameras to provide scalability. The major problem with face detection in these environments is probably when the user is located very close to the display. In this case, a single camera is unable to capture the user's entire face since there is no room to position the camera between the user's head and the display. However, two cameras located on each side of the user could probably be used to calculate the head position.

Natal [10] is an upcoming product from Microsoft. It enables the Xbox game console to perform full body motion tracking of multiple users and face detection. This is achieved by placing a camera rig in front of the user that is able to perceive depth information. Users will be able to interact with games without wearing any markers.

Another low cost tracking system is Playstation Move[20] from Sony. A camera located in front of the user detects the hand-held Playstation Move controller.

Chapter 2

Camera and Transformations

In this section, some of the terms and mathematics behind camera models are discussed. Techniques that this project uses, such as camera calibration and triangulation, are based on the mathematical camera model.

2.1 Camera

The most simple camera model is the pinhole camera [3]. As illustrated in Figure 2.1, a pinhole camera consists of a pinhole plane located in front of an image plane. The pinhole plane has a tiny hole that let light rays trough it. Some light rays that pass through the hole hits the image plane. In physical cameras, the image plane consists either of a film or a CCD[1].

The pinhole plane's hole defines the *projection center*[7]. The *optical axis* of a pinhole camera goes through the projection center and is perpendicular to the pinhole plane. The *focal length*, f, is the distance between the pinhole plane to the image plane. The coordinate system of cameras is normally defined using three axes. The X and Y axes are perpendicular to each other and span the pinhole plane. The Y axis defines the up direction of the camera. The Z axis is perpendicular to the pinhole plane, and defines the forward direction of the camera. The origin is located at the projection center of the camera. See Figure 2.2.

In Figure 2.1, there is an object located a position $(0, Y, Z)$. The two right triangles formed with the catheti y, f and Y, Z respectively are proportional. This proportional relationship can be described with the equation $-y/f = Y/Z$. The reason why y is negative is because it is flipped around the optical axis.

An imaginary image plane located at distance f *in front* of the pinhole plane results in images that are not flipped like the ones that appear on an image plane

[1]Charge-Coupled Device. Electronic light sensor used in digital cameras.

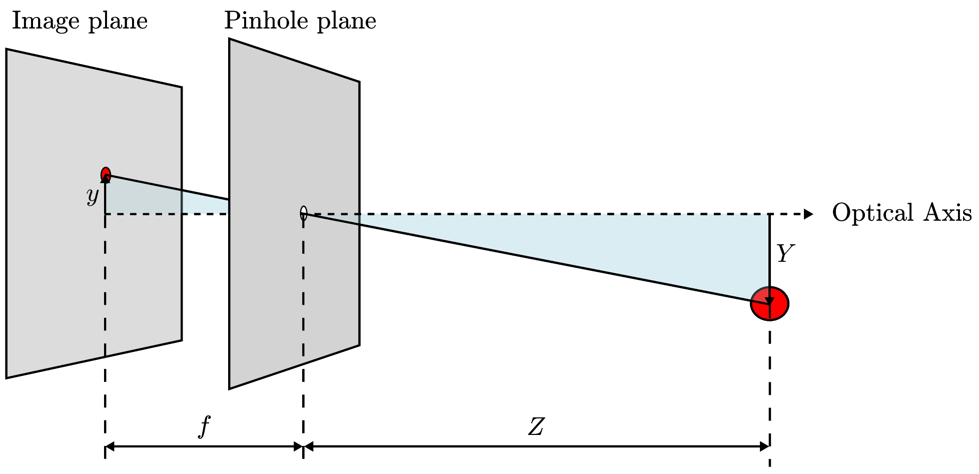


Figure 2.1: Pinhole camera model. (Based on Figure 11-1 in [3].)

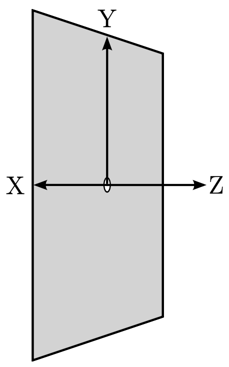


Figure 2.2: The coordinate system of a camera.

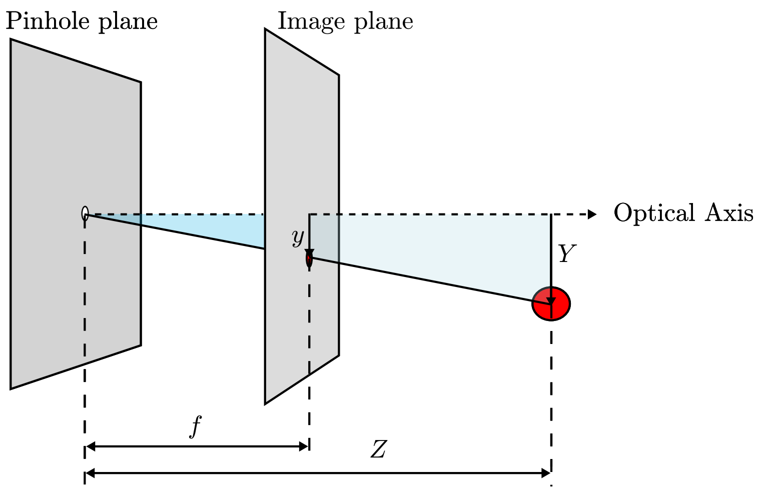


Figure 2.3: Pinhole camera model with frontal image plane.

located behind the pinhole plane. Figure 2.3 illustrates this setup. It is not possible to construct frontal image planes for physical cameras, but virtual cameras such as those used in computer graphics, use this type of abstraction. Having the image plane in front of the pinhole plane gives the following relationship

$$\frac{y}{f} = \frac{Y}{Z} \tag{2.1}$$

A point located at (X, Y, Z) in camera coordinates, is therefore projected onto the frontal image plane with

$$\begin{bmatrix} x \\ y \end{bmatrix} = \frac{f}{Z} \begin{bmatrix} X \\ Y \end{bmatrix} \tag{2.2}$$

The x and y values describe where on the image plane the object projected. Since digital images use pixels as units, the x and y values have to be converted to pixels. This is done by scaling the image coordinate with the scale factors s_x and s_y. The reason why two scaling factors are needed is because pixels are often rectangular. If $f_x = s_x f$ and $f_y = s_y f$ then

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x'/Z \\ y'/Z \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ Z \end{bmatrix} = \begin{bmatrix} f_x & 0 & 0 \\ 0 & f_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \mathbf{F} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{2.3}$$

The $\begin{bmatrix} x' & y' & Z \end{bmatrix}^T$ term is given in homogeneous coordinates. Homogeneous coordinates are converted to Cartesian coordinates by dividing all but the last component with the last component, i.e. $\begin{bmatrix} x'/Z & y'/Z \end{bmatrix}^T$.

The point where the optical axis intersects the image plane is called the *principal point.* In this point, the center of a captured image is located, i.e. $x = y = 0$. Due to imperfections in the camera manufacturing process, the principal point and the center of the image plane do not coincide. The distance from the principal point to the center of the image plane must therefore be taken into account when defining a camera mathematically. Let c_x and c_y describe the principal point in pixels as an offset from the top left corner of the image plane, then the following matrix can be constructed,

$$\begin{bmatrix} 1 & 0 & c_x \\ 0 & 1 & c_y \\ 0 & 0 & 1 \end{bmatrix} = \mathbf{G} \tag{2.4}$$

The focal length and the principal point are defined to be a camera's *intrinsic* parameters. The intrinsic camera matrix is constructed by the product of **G** and **F**, i.e.

$$\mathbf{GF} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} = \mathbf{P}. \tag{2.5}$$

2.1.1 Distortions

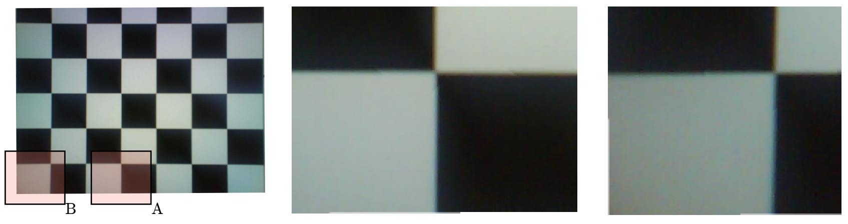
Due to the small size of the hole, the rate that rays hit the image plane is low. Therefore, the number of rays hitting the image plane during a given time depends on the size of the hole. Hence, the time the image plane is exposed to light (the exposure time) needs to be long for pinhole cameras. If the goal is to make an object appear sharp on the image plane, the camera should be placed stationary relative to the object during the exposure time. This property makes pinhole cameras unsuitable for moving objects. To overcome this problem, lenses are used to focus the light rays onto the image plane. A side effect with lenses is that they add *radial distortion* to the image. Radial distortion is caused by the lens bending the light rays non-uniformly. No radial distortion is apparent in the center of the image. But the distortion increases as a function of the distance from the image center. Figure 2.4 shows a photograph taken by a camera phone. The enlarged areas in (b) and (c) show that the vertical lines are straight and bent respectively. The bent lines are a result of lens distortion.

Another type of distortion is *tangential distortion.* This type of distortion occurs when the lens and image plane are not exactly parallel.

Distortions are categorized as the intrinsic parameters of a camera. The formulation of the distortion functions can be found in [7] and [3].

2.2 Transformations

This section is a brief introduction on how transformations are normally done in computer graphics. The change of position, orientation and size of objects in



(a) Original photograph. (b) Area A enlarged. (c) Area B enlarged.

Figure 2.4: A photographed checkerboard pattern demonstrating radial distortion. The vertical line appearing in (c) is slightly bent.

the real world and in a virtual world can be described by a *transformation matrix*. Objects in computer graphics normally consist of multiple interconnected vertices that create a shape that represent the object. A vertex contains a 3D point. When this point is multiplied with a transformation matrix, it will be translated (moved), rotated and/or scaled depending on the matrix. As shown in Equation 2.6, the matrix T represents a translate operation.

$$\mathbf{Tv} = \begin{bmatrix} 1 & 0 & 0 & t_0 \\ 0 & 1 & 0 & t_1 \\ 0 & 0 & 1 & t_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ 1 \end{bmatrix} = \begin{bmatrix} t_0 + v_0 \\ t_1 + v_1 \\ t_2 + v_2 \\ 1 \end{bmatrix} = \mathbf{v}' \tag{2.6}$$

where vertex, v, was translated with (t_0, t_1, t_2). Rotations are specified in the upper-left 3x3 submatrix of the transformation matrix.

$$\mathbf{Rv} = \begin{bmatrix} r_0 & r_1 & r_2 & 0 \\ r_3 & r_4 & r_5 & 0 \\ r_6 & r_7 & r_8 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ 1 \end{bmatrix} = \begin{bmatrix} r_0v_0 + r_1v_1 + r_2v_2 \\ r_3v_0 + r_4v_1 + r_5v_2 \\ r_6v_0 + r_7v_1 + r_8v_2 \\ 1 \end{bmatrix} = \mathbf{v}' \tag{2.7}$$

The three most elementary rotation operations are those around each axis. The matrices $\mathbf{R_x}$, $\mathbf{R_y}$, and $\mathbf{R_z}$ rotate with angle θ around the X-, Y-, and Z-axis respectively.

$$\mathbf{R_x} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.8}$$

$$\mathbf{R_y} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.9}$$

$$\mathbf{R_z} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.10}$$

Transformation matrices can be multiplied together into a transformation matrix that represents the combined operations. The order the matrices are multiplied together is relevant for the end result. In general it is true that $\mathbf{RT} \neq \mathbf{TR}$.

Each object has its own coordinate space, where the origin is defined relative to the object. If the object rotates, so does the space that is tied to the object. The coordinate space that is defined by an object is called the *object space*. Another space is the *world space*. The world space can be thought of as the room where the objects are located. The *camera space* is defined to have the origin located at the projection center of the camera. See Figure 2.2.

The inverse of a transformation matrix represents the inverse operation. For example, the inverse operation of a 90 degree rotation around the X axis, is a -90 degree rotation around the X axis. The inverse operation of a translation is the negative translation. This means that the operations a transformation matrix does are undone by the inverse of that matrix. Inverse matrices play an important role when it comes to transforming between spaces.

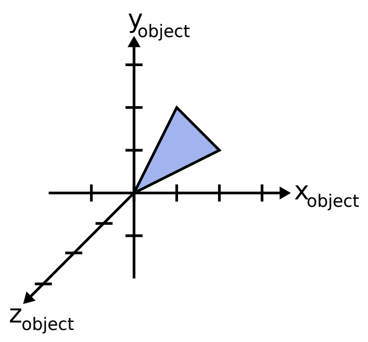
In computer vision and computer graphic systems, it is often important to know an object's position in camera space. This is done by multiplying the camera's inverse matrix with an object's world coordinates.

The example in Figure 2.5 shows a triangle shape in three different spaces: the object space, the world space and the camera space. The vertices of the triangle are defined in object space. As illustrated in Figure 2.5(a) they correspond to $\mathbf{v_0} = (0,0,0), \mathbf{v_1} = (2,0,0), \mathbf{v_2} = (0,2,0)$. The Z-axis is pointing outwards[2]. The transformation matrix $\mathbf{M}$ is used to transform the object coordinates into world space. In 2.5(b) the object space of the triangle is first rotated 45 degrees around z_{world} then translated $(3,2,0)$. The matrix that corresponds to these operations is,

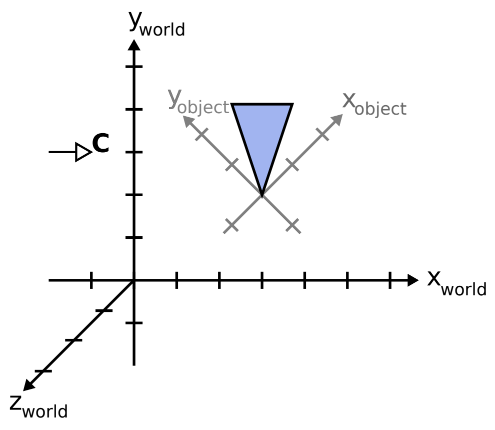
$$\mathbf{M} = \begin{bmatrix} \cos 45 & -\sin 45 & 0 & 3 \\ \sin 45 & \cos 45 & 0 & 2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.11}$$

The arrow next to the C symbol in 2.5(b), represents a camera position and its direction. The camera is located at $(-1,3,0)$. Since the object should be projected into an x, y coordinate on the camera's image plane, the camera coordinate space has to be rotated relative to the world coordinate space. A -90 degree rotation around the camera's Y-axis makes the camera's X-axis point outwards and its Z-axis point to the left. Hence, the negative Z-axis defines the direction where the camera "sees".

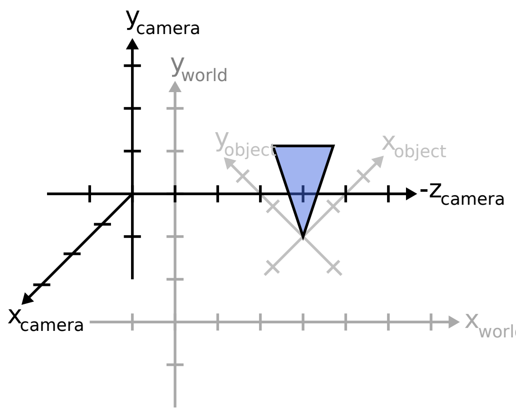
[2]This follows the right-handed coordinate system convention.



(a) Object space.



(b) World Space.



(c) Camera Space.

Figure 2.5: An object in three different spaces.

The camera matrix, **C**, defines the *extrinsic* parameters of the camera. Transforming world coordinates into camera coordinates is done by using the inverse of the camera matrix. The camera matrix and its inverse are defined as,

$$\mathbf{CC^{-1}} = \begin{bmatrix} \cos -90 & 0 & \sin -90 & -1 \\ 0 & 1 & 0 & 3 \\ -\sin -90 & 0 & \cos -90 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos 90 & 0 & \sin 90 & 0 \\ 0 & 1 & 0 & -3 \\ -\sin 90 & 0 & \cos 90 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} = I \ . \tag{2.12}$$

The complete transformation from object coordinates to camera coordinates is done with

$$\mathbf{C^{-1}Mv_i} = \mathbf{v_i'} \ . \tag{2.13}$$

$\mathbf{v_i}$ is transformed from object coordinates to world coordinates with **M**. The result is transformed further from world coordinates to camera coordinates with $\mathbf{C^{-1}}$. To project a point onto the camera's image plane, the camera's intrinsic matrix (Equation 2.5) is multiplied with the camera coordinate. Since $\mathbf{v_i'}$ is a 3D point in homogeneous coordinates it consists of 4 elements. The intrinsic camera matrix is a 3x3 matrix. As described in [5], an extra column of zeros has to be added to the camera intrinsic matrix, so that it has a valid dimension for multiplication with $\mathbf{v_i'}$,

$$\mathbf{P'} = \mathbf{P} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.14}$$

The two dimensional screen coordinate in Cartesian coordinates, $\mathbf{v_i''}$, is therefore expressed with the following equation,

$$\mathbf{P'C^{-1}Mv_i} = \mathbf{v_i''} \ . \tag{2.15}$$

2.3 View frustums

In computer graphics the term view frustum refers to the volume that is projected onto the image plane [18] of the camera. Everything outside this volume will not be visible. As illustrated in Figure 2.6, the shape of a view frustum is a pyramid with the tip cut off. The volume is specified by six parameters – left, right, bottom, top, near, and far. The near and far parameters define the distances from the camera to two planes – the near plane and the far plane. The left, right, bottom and top edges define the area of the near plane. These edges

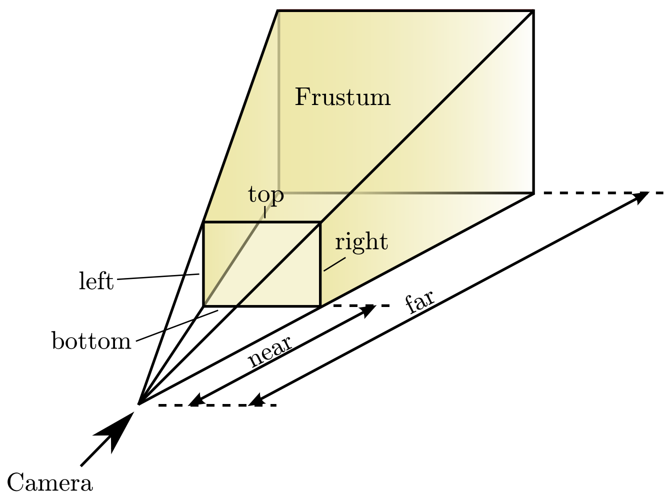


Figure 2.6: View frustum. (Based on Figure 3-13 in [18])

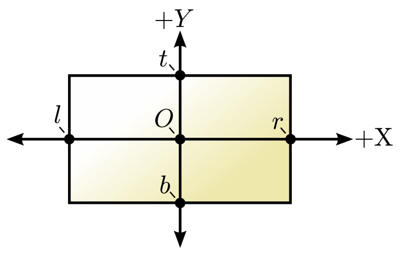


Figure 2.7: Near plane.

are specified relative to the origin of the near plane. Figure 2.7 shows that l, r, b, and t are coordinates on the X- and Y-axis of the near plane. A symmetrical view frustum is a frustum where the origin coincides with the center of the near plane, i.e. $-l = r$ and $-b = t$. The optical axis of the camera goes through the origin of the near plane. Hence, the origin of the near plane defines the image center.

A 3D scene that is projected onto a 2D image has a view that is based on the camera location relative to the image plane. The view in the projected image will appear correct when observed from the same location. If the image plane is configured to have the same measurements as a physical screen, the ideal viewing position will simply be the position of the camera. For example, if a camera is centered 0.5 meters in front of the image plane, the observer should also be centered 0.5 meters in front of the monitor.

3D applications, such as computer games, normally assume that the observer is stationary. Hence, the view frustum in these applications is often static.

Chapter 3

Design

3.1 Architecture

Figure 3.1 shows the architecture of the system. Wiimotes capture positions from infrared sources worn by the user (1). A computer provided with a Bluetooth interface is required to be able to read data from the Wiimotes. The positional data captured by the Wiimotes are sent (2) to the computer running the tracking system (3). The tracking system translates the received positional data to a tracking event. The tracking event is forwarded to a server running on the display wall's front-end (4). The role of the front-end is to avoid direct communication with all tiles. The server translates the tracking event into a new event that is distributed to all tiles (5).

With this architecture it is possible to track a user's head. In addition, it is possible to give the user visual response based on the position of the user's head. The architecture also allows for multiple users wearing infrared LEDs to be tracked.

3.2 The Tracking System

The system follows these steps to calculate the eye position of the user.

1. One or more infrared LEDs is observed by two Wiimotes.
2. The observed 2D positions are analyzed and triangulated.
3. The triangulated point(s) is used to calculate the eye position.

In Figure 3.2 the overall design of the system is outlined. The environment refers to the room where the user is located. The cameras capture the infrared LEDs that the user is wearing. The infrared LEDs are able to move in all three dimensions. The cameras are only able to present what they see in two dimensions. Since the system's knowledge about the environment is only based on

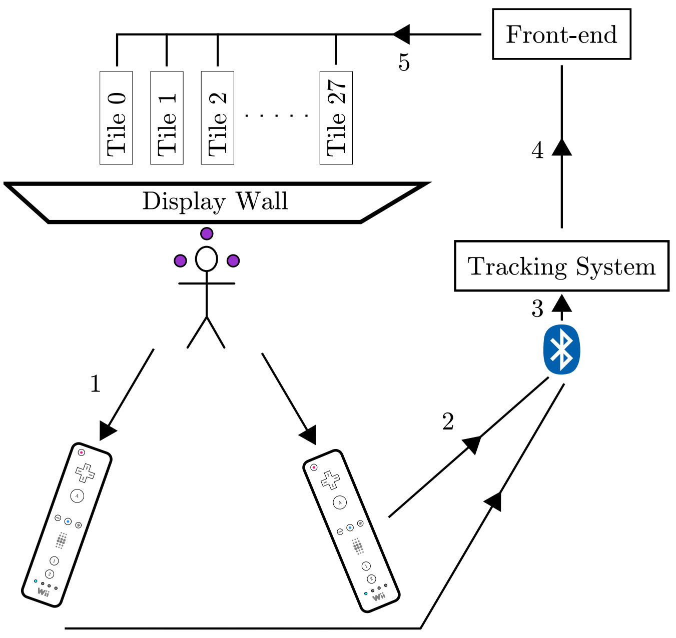


Figure 3.1: System architecture.

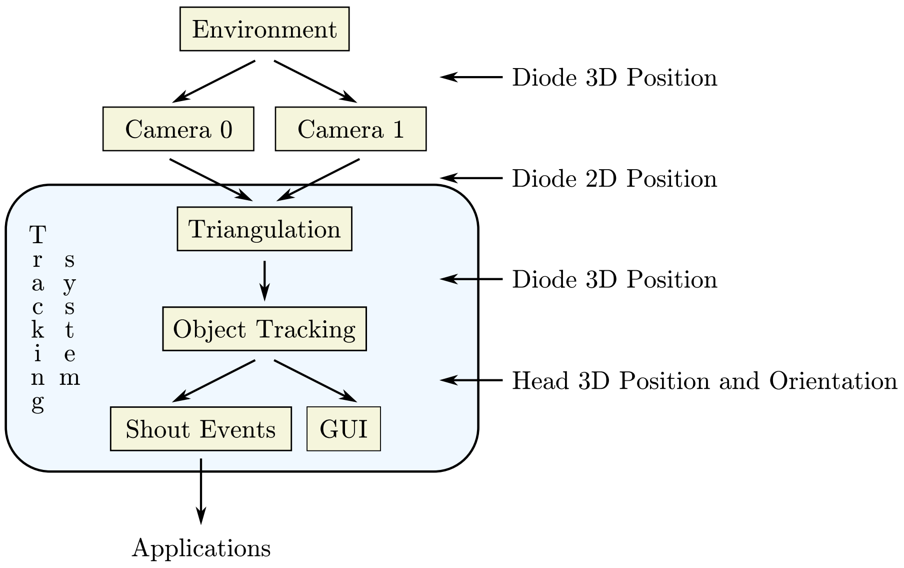


Figure 3.2: The overall design of the tracking system.

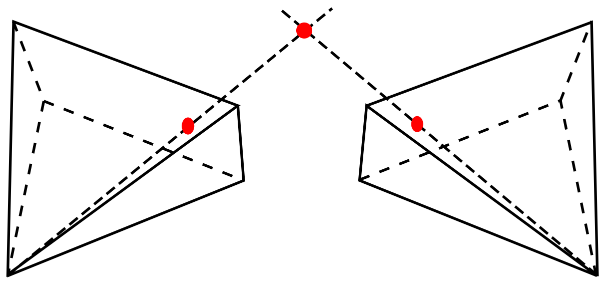


Figure 3.3: Triangulation (Based on Figure 2 in [17])

what the cameras see, the system only has 2D representations of the environment. If multiple 2D images are available, triangulation can be used to calculate the 3D position of an observed point.

After the triangulation step in the tracking system, one or multiple 3D positions are available. Knowledge of how these 3D positions are located relative to an object, assumptions of that objects position and possible orientation can be made. The pose (position and orientation) of the tracked object can be presented in a GUI or be sent as a network event. The network events make it possible to control other parts of the system, such as the display wall.

3.2.1 Triangulation

Triangulation is a technique that can be used for obtaining a 3D position from two 2-dimensional images [17]. To perform triangulation, a camera's intrinsic and extrinsic parameters must be known. To find the 3D position of a point, two rays are constructed. These rays begin in each camera's projection center and goes through the projected point on the image plane. The point where these two rays intersect is the result of the triangulation. This is illustrated in Figure 3.3. In reality these two rays don't intersect, but are very close to an intersection. Therefore, it is required to find the point, where the distance between the rays is the smallest.

3.2.2 Object Tracking

The system offers three types of tracking. We call these different types of tracking, 6 DoF tracking, 6 DoF hybrid tracking, and position tracking.

6 DoF tracking

6 DoF stands for six degrees of freedom. Three of these six degrees of freedom refers to the 3D position of the tracked object. The other three degrees of freedom refers to the rotation around three axes. Hence, with 6 DoF tracking it is possible to track an object's 3D pose.

6 DoF tracking is achieved by triangulating three points. These three points must be positioned in the same triangular pattern as the positions of the IR

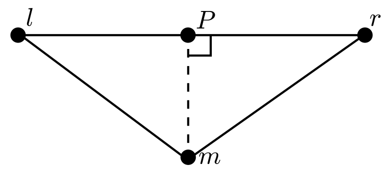


Figure 3.4: Calculating the object space.

LED clusters shown in Figure 1.4. To calculate the pose of the object where these three infrared sources are attached is done as follows. Three 3D points are given. These 3D points are sorted from left to right (along the X axis of the coordinate system they are located in). The three points can now be addressed as left, middle and right (l, m, and r respectively in Figure 3.4). The line $\overrightarrow{lr}$ is defined and goes from the left point to the right point. The point on $\overrightarrow{lr}$ that has the shortest distance to m is called P. The line $\overrightarrow{mP}$ will be perpendicular to $\overrightarrow{lr}$. The normalized vector $\overrightarrow{mP}$ serves as the forward vector of the object, and the normalized vector $\overrightarrow{lr}$ the right-vector. The cross product of the forward-vector and the right-vector defines the up-vector of the object. The rotation matrix of the tracked object is constructed by using the forward-, right- and up-vectors ($\mathbf{f}$, $\mathbf{r}$, and $\mathbf{u}$, respectively) as columns. The middle point, m, is defined as the origin of the tracked object's space. The transformation of a point $\mathbf{e}$ in object space to world space is shown in Equation 3.1.

$$\begin{bmatrix} f_x & r_x & u_x \\ f_y & r_y & u_y \\ f_z & r_z & u_z \end{bmatrix} \mathbf{e}_{object-space} + \mathbf{m} = \mathbf{e}_{world-space} \tag{3.1}$$

6 DoF hybrid tracking

Triangulating three points requires three points to be observed by both cameras. This limits the space where the object is able to be located. The term hybrid in 6 DoF hybrid tracking refers to the combination of triangulation and a geometric approach. With 6 DoF hybrid tracking, it is only required for each camera to see two points. As illustrated in Figure 3.5, the left Wiimote sees two points, point 0 and point 1. It is assumed that point 2 is occluded by the head. The right Wiimote only sees point 1 and 2 whereas point 0 is occluded. The common point appearing in both Wiimotes is point 1. This point is triangulated to obtain its 3D position. To calculate the orientation of the object the 3D position of point 0 and point 2 needs to be known.

A line is created that starts in the left Wiimote's projection center and goes through point 0. This line is created by extending the ray that starts in the projection center and ends in point 0's projected position in the image plane. As with triangulation, it is not possible to know where on this line point 0 is located. However, the distance between point 0 and 1 is known. This makes it possible to create a sphere that is centered on point 1 with a radius equal to the distance between point 0 and 1. The position where the line and the

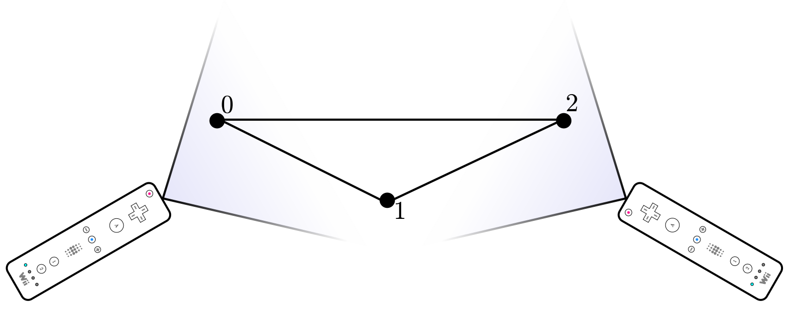


Figure 3.5: 6 DoF hybrid tracking.

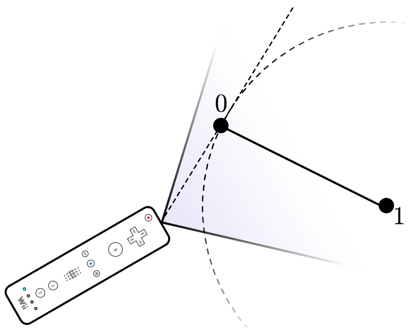


Figure 3.6: Ray-sphere intersection test.

sphere intersect will therefore be the 3D position of point 0. This is illustrated in Figure 3.6. The same technique is used with the right Wiimote to calculate the position of point 2.

When the position of all points are known the orientation of the object is found in the same way as described for 6 DoF tracking, i.e. find the forward-, right-, and up-vector.

A line-sphere intersection test results in: (i) no intersection if a ray does not intersect the sphere; (ii) one intersection if a ray does not penetrate the sphere, but only touches the sphere's surface; (iii) two intersections are returned if the ray goes through the sphere, i.e. the entry point and the exit point. The intersection test often fails when using a sphere that has a radius that equals the distance between the two points. When the sphere is slightly expanded, the line is forced to intersect sphere. The point of intersection in this case will not be exact. Hence, the 6 DoF hybrid tracking technique is more approximate than the previously described 6 DoF tracking. However, the advantage with 6 DoF hybrid tracking is that the object can be tracked in a larger space.

Position tracking

Position tracking has a simpler implementation, since the orientation here is predefined and not calculated. This type of tracking only requires one triangulated point. This point is used as the origin of the tracked object space. The

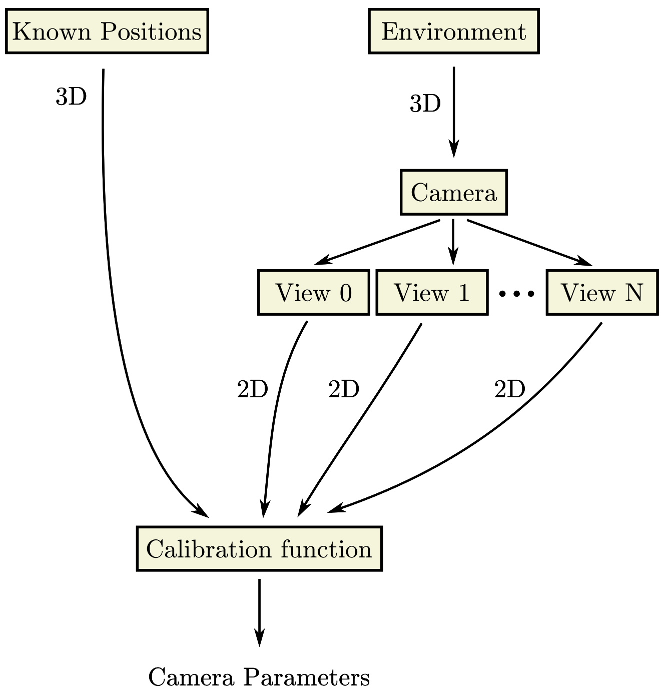


Figure 3.7: Camera calibration.

axes in the tracked object space are aligned with the axes of world coordinate system.

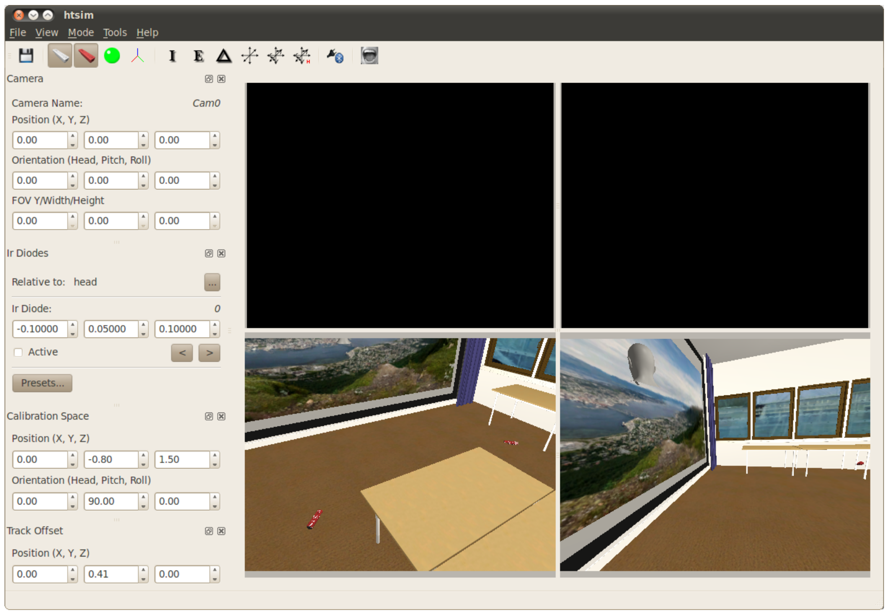
With position tracking, it is only possible to know where the tracked point is located. Since the object orientation is unknown, other points of the object cannot be determined. For example, if it is known that point **e** is located 2 units in front of a tracked point, it is not possible to describe where that point is located in the room because of the lacking orientation of the object. However, with the 6 DoF tracking technique this is possible.

When 6 DoF tracking is used, the system requires both IR cameras to observe three points each. If a camera observers less than three points, triangulation of all three points cannot be accomplished. Hence, the tracked pose cannot be reported. The requirement to observe three points each frame limits the space the object can be tracked in. Since position tracking only requires one infrared source to be tracked, it is possible to track the object in a larger space but with lacking orientation information.

3.3 Calibration System

Camera calibration is essential to get accurate results for triangulated points. Both the intrinsic and extrinsic parameters of the camera need to be calibrated.

Figure 3.7 illustrates the camera calibration step. To calculate the camera parameters automatically, multiple points with known 3D positions, together



Camera

Camera Name: *Cam0*

Position (X, Y, Z): 0.00, 0.00, 0.00

Orientation (Head, Pitch, Roll): 0.00, 0.00, 0.00

FOV Y/Width/Height: 0.00, 0.00, 0.00

Ir Diodes

Relative to: head

Ir Diode: *0*

-0.10000, 0.05000, 0.10000

☐ Active

Calibration Space

Position (X, Y, Z): 0.00, -0.80, 1.50

Orientation (Head, Pitch, Roll): 0.00, 90.00, 0.00

Track Offset

Position (X, Y, Z): 0.00, 0.41, 0.00

Figure 3.8: Overview of the HTSIM GUI.

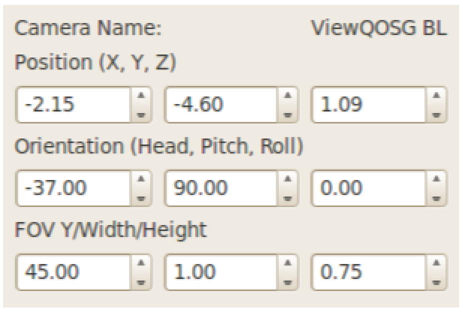
with camera images that contain the corresponding 2D projected points are fed into a camera calibration function. The calibration function returns both the intrinsic and extrinsic parameters for the camera.

The known 3D points are the infrared LEDs of the calibration rig. A Wiimote directed to the calibration rig reports the observed positions to the system. When the calibration rig (or the Wiimote) is moved, a new view of the calibration rig is obtained. When enough multiple views of the calibration rig are registered, the calibration function together with the known LED positions are used to calculate the camera's parameters.

3.4 htsim – The Head Tracking Simulator

The head tracking simulator, called HTSIM, is a tool that integrates the tracking system and the calibration system. The application allows for developing in a simulated environment, which requires less effort than developing in a physical environment. Algorithms developed in the simulated environment can be used directly in the physical environment. HTSIM is not only bound to the simulated environment. The application has the possibility to be used together with the physical environment. Since the application is not only a simulator, the name HTSIM is somewhat misleading.

The program has two modes – *real* mode and *virtual* mode. When real mode is enabled, the purpose of the application is to perform head tracking with the physical environment. When virtual mode is enabled, HTSIM is used as a head tracking simulator.


Camera Name: ViewQOSG BL

Position (X, Y, Z)

-2.15 -4.60 1.09

Orientation (Head, Pitch, Roll)

-37.00 90.00 0.00

FOV Y/Width/Height

45.00 1.00 0.75

Figure 3.9: Camera paramers.

The application, shown in Figure 3.8, has four configurable viewports. The upper two viewports show what the infrared cameras sees. The content of these viewports depends on what mode the program is in. In real mode, the infrared output of two Wiimotes is shown in their respective viewport. In virtual mode, the output of virtual cameras is displayed in the viewports.

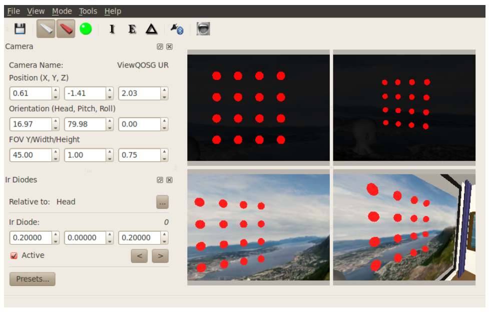
The bottom two viewports show the entire virtual scene. The room is an accurate model of the Tromsø Display Wall laboratory. It contains windows, tables, and the display canvas. An animated human head model is placed in front of the display wall. It is possible to place infrared sources at different positions inside the room. Wiimote models are used to illustrate where the Wiimotes are placed.

In virtual mode, each viewport represent a virtual camera. When the user clicks on a viewport, the viewport is activated and that camera's parameters are shown in the left pane of the GUI. The user has the possibility to modify the camera's parameters by manually typing the values in the GUI or by dragging the view in the viewport. As shown in Figure 3.9, the controllable values for each camera are position, orientation, horizontal field of view, and aspect ratio. Rotations are first applied around the Z-axis (heading), then in the oriented X-axis (pitch), and finally around the twice oriented Y-axis (roll)[1]. Horizontal field of view is specified in degrees. The relationship between the width and height parameters defines the aspect ratio.

3.4.1 IR Diode placement

The user has the possibility to place virtual IR sources in the scene. This is done in the GUI. It is possible to use up to 16 diodes in virtual mode. In real mode, this is limited to four since the Wiimote tracks at most four IR diodes. The diodes are positioned relative to a user selected scene object. This makes it possible to define what coordinate space the diodes are placed in. E.g. head space, wall space, table space etc. The position of each diode can be manually modified, as well as its state. HTSIM has a tool for constructing calibration patterns, which creates a set of diodes placed in a grid. The Preset button makes it possible for the user to set the number of diodes (2x2, 3x3, or 4x4). A

[1]The entire coordinate system rotates when rotation around an axis is performed. This is contrary to rotating around the axes' original position.



Camera Name: ViewQOSG UR
Position (X, Y, Z) 0.61 -1.41 2.03
Orientation (Head, Pitch, Roll) 16.97 79.98 0.00
FOV Y/Width/Height 45.00 1.00 0.75
Relative to: Head
Ir Diode: 0
0.20000 0.00000 0.20000
☑ Active

Figure 3.10: A 4 × 4 pattern with 0.1 spacing.

	Virtual Mode	**Real Mode**
Input to Calibration Function	Yes	Yes
Observed Positions	Yes	No

Table 3.1: Usage of virtual IR sources.

spacing parameter can be adjusted that defines the space between each diode. Finally, the entire pattern can be offset relative to the origin of the space it is connected to. See Figure 3.10.

The purpose of the virtual IR sources is twofold. The positions of the virtual IR sources are used in the calibration process in both virtual mode and real mode. The calibration function needs information about the 3D positions of the observed 2D points. The IR source positions, configured in the application, are used as these 3D positions for the calibration function. Both modes use the same calibration function.

The cameras in virtual mode project these 3D points onto their image plane and use these projected 2D points as the observed points. In real mode, the observed points are obtained from the Wiimotes based on the real environment. Observed points are used in the calibration function as well as in the tracking system. The usage of the virtual IR sources is summarized in table 3.1.

3.4.2 Calibration

Intrinsic calibration has to be done once for each individual camera. Virtual cameras don't need calibration because their intrinsic parameters are the user defined parameters of the camera. However, the application allows for calibrating virtual cameras. In this way, it is possible to compare the calibrated values with the known values of the virtual camera. This is a good way to validate the calibration function. By varying the number of calibration points and the pose of the IR camera relative to the calibration rig it is possible to see what influence it has over the calibration correctness. Once the intrinsic calibration is done, the parameters can be saved and be used in all subsequent runs of the application as long as the camera type does not change. Extrinsic calibration should be done every time a camera's pose is altered.

Since the calibration function uses the virtual IR diodes as input, it is important the correct spacing between the diodes are specified GUI so that they match the real diode spacing.

3.4.3 Triangulation

When the triangulation state is entered, the cameras have to be stationary. The previously calibrated camera parameters are used in the triangulation process. Triangulated diodes appear as green spheres in the bottom two viewports. Triangulation is only working when the number of diodes is equal for both cameras.

3.4.4 Connect Wiimotes

To connect the Wiimotes to HTSIM a Bluetooth interface has to be used. The current implementation of the HTSIM only uses the Bluetooth interface with `hci0` as device id. Every new Wiimote that should be used with the system has to be specified. Under the Settings dialog, found in the Tool menu, it is possible to search for new Wiimotes. By clicking the ellipsis button a Bluetooth device scan is initiated. The devices that appear as "Nintendo" are the Wiimotes. When a device scan is made, it is important to set the Wiimotes in discovery mode. This is done by pressing the 1 and 2 buttons simultaneously on the Wiimotes.

3.5 Applications

To demonstrate the head tracking system a head tracking application was created. The OSGDWL application (OpenSceneGraph Display Wall Lab viewer), developed as part of this project, is a program that displays 3D models on the display wall. The program can be started in several modes – one client mode and three different server modes.

3.5.1 Client

OSGDWL is started on each tile in client mode. A client instance of the program loads and displays a specified 3D model. The 3D model will appear on the display wall as one coherent image. This is achieved by letting each tile render a part of the image. With other words, each tile needs to have its own view frustum. Keeping track of 28 individual view frustums can be cumbersome during developing. Therefore, one view frustum for the entire display is specified – the wall view frustum.

The wall view frustum is setup to match the physical measurements of the entire display with width, W, and height, H[2]. The width of a tile, w, equals W/X, where X is the number of horizontal tiles. The height of a tile, h, equals H/Y, where Y is the number of vertical tiles. The host names of the tiles are in the form `tile-X-Y`, where `X` defines the horizontal tile coordinate and `Y` defines the

[2]For Tromsø Display Wall laboratory, $W = 5.185$ m and $H = 2.238$ m

vertical tile coordinate. `tile-0-0` is located in the lower left corner. If x and y are the tile coordinates, the l, r, b, and t parameters (introduced in Section 2.3) for a tile becomes,

$$\begin{bmatrix} l_{x,y} \\ r_{x,y} \\ b_{x,y} \\ t_{x,y} \end{bmatrix} = \begin{bmatrix} (-1 + (x+0) * (2/X)) * (W/2) \\ (-1 + (x+1) * (2/X)) * (W/2) \\ (-1 + (y+0) * (2/Y)) * (H/2) \\ (-1 + (y+1) * (2/Y)) * (H/2) \end{bmatrix} \tag{3.2}$$

3.5.2 Server

When OSGDWL is started in one of the three server modes it sends out messages on a regular basis to all clients. These messages contain a camera pose and view frustum data. The tiles use the same messages. The camera pose can be used directly by the clients since they are supposed to display a scene from a single camera's point of view. However, the frustum data cannot be used directly since each tile has its own dedicated frustum. This is solved by defining the frustum data in the message to be relative to the original calculated frustum. When a client receives the frustum data it adds the data to its already calculated frustum data in this way,

$$\begin{bmatrix} l'_{x,y} \\ r'_{x,y} \\ b'_{x,y} \\ t'_{x,y} \end{bmatrix} = \begin{bmatrix} l_{x,y} \\ r_{x,y} \\ b_{x,y} \\ t_{x,y} \end{bmatrix} + \begin{bmatrix} m_l \\ m_r \\ m_b \\ m_t \end{bmatrix} \tag{3.3}$$

where m_l, m_r, m_b, and m_t are the received data. $l'_{x,y}$, $r'_{x,y}$, $b'_{x,y}$, and $t'_{x,y}$ is the final value for a tile's view frustum. The near and far values are also part of the camera message, but these are specified with absolute values, since these values should be the same for all tiles.

Server modes

The three server modes are called *loop*, *touch*, and *track*. The loop-mode is non-interactive mode where the server sends a series of camera positions based on a hard-coded animation function. This mode is useful for testing the tiles and to demonstrate a model without interaction.

In touch-mode, the server obtains touch-events from the camera-sense system. The touch events are converted to camera events, which are distributed to all tiles. The purpose of this mode is to present a correct view of a 3D scene for the user. This is done by estimating the head position of the user based on the user's hand position. When the hand of the user is held at the same height as where the user's eyes are located, the system is able estimate the position on the display where the head is located. The near plane value specified in the configuration file of OSGDWL, serves as the approximate distance between the user and the screen. The system is now able to calculate an approximate 3D

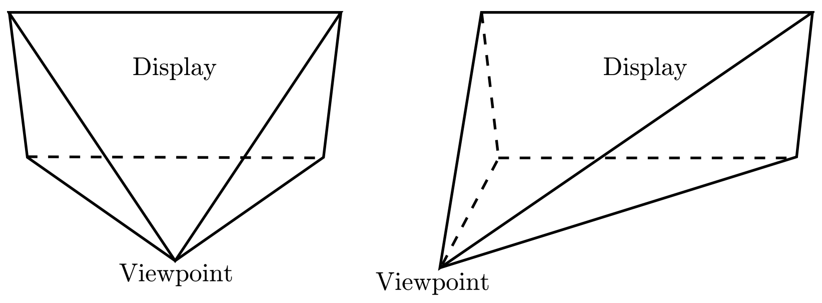


Figure 3.11: The user dependent view frustum.

eye position. This position is used for constructing a view frustum. When the user moves in front of the image with the arm extended, the view frustum will "follow" the user.

In track-mode, OSGDWL uses the 3D position provided by the tracking system.

Chapter 4

Implementation

All code developed in this project is written in C++ except some small Python programs used for statistical purposes in the experiment section. The computer where all development and testing took place uses Ubuntu 10.04 as operating system. The display wall cluster uses CentOS 4.2 as operating system.

4.1 3rd party libraries

HTSIM uses Qt [14] as graphical user interface API. The OpenGL based scene graph library, OpenSceneGraph [13], is used for all 3D graphic rendering. For camera calibration and triangulation OpenCV [12] is used. Shout [21] is used for all network communication. The CWIID [19] library is used for communicating with the Wiimotes.

4.1.1 Development Environment

Qt Creator [14] is used as this project's integrated development environment (IDE). The IDE is part of the Qt SDK. However, programs made with the Qt Creator do not necessarily need to be Qt dependent. On Linux, the C++ compiler from GCC is used by the IDE. Qt Creator stores the project file in the Qt native .pro format. The tool `qmake` uses .pro files and makes it possible to generate project files for various build systems, e.g. Makefile for GCC and sln/vcproj files for VisualStudio. Qt Creator has a graphical debugger. Programs meant to be run on the cluster of the display wall lab also need to be built on the cluster. Qt Creator has the possibility to add different build configurations. Build configurations were created making it convenient to switch between local building and building on the cluster.

4.1.2 OpenSceneGraph

OpenSceneGraph is a scene graph library that uses OpenGL for rendering. A scene graph is a way to structure 3D graphic data. In OpenSceneGraph this is represented as a directed acyclic graph (DAG). This hierarchical structure has several benefits when it comes for real-time 3D computer graphics rendering. The nodes in the scene graph consist of different types. Typically the leaf nodes contain some sort of visual information, such as geometries and colors. In OpenSceneGraph, nodes that contain visual information are called `Geodes`, which stands for Geometry Node. A `Group` node is able to contain several other nodes. The `MatrixTransform` node type is special group. It contains a matrix that defines a transformation that will be applied to all its children. `MatrixTransform` nodes can be used to define a coordinate space for the underlying `Geodes`. This property makes the scene graph a helpful abstraction for 3D artists and developers to organize large scenes.

File formats

OpenSceneGraph provides readers and writers for many popular 3D graphics and image formats. OpenSceneGraph provides two native formats for storing and reading the scene graph – the .osg and the .ive format. The .osg format is an ASCII formatted file format while the .ive format is a binary format. The images used on 3D geometry are called textures. The textures are not stored in the .osg format, but are referred with file paths inside the .osg file. The .ive format stores all textures within the file. All models used in this project are stored in one of these two formats.

Rendering pipeline

The 3D scene is typically rendered with the same frequency as the screen's refresh rate. For each frame, OpenSceneGraph prepares the scene and draws it. This is done in three phases, the update-, the cull-, and the draw-phase [9]. Each phase consists of one or multiple scene traversals.

In the update phase, the entire scene graph is traversed. Nodes that are animated or should be moved to new locations are updated in this phase. Each node in OpenSceneGraph is able to have an update callback tied to it. This callback is executed when the node is traversed. The callbacks enable the developer to add custom functionality to OpenSceneGraph native node types.

In the cull phase, a rendering list is built that is dependent on the current view. Culling is used to remove details from the scene that will not be visible or that is not needed in the rendered image. This phase is used to improve rendering timings. Also here, the developer is able to define custom callbacks for the nodes.

In the draw phase, the rendering list that was built in the culling phase is used for rendering. The rendering list is translated into several OpenGL function calls. Hence, the developer does not need to use any OpenGL functions directly, this is handled by OpenSceneGraph.

OpenSceneGraph is able to perform scene traversals in parallel by distributing the traversals on different threads. The update traversal is always run on a single thread. Since the subsequent traversals depend on the output of the update traversal, they cannot start until the update phase has finished. If

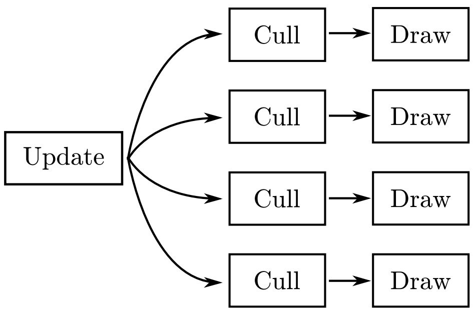
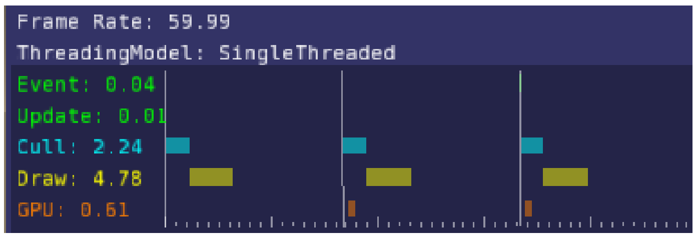


Figure 4.1: OpenSceneGraph scene traversals.



```
Frame Rate: 59.99
ThreadingModel: SingleThreaded
Event: 0.04
Update: 0.01
Cull: 2.24
Draw: 4.78
GPU: 0.61
```

Figure 4.2: OpenSceneGraph statistics.

multiple views are used, as in the case of HTSIM, multiple view-dependent culling executions need to be initialized. Since the culling executions do not depend on each other, and do not modify the scene graph, these can be executed on individual threads. The number of draw executions equals the number of culling executions. This is illustrated in Figure 4.1.

Figure 4.2 is a screenshot from the built-in statistics of OpenSceneGraph. The statics presents timings in ms for the various executions on the program. The scene used when obtaining the statistics is a relative complex scene. However, no update callbacks are attached to this scene. This explains the short time used for the update traversal (0.01 ms). The cull and draw phase use 7 ms together. The total time for the frame preparation is therefore slightly more than 7 ms. New frames are drawn synchronous to the refresh rate of the display, in this case every 16.7 ms. Consequently, even if the frame has been prepared nothing is yet displayed. The frame will be displayed at the next screen refresh. The vertical white lines in the figure indicate the beginning of a new frame, i.e. a screen refresh. This means that OpenSceneGraph will be idle in $16.7 - 7$ ms before the next frame is handled.

In HTSIM, many of the computer vision related tasks are implemented in update callbacks. This means that tracking is performed at the same frequency as the scene is rendered, namely 60 Hz.

4.1.3 OpenCV

OpenCV is a library that contains functionality for several areas in the computer vision field. This project uses camera calibration functions and other mathematical functions provided by the library

Calibrating a camera's intrinsic parameters is done with the `calibrateCamera` function.

```
double calibrateCamera (objectPoints, imagePoints, imageSize,
                        cameraMatrix, distCoeffs, rvecs, tvecs)
```

objectPoints is a list of the known 3D points of the LEDs in the calibration rig. INPUT

imagePoints is a list of lists – a list of images where each image is a list of points. The images represent different views of the calibration rig. INPUT

imageSize is the size of the image. When working with Wiimotes, the image size should be 1024×768. INPUT

cameraMatrix is the camera's intrinsic matrix calculated by the function. This matrix has the same form as the matrix in Equation 2.5. The matrix describes the camera's focal length and its principal point. The function is also able to use this matrix as an input parameter. In this case the matrix is used as a hint for the calibration calculations. INPUT/OUTPUT

distCoeffs are the coefficients for the radial and tangential distortion functions. See Section 2.1.1. OUTPUT

rvecs is a list of orientations describing the camera orientation for each view in `imagePoints`. OUTPUT

tvecs is a list of translations describing the camera translation for each view in `imagePoints`. OUTPUT

The `calibrateCamera` function outputs both the intrinsic and extrinsic camera parameters. However, once the intrinsic parameters have been calibrated they remain unchanged. Calibrating extrinsic parameters only is done more efficient with the `solvePnP` function.

```
void solvePnP(objectPoints, imagePoints, cameraMatrix,
              distCoeffs, rvec, tvec)
```

objectPoints is a list of the known 3D points of the LEDs in the calibration rig. INPUT

imagePoints is a list of points from an image. INPUT

cameraMatrix is the camera's intrinsic matrix. This matrix is the previously calibrated matrix by the of the `calibrateCamera` function. INPUT

distCoeffs are the coefficients for the radial and tangential distortion functions. The distortion coefficients calculated by `calibrateCamera` function should be used here. INPUT

rvec is the calculated camera orientation, which is part of the camera's extrinsic parameters. OUTPUT

tvec is the calculated camera translation, which is part of the camera's extrinsic parameters. OUTPUT

Both functions require that the order of the specified 3D object points corresponds to the order of the 2D image points. The implementation of correspondence finding is discussed later in this chapter.

4.1.4 Shout

Shout [21] is an event system used for communicating with all participating computers in the system. It is based on the client-server model, where one central server communicates with one or more clients. Only reliable communication channels are used. This guarantees for no data loss and in-order delivery of events. Events are sent in binary form. The events have a predefined header, and allows for attaching user data of variable size.

It is possible to configure Shout so that a central server sends the same events to all its connected clients. This makes it possible for synchronizing the visual output of multiple clients.

In this project, two types of event types have been created – the object pose event and the camera event. The object pose event is created by the tracking system and represents the pose of a tracked object. It contains an object id, a rotation matrix, and a translation vector. The object id allows for distinguishing between multiple tracked objects. The rotation matrix and the translation vector define the pose of the tracked object. The rotation matrix could have been replaced by three Euler angles and a convention of how these Euler angles should be applied. However, a rotation matrix is used for implementation conveniences and to avoid ambiguities.

The camera event is used for informing the tiles of the display wall what region of the scene to draw. The camera event consists of a camera position, a camera rotation, a distance parameter, and parameters that define the view frustum. The camera position and orientation define where the camera is located in the 3D world. The distance parameter is unused, but can be used for defining the distance from the camera to an object that the camera is directed at. The view frustum parameters are defined with relative values. The tiles have individual regions of the entire wall view frustum. The view frustum parameters in the camera event define therefore how much their view frustums should be changed.

4.2 htsim

The section starts with a brief description of the various tools followed with more detailed description of the various parts HTSIM.

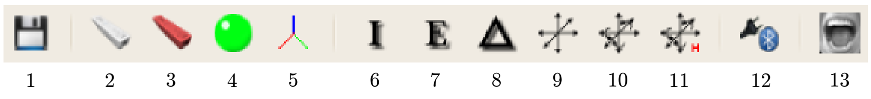


Figure 4.3: The toolbar of HTSIM.

4.2.1 Tools

Figure 4.3, shows the toolbar in the program. Every icon represents an action. These actions are also found in the program's menu. The following list describes the numbered actions in 4.3.

1. Save the configuration
2. Show Wiimotes
3. Show Calibrated Wiimotes
4. Show triangulated LEDs
5. Show axis of the calibration space.
6. Start the intrinsic calibration tool
7. Start extrinsic calibration
8. Enable triangulation
9. Enable Position tracking
10. Enable 6 DoF tracking
11. Enable 6 DoF hybrid tracking
12. Connect the Wiimotes
13. Send tracked poses as shout events.

Less commonly used actions can only be found in the menu. These actions are,

Mode/Virtual the program will use virtual cameras as image sources.

Mode/Real Real the program will use Wiimotes as image sources.

Tools/Match Wiimotes to Viewports Overrides the extrinsic calibration of the Wiimotes by matching the calibrated Wiimotes with the current view of the topmost viewports.

Tools/Settings... Specifies the MAC addresses of the Wiimotes.

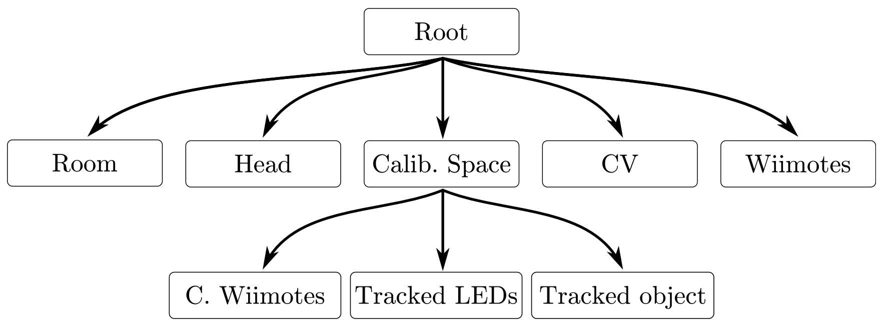


Figure 4.4: The scene in HTSIM.

4.2.2 Scene Construction

When HTSIM is started, the scene graph for the virtual scene is constructed. Figure 4.4 shows an overview of the scene graph. This section details out the various parts of the scene graph.

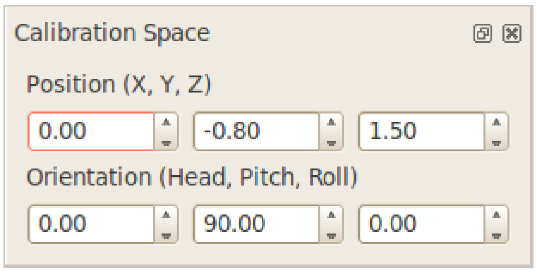
All 3D models are stored in a directory named `models`. Their file format is one of OpenSceneGraph's native file formats, ive and osg. The root node of the scene graph is of type `Group`. All nodes in the scene are children of this node. The model of the display wall lab is loaded from the `dwl.ive` file and is added as a child of the root node. The model contains its own scene graph where all objects in the room are represented with various nodes. The node is labeled *Room* in the figure.

An animated head model, located in the `head.ive` file, is also loaded. The animation cycles between three head orientations. The head turns left, then right, and finally straight. The purpose of this model is to simulate a human head that is in motion. The model is not added directly to the scene root. Instead, a `MatrixTransform` node, named *Head* in the figure, is created and added as a child of the root node. The loaded head model is added to this matrix transform node. The *Head* node acts as the "head-space", i.e. when the pose of the matrix transform is altered it will affect the pose of the head model accordingly.

Another `MatrixTransform` node defines the calibration space (labeled *Calib. Space* in the figure). The purpose of the calibration space is described in more detail later in this report. The calibration space node has models of Wiimotes, tracked LEDs, and the tracked object. The Wiimotes represent the calibrated position of the Wiimotes, i.e. the position where the system believes the real Wiimotes are located. The tracked LEDs are the triangulated positions. These are represented as green dots in the scene. The tracked object node contains one or more tracked objects. The tracked objects are represented by three axes in different colors.

The *CV* node does not contain any geometry. Instead, it has a callback that performs computer vision related tasks each update traversal.

Finally, two Wiimote models are added as children of the root. These models represent the pose of the virtual cameras for the topmost viewports. To distin-


Calibration Space

Position (X, Y, Z)

0.00 | -0.80 | 1.50

Orientation (Head, Pitch, Roll)

0.00 | 90.00 | 0.00

Figure 4.5: The calibration space. It is possible to offset the calibration space relative to the world coordinate space.

guish these Wiimote models with the calibrated Wiimote models different colors are used. The Wiimotes residing in the calibration space have a red color while the Wiimotes representing the virtual cameras are white.

In addition, the scene graph contains user placeable LEDs that acts as infrared light sources and input to the calibration function. The user is able to decide under which node they should be located. For example, to simulate head worn LEDs the LEDs should be added as children of the animated head node. Using the LEDs for calibration, they should be added as children to the calibration space. The LEDs are not illustrated in Figure 4.4 since their location is controlled by the user.

4.2.3 Calibration Space

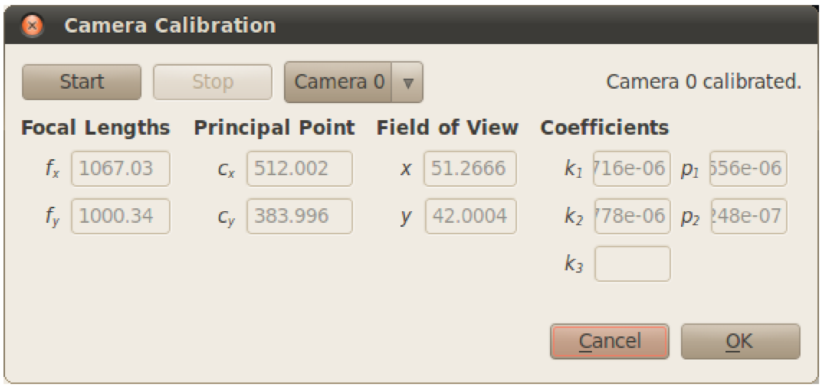
The calibration space is a coordinate system where the calibration rig, calibrated Wiimotes, and the tracked object are specified in. The OpenCV calibration function requires the 3D positions of the calibration rig's LEDs to be specified. The current implementation of OpenCV[1] requires the Z-coordinates of all LEDs to be 0. Therefore, all LEDs must be specified in the XY-plane. OpenCV outputs the extrinsic parameters of the camera relative to the LEDs.

In addition, the 3D positions of the LEDs used for calibration needs only to be specified relative to each other. If the pose of the physical calibration rig is altered, it is only required to move the calibration space accordingly without the need for specifying the position of each individual LED. Figure 4.5 shows the controls for adjusting the calibration space relative to the world space. These controls appear in the left pane of htsim.

4.2.4 Calibration

The intrinsic calibration tool lets the user select which camera to calibrate. When the user starts the calibration, the application captures observed points periodically with 0.5 seconds intervals. When the calibration rig is moved between each observation, the system will end up having different views of the rig. When the system has captured 30 views, it uses the `calibrateCamera` function

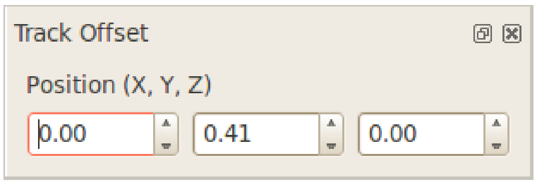
[1]Version 2.0



Camera 0 calibrated.

Focal Lengths		Principal Point		Field of View		Coefficients			
f_x	1067.03	c_x	512.002	x	51.2666	k_1	716e-06	p_1	556e-06
f_y	1000.34	c_y	383.996	y	42.0004	k_2	778e-06	p_2	248e-07
						k_3			

Figure 4.6: The camera calibration tool.



Position (X, Y, Z)

0.00	0.41	0.00

Figure 4.7: Tracking offset. The offset specified here describes the eye position relative to the tracked pose.

in OpenCV to calculate the intrinsic camera parameters. When the calibration finishes, the GUI shows the calibrated principal point, focal length, field of views, and aspect ratio. The user accepts the calibrated intrinsic values by clicking the OK button.

When the extrinsic calibration state is activated in the GUI, the bottom two viewports display two red Wiimote models in the 3D scene indicating where the system believes where the Wiimotes are placed. In extrinsic calibration the `solvePnP` function of OpenCV is called each frame.

4.3 Tracking offset

The tracked poses reported by the tracking system do not always correspond to the eye position. For example, when 6 DoF tracking is used the rear IR source defines the position of the tracked pose. The offset between the eye-position and the tracked pose needs therefore to be specified. Figure 4.7 shows the controls for adjusting the eye position relative to the tracked pose. These controls appear in the left pane of the application.

4.3.1 The ComputerVision class

In HTSIM, the class `ComputerVision` encapsulates all computer vision related tasks. Its interface makes it possible to use the class in a general way, i.e. the class is not aware whether a Wiimote or a virtual camera is using it.

The public functions of the class are listed below.

```
calibrateIntrinsic(cameraId)
calibrateExtrinsic(cameraId, cameraMatrix, distCoeffs)
addKnownPoint(point)
addImagePoint(cameraId, point)
clearPoints()
triangulate()
triangulateHybrid()

signals:
cameraIntrinsics(cameraId, cameraMatrix, distCoeffs, fov);
cameraMatrix(cameraId, cameraMatrix);
triangulatedPoint(pointId, position, error);
```

Qt provides a mechanism called signals and slots. Signals and slots are functions that the Qt SDK treats differently than other functions. The `connect` function in Qt connects signals to specified slots. When a signal-function is called its connected slot-function is called. Calling a signal is in Qt referred to as emitting the signal. The signals and slots mechanism allows for connecting slots to signals transparently of an object.

The signals in the `ComputerVision` class are connected to other parts of `htsim`.

calibrateIntrinsic calibrates the intrinsic parameters of the camera that corresponds to the `cameraId`. The camera parameters are emitted by the `cameraIntrinsics` signal.

calibrateExtrinsic calibrates the extrinsic parameters of the camera that corresponds to the `cameraId`. This function emits the entire matrix, i.e. both the intrinsic and extrinsic parameters with the `cameraMatrix` signal.

addKnownPoint adds a known 3D object point. This function should be called once for each object point. The points added here will be used by `calibrateIntrinsic` and `calibrateExtrinsic`.

addImagePoint adds an observed image point. `cameraId` specifies what camera the image point belongs to. This function should be called once for each observed image point.

clearPoints removes the points added by `addKnownPoint` and `addImagePoint`. This function is typically called at the start of a new frame.

triangulates triangulates the added images points. Successfully triangulated points are emitted by the `triangulatedPoint` signal.

triangulateHybrid needs three points from two camera sources. It triangulates one of the points, the remaining points are calculated as described in Section 3.2.2. All found points are emitted by the `triangulatedPoint` signal.

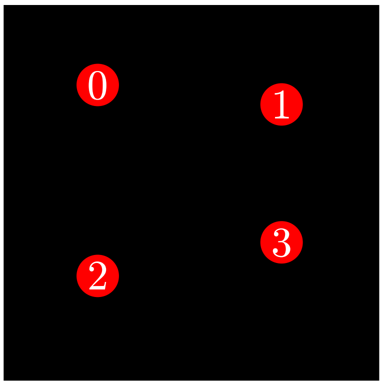


Figure 4.8: Labeling of observed LEDs. The top left LED is labeled 0, the top right is labeled 1, the bottom left 2, and the bottom right 3. This scheme is always applied for four observed points during the calibration process.

Finding Correspondences

The calibration functions require a correct correspondence between the 3D points of the calibration rig and the 2D camera image points. The correspondences are found by sorting the incoming object- and image-points. These points are sorted relative to their position. The order is illustrated in Figure 4.8.

This approach introduces limitations to the orientation of the Wiimotes during calibration. For example, if a Wiimote is held up-side-down the sorted image points will not correspond to the sorted object points.

Triangulation requires correspondences between the point observed by both camera images. The method chosen in [17] is a brute-force approach that triangulates all possible correspondences. The triangulation that results in the lowest error is used. They argue that the brute-force approach is affordable because of the low number of points.

Because of implementations conveniences HTSIM do not use brute-force approach. Instead, observed points are sorted from left to right in both images. This introduces the same orientation limitations as with the calibration rig. In addition, it requires a horizontal layout of the tracked object's LEDs. This is the case with the head mounted light sources in this project.

Chapter 5

Experiments

The experiments in this section focus on finding possible sources of jitter in the system and on application latency. First a jitter measurement experiment was conducted. The goal of this experiment was to see if jitter was introduced when the Wiimote captured a stationary IR source. Thereafter, three experiments measuring the receive rate of IR messages sent from the Wiimote are presented. Latencies that the tracking system introduces is covered in the final experiment.

5.1 Methodology

The computer used in all experiments was equipped with a 64-bits hyper-threaded Intel CPU running at 3.2 GHz and 2 GB of system memory. The Bluetooth interface was a D-Link DBT-122 USB dongle.

All timings were done using the `osg::Timer` class. Unix' `gettimeofday` function is used by this class, which reports timings with microsecond resolution. The application and all its third party libraries used when collecting test results, were compiled with optimization (-O2 flag).

5.2 Jitter

A jitter experiment was conducted by placing an infrared source in front of a stationary Wiimote. The experiment program registered 10 000 samples of the infrared source at various distances and angles relative to the Wiimote.

It was not possible to detect any deviation in the samples for each test.

[17] describes that the pixel positions reported by the Wiimote jitter. When the Wiimote and the IR sources were stationary, the experiment program was unable to detect any jitter.

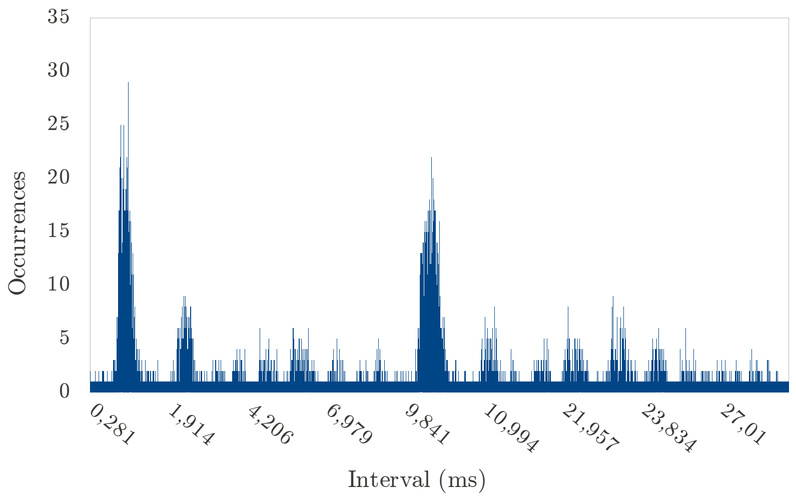


Figure 5.1: Wiimote sampling intervals for the HTSIM application.

5.3 IR Message Rate

5.3.1 IR Message Rate in htsim

The CWIID library uses a callback function for reporting messages from the Wiimote. A message can represent any of the Wiimote's outputs, i.e. the IR camera, the buttons, and the accelerometer. Each time a new message from the Wiimote is ready, the callback function is called by the library. In this test, the time difference between each IR camera message that addresses IR source 0 is logged. 5832 delta times were logged. This experiment obtains its timings from the HTSIM application. The timings for the Wiimote callback function in the HTSIM application is shown in Figure 5.1.

It is possible to see that the delta times are spread from fractions of a millisecond to 30 ms. There are two notable peaks around 1 ms and 10 ms. But overall, the delta times appear with high irregularity.

The callback function is run on a different thread than the main application thread. The main application thread polls the IR source positions 60 times per second. The freshest IR position reported is always used by the main thread, i.e. if multiple IR positions were reported during the 16.7 ms period; the latest of these positions is used by the main thread. Consequently, a locking mechanism is required to pass the IR positions reported in the callback function in a thread safe manner to the main application thread. There is, with other words, a possibility to get conflicting locks. The overhead of using this lock is minimal. There is only one producer and consumer in the system and a relative small data structure is locked for a short period of time. Measurements of the delta times for IR position messages with and without the lock showed no noticeable difference.

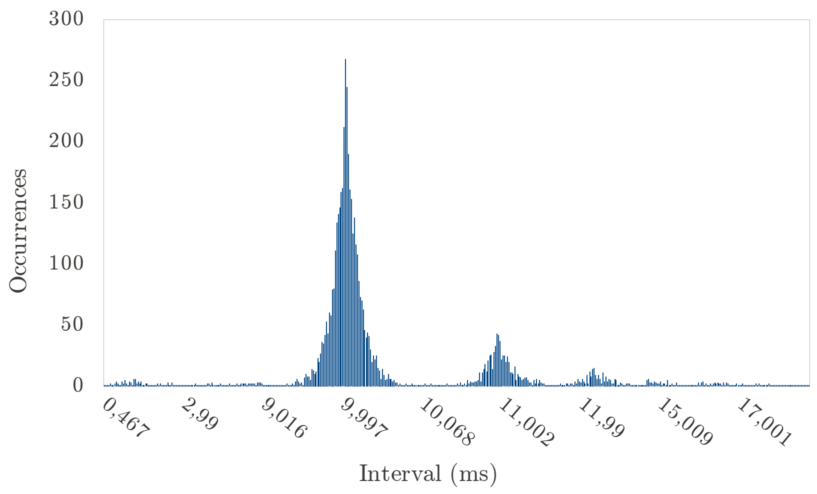
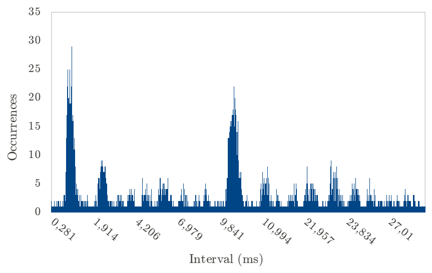


Figure 5.2: Wiimote sampling intervals for standalone experiment application.

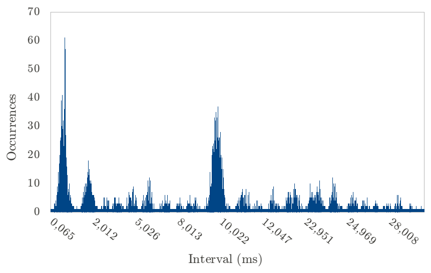
5.3.2 IR Message Rate in standalone experiment application

A new experiment was conducted where a standalone experiment application was used instead of using HTSIM. This application does not draw graphics and does not perform network communication as HTSIM does. Delta times of the Wiimote callback function were collected in same manner as for the HTSIM application. The results, shown in Figure 5.2, reveal more regular delta times than the results from HTSIM.

The reason why the delta times are distributed irregular in the HTSIM case and why they appear more regular in the standalone experiment application was first believed to be caused by the workload of other threads in the application. As mentioned, HTSIM uses much time on tasks such as drawing graphics and updating the GUI. The standalone experiment application has only two threads, the main thread and the Wiimote callback thread. The main thread in the standalone application only uses minimal CPU time. Based on this, the first conclusion was that the system workload created irregular delta times. However, there is one significant difference between the HTSIM experiment and the standalone application experiment that was first overlooked – in the HTSIM experiment two Wiimotes were connected, while in the standalone application only one Wiimote was connected. When the standalone application was configured to connect to two Wiimotes, the pattern of collected delta times was the same as the pattern in the HTSIM experiment. Figure 5.3, compares the results from the HTSIM application with the results from the standalone application that now has two connected Wiimotes. In both cases, the secondary connected Wiimote does not report any data, i.e. it is not aimed to the infrared source.



(a) Results from the HTSIM application. This is a copy of Figure 5.1.



(b) Results from the standalone application that now uses two connected Wiimotes.

Figure 5.3: Comparison between HTSIM and standalone application, both using two connected Wiimotes.

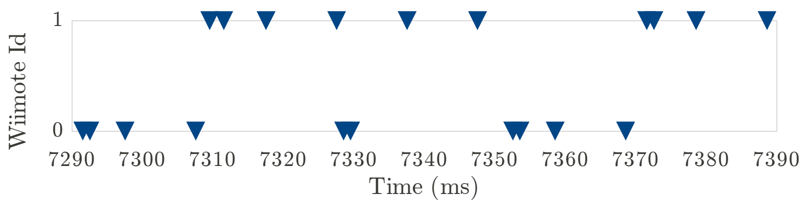


Figure 5.4: The arrival times of IR data for respective Wiimote.

5.3.3 Message Arrival Times for Two Wiimotes

In this experiment two Wiimotes were aimed at a single IR source. The experiment used the standalone experiment application. When the callback function received a new IR message, the arrival times as well as which Wiimote it belonged to was logged. The results are presented in 5.4.

By looking at the pattern of how the IR messages arrive, it seems that the system is only able to handle one Wiimote at a time. The figure shows that the first four messages belong to Wiimote 0. The next four messages belong to Wiimote 1, then two messages for Wiimote 0, two messages for Wiimote 1, four messages for Wiimote 0, and finally four messages for Wiimote 1.

5.3.4 Discussion

The mean with a 95% confidence interval is 10.76 ± 0.23 for the HTSIM experiment and 10.73 ± 0.25 for the standalone experiment that used two Wiimotes. These values are almost identical. The standalone experiment that only used one Wiimote resulted in 10.25 ± 0.05 ms, which is a more stable result. Even if the delta times fluctuate when using two connected Wiimotes, all experiments show a frequency close to the expected 100 Hz (10 ms delta time).

The small delta times (≈ 1 ms) that appear when using two connected Wiimotes can be interpreted as the data is achieved at a higher rate. However, by looking at the order of the delta times, this is probably not the case. Table 5.1 shows an

Sample	**Delta time (ms)**
0	10.017
1	10.020
2	10.138
3	**23.813**
4	**0.969**
5	**5.076**
6	9.965
7	**20.999**
8	**0.963**
9	9.166
10	**27.908**
11	**2.248**
12	**0.567**
13	10.150

Table 5.1: Sequence of delta times sampled from the HTSIM application. Unexpected values in bold.

excerpt from the sampling delta times from one Wiimote when the application has two connected Wiimotes. The expected delta time is 10 ms. It can be seen that sample 3, 7, and 10 are much higher than what is expected. It seems that some part of the system compensates for the high delta times by invoking the callback function at a higher rate. For example, the relative small delta times at samples 4 and 5, succeed the high delta time at sample 3. The average of sample 3-5 is 9.95 ms, which is close to the expected 10 ms. The average of sample 8-9 and 10-12 are 10.98 and 10.24 respectively. This explains the left most peaks in Figure 5.1 and 5.3 – several small delta times are needed for compensating one large delta time.

Based on result presented in Figure 5.4, it seems that the system dedicates recurring time slots for each Wiimote. Furthermore, it seems that somewhere in the system there is a queue for each Wiimote that stores the registered IR data. Each of these queues seems to be emptied during a time slot. This can explain why the messages are registered at a higher rate in the beginning of each time slot. These messages were probably registered by the Wiimote during the other Wiimote's time slot. Therefore, they are already in the queue when the time slot starts. Removing old messages from the queue is probably performed at a faster rate than 100 Hz.

We have not investigated where in the system these possible queues are stored. Neither have we investigated where in the system the sizes of the time slots are specified or restricted.

Consequences

The main thread of HTSIM polls the Wiimote callback thread for new data each 16.7 ms . When delta times between each Wiimote callback call are higher than 16.7 ms, it forces the main thread to use old data. The practical consequences

	Mean	**Error**
`addWiimoteDataToComputerVision`	0.051	±0.002
`triangulate`	0.190	±0.013
`tracking6DoF`	0.004	±0.000

Table 5.2: 95% confidence intervals for selected functions. All units are ms.

for the HTSIM application is that sometimes the same IR positions are used for as much as three subsequent frames (when delta time > 33 ms).

When triangulating a moving IR source, it important to use IR positions from both Wiimotes that were captured at the same time. If Wiimote 0 captures a moving IR source at time t_0 and Wiimote 1 captures the same IR source at time t_1, the triangulated IR source will have some error.

5.4 Tracking latency

The main thread of the application performs tracking at 60 Hz. The reason why 60 Hz is chosen is because the application is synchronized with the refresh rate of the screen. Tracking includes: (i) fetching the IR positions from the main thread; (ii) triangulation; and (iii) tracking. The timings of these respective functions are presented as 95% confidence intervals in Table 5.2.

Chapter 6

Discussion

6.1 System Latency

This section details out the sources of latency in the system. The following list describes chronological different parts of the system that contributes for the total latency. Every part is then discussed in more detail.

1. Wiimote captures an IR image
2. Wiimote analyzes four brightest IR sources
3. Wiimote sends the position of the IR sources via Bluetooth to the tracking system.
4. The tracking system polls the received data at 60 Hz.
5. The tracking system performs triangulation and tracking.
6. The tracking system sends the tracked pose to a Shout server.
7. The Shout server distributes the camera events to all tiles.
8. The tile displays the effect of the camera event in its next frame.

The Wiimote specifications found at [22] states that the Wiimote has a 24 MHz pixel clock. This means that the Wiimote is able to read 24 million pixels per second. The 128×96 monochrome resolution contains 12,288 pixels. 24 million pixels per second results in $24000000/12288 = 1953$ images per second. The time it takes for one image to be captured, is therefore $1/1953 \approx 0.512$ ms. This is just the theoretical time, based on the pixel clock frequency. Other parts in the Wiimote might make the capturing slower.

The Wiimote analyzes the four brightest points of the image. The latency this adds to the system is unknown.

The latency Bluetooth communication adds has not been measured. [1] lists throughputs with various packet sizes for the Bluetooth protocol. The highest

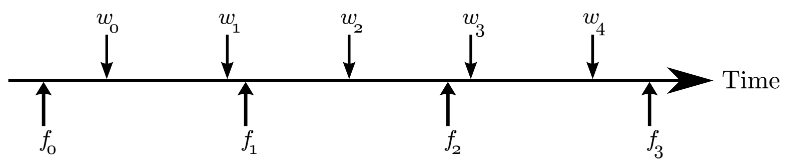


Figure 6.1: 100 Hz compared to 60 Hz. w_i indicates when a new IR message is received. f_i indicates when IR messages are polled.

throughput is 723.2 kbps and the lowest 36.3 kbps. However, the throughput does not reveal anything about the latency. It is unknown if the Wiimote, or the host it communicate with, buffers several packets before they become visible in the application layer.

The time it takes from an IR source is registered until it is available at the host can be roughly measured if another IR camera, with known latency, is used. If an IR LED that both cameras sees is turned on, both cameras will be exposed by the light at the exactly same time. The host will receive the IR data from the cameras at different times, t_w for the Wiimote and t_k for the camera with known latency. It is now possible to calculate the latency of the Wiimote by looking at the delta time between two times, i.e. $t_w - t_k$.

The time it takes to perform triangulation and tracking was measured in Section 5.4 to be approximately 0.25 ms. The tracking system polls the Wiimote thread at 60 Hz for new IR data. When the system uses one connected Wiimote, IR data is received regularly at 100 Hz. This scenario is demonstrated in Figure 6.1. Here, w_i indicates a received message and f_i indicates when the messages are polled. w_0 and w_3 are not used since w_1 and w_4 succeed them before the polling takes place. Polling a 100 Hz message source at 60 Hz, results in 40% loss of messages. The oldest message is w_2 when it is polled at f_2. This polling scheme can delay a message up to 3/5 of one polling interval, i.e. a received message is up to 10 ms old before it is used.

However, two connected Wiimotes can result in even older messages to be used. This is the result of time slot shifting described in the experiment chapter. In addition, [17] suggests a 4-sample moving average filter for removing jitter. A filter will further increase the system latency.

Latency the Shout event system introduces is presented in [21]. Based on these results, sending the tracking event from the tracking system to the front-end takes 0.13 ms on average. Furthermore, it takes 0.45 ms for the front-end to distribute an event to all tiles.

The tiles are displaying graphics at 60 Hz. The consequence is that when a tile receives the event sent from the front-end, the result will not be visible immediately. As mentioned in Section 4.1.2, the preparation of each frame begins at 16.7 ms intervals. The prepared frame is displayed at the start of the next 16.7 ms period. The time it takes for a tile to display the effects of a received Shout event is between 1 and 2 frames (16.7 – 33.3 ms). As illustrated in Figure 6.2, events that are received between frame f_0 and f_1 are displayed at f_2.

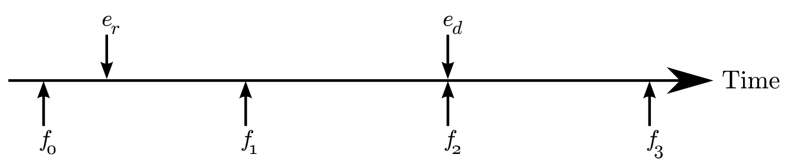


Figure 6.2: Latency for a tile receiving a Shout event. f_i indicates the start of new frames. e_r is the time when the Shout event is received. e_d is the time when the effect of the Shout event is displayed.

The exact total system latency is unknown since there are parts of the system that contributes with unknown latency. However, it can be said something about the minimum latency of the system. Referring to the number list in the beginning of this section, (1) has minimum latency of 0.512 ms, (2) and (3) are unknown, (4) has a maximum latency of 10 ms, (5) has a latency of 0.25 ms, (6) 0.13 ms, (7) 0.45 ms, (8) has a minimum latency of 16.7 ms and a maximum latency of 33.3 ms.

The worst case latency of the system is,

$$0.51 + 10 + 0.25 + 0.13 + 0.45 + 33.3 + x = 44.64 + x \tag{6.1}$$

ms , where x is the unknown latency.

The best case latency of the system is,

$$0.51 + 0 + 0.25 + 0.13 + 0.45 + 16.7 + x = 18.04 + x \tag{6.2}$$

ms.

6.2 Scalability

This project has implemented head tracking by using two Wiimotes. To increase the usable head tracking area, more controls can be added. However, the software made in the project only supports two Wiimotes. Supporting more than two Wiimotes needs some extensions to the software, especially the GUI part of HTSIM.

The time slots mentioned in 5.3.4 indicates more latency with two Wiimotes. If the latency increases linearly when the number of Wiimotes increases, scalability can be an issue. Using multiple Bluetooth interfaces on the tracking system computer, might solve this problem. However, this is not tested.

6.3 Hardware

The plastic material of the helmet made IR sources reflect in the helmet. The Wiimote reported these reflections as IR sources. This problem was most noticeable for the IR clusters located on the sides of the helmet. To prevent the reflections, the helmet was sprayed with black matte spray, which reduced the reflections.

However, the Wiimotes still detect some reflections. This shows that the helmet construction is not well suited for this type of head tracking. A cloth material would have reduced the distinct reflections. A head worn device that does not cover the head in the way that the helmet does is maybe an even better solution.

6.4 Calibration

Calibrating the Wiimotes with the calibration rig used in the project gave inaccurate results. To improve the accuracy, the intrinsic matrix presented i [6] was used instead of the calibrated one. Also, placing the Wiimotes manually instead of calibrating their pose showed to give more accurate results.

The calibration rig used in [17] uses 35 IR sources when they calibrate theirs controls. This is contrast to the four IR sources used by this project's calibration rig. In [17], the 35 IR sources are exposed to the Wiimote by four at the time.

6.5 Future work

Developing a standalone Wiimote tracking server, that does not rely on a GUI, is the next natural step to take. This will allow the tracking system to run permanently on a Bluetooth equipped server machine. Running this tracking server on the display wall's front-end makes the computer that runs the tracking system today redundant. This also shortens the overall latency in the system, since network communication is avoided between the tracking system and the front-end. However, this will not make HTSIM redundant. A GUI simplifies the calibration process and the tracking configuration. To manually type a Wiimotes pose in a configuration file is much more cumbersome and error-prone than positioning the Wiimote visually in a GUI. HTSIM could act a tool for creating relevant system configurations that the tracker server uses.

The system implemented in this project only supports Wiimotes as its camera sources (except for the virtual cameras). Extending HTSIM, so that any camera equipped with an infrared pass filter could be used, is also a possible solution.

More applications can be made. Since it is possible in our system to detect the head orientation, head gestures like nodding and shaking the head can tracked. Future applications could also be to select object on the display wall by facing them.

Chapter 7

Conclusion

In this thesis head tracking was implemented with the help of the HTSIM application. HTSIM was a great help for testing and to get all mathematics and algorithms concerning head tracking correct. The application also showed to be a good tool for configuring the head tracking for use with the physical environment. OSGDWL demonstrated the head tracking system by presenting a user dependent view frustum the display wall.

The relative small area that the head tracking system covers is not sufficient to allow a user to move around freely in front of the display wall. More experiments, with other type of IR worn devices or with more than two Wiimotes, could have been made to investigate more around this issue.

Timing experiments indicated that scaling the system could contribute to the latency.

Appendix A

User guide of htsim

This chapter presents a brief user guide for the HTSIM application. All descriptions are based on the real-mode of the application, i.e. when using Wiimotes as cameras.

A.1 Identifying Wiimotes

When two Wiimotes are used for the first time with HTSIM they need to be identified.

1. Select `Tools/Settings...` from the menu. This will open a dialog where two MAC addresses can be specified.
2. Pressing one of the ellipsis buttons will open the Bluetooth scanning dialog.
3. Enable one of the Wiimotes by pressing the 1- and 2-button on the Wiimote simultaneously.
4. Press the refresh button in the dialog. This will start to scan for all Bluetooth devices in your proximity.
5. Select the Bluetooth device that appears as "Nintendo".
6. Press OK.
7. Go to Point 2 and repeat the procedure for the second Wiimote.

A.2 Connect Wiimotes

Connecting to Wiimotes already identified is done in the following way

1. On both Wiimotes, Press the 1- and 2-button simultaneously.

2. Within 20 seconds, press the Connect button in the toolbar.
3. A progress bar indicates if the Wiimotes connects successfully.

A.3 Creating Calibration Patterns

Calibration requires the virtual LEDs in HTSIM to be setup with the same measurements as the physical calibration rig. This describes how it is done.

1. In the Ir Diodes dialog, located in the left pane of the GUI, press the ellipsis button. A dialog is opened that enable you to select a node to which the virtual diodes will placed relative to.
2. Select the "Calibration Space" node and click OK.
3. Press the "Presets..." button. This will open a dialog that makes it possible to define the properties of physical calibration rig.
4. Select the size of the IR LED grid, the offset from the calibration space origin, and the spacing between each IR LED.
5. Pressing OK will now create the specified calibration pattern.

A.4 Calibrating the Intrinsic Parameters of the Wiimote

This describes how to calibrate the intrinsic parameters of a Wiimote.

1. Connect the Wiimote's to the system.
2. Create calibration pattern that matches the physical rig and that is relative to "Calibration Space".
3. Start the intrinsic calibration tool, by pressing the I button.
4. Select which of the Wiimotes that should be calibrated (Camera 0 or Camera 1).
5. Place the Wiimote stationary on a desk. (Make sure that IR LEDs are not reflected in the surface of the desk).
6. Make sure that the physical calibration rig is turned on, and hold it in front of the Wiimote.
7. Press "Start". HTSIM will now start capturing the IR LEDs with 0.5 s intervals. Make sure the Wiimote observes the calibration rig from different poses by moving around the calibration rig while capturing.
8. When the calibration has captured 30 images, it will calculate the intrinsic parameters of the camera.
9. Press OK to store the parameters.

A.5 Calibrating the Extrinsic Parameters of the Wiimotes

This describes how to calibrate the extrinsic parameters of the Wiimotes, i.e. the pose of the Wiimotes.

1. Connect the Wiimote's to the system.
2. Create calibration pattern that matches the physical rig and that is relative to "Calibration Space".
3. Enable the rendering of the calibration space axes by pressing the button.
4. Move the calibration so that the virtual rig matches the physical rig's pose. This is done with the Calibration Space dialog found in the left pane of the GUI. For example, if the calibration rig is positioned one meter in front of the center of the canvas, the calibration space should be adjusted so that the virtual calibration rig matches this position in the virtual scene.
5. Place both Wiimotes so that they observe the calibration rig and press **E** button.
6. HTSIM will continuously update the extrinsic parameters until the **E** is pressed again.

A.6 Define the Extrinsic Parameters of the Wiimotes manually

Since the extrinsic calibration is not always accurate, defining the extrinsic parameters manually is possible. The position of the calibration space is irrelevant when defining cameras manually.

1. Give the virtual cameras a pose that matches the Wiimotes. This can be done in two ways:
   (a) Click one of the top viewports, and edit the pose of the virtual camera in the left pane, or...
   (b) Enter virtual mode, and move the virtual camera by dragging with the mouse in the top viewports.
2. When the pose of both virtual cameras matches the Wiimotes, select `Tools/Match Wiimotes to Viewports`.

A.7 Object Tracking

This describes how to track an object. See Section 3.2.2 for more information about the tracking modes.

1. Make sure that the camera intrinsic and extrinsic parameters are correct.
2. Connect the Wiimotes.
3. And press the Triangulate button, ▲. IR sources will now be tracked in 3D space. The tracked IR sources appear green in the viewports of HTSIM.
4. Object tracking is enabled with one of the following modes:
   (a) Enables position tracking. This requires the cameras to see one common IR source.
   (b) Enables 6 DoF tracking. This requires the cameras to see three common IR sources.
   (c) Enables 6 DoF hybrid tracking. This requires the cameras to see two IR sources each, where one of these IR sources is a common.

A.8 Send Tracked Poses as Shout Events

This describes how to send tracked poses to the OSGDWL application running in server mode. The current implementation of HTSIM has the Shout server's address hard coded to `rocksvv`.

1. Start OSGDWL in one of its server modes on the front-end (`rocksvv`). See Section 3.5.2.
2. Setup HTSIM to track an in object with preferred tracking mode.
3. Use the shout button, , to connect and send the tracked poses to the Shout server. Pressing the button once again stops sending Shout events and disconnects from the server.

Appendix B

Configuration System

A configuration system called CFGG was developed during this project. CFGG stands for configuration generator. The purpose of this system is to make it convenient to add new parameters to a configuration file, as well as reading and writing the configuration file.

A configuration file consists of multiple key-value pairs. For example, `Name Alice`, `Age 27`. The keys `Name` and `Age` refer to the values `Alice` and `27` respectively. If the two values are supposed to be handled as a string and an integer in the application, some data informing about this is required.

One method is to specify the data type in front of the key name in the configuration file, e.g. `string Name Alice`. The mapping between a key and its data value can also be hard coded in the application or be defined in a different file that describes the data types for all keys. In CFGG, a configuration description file is used that defines data types/key pairs. For example, `string Name`, `int Age`. The configuration description file has `.cfgg` as file name extension. The configuration description is used by the CFGG program to generate C++ code.

The generated code parses a configuration file by taking care of the data types for each parameter. The values from a configuration file are accessible from the interface the generated code provides. An application interested in the `Age` value, specified in a configuration file, calls the `age()` function from the generated code interface. The `age()` function in this case will return an integer.

It is also possible to modify configuration parameters from the code. This is done by a simple assignment, `age() = 28`. When the write function of the configuration system is called, all modified values will be saved into the configuration file.

B.1 The Configuration Description File Format

CFGG supports the built-in data types, std::string, char, int, float, and double. It is possible to extend `cfgg` with custom data types. At the beginning of the .cfgg file, it is possible to define additional header files that should be included in the generated code. Each header that should be included should be specified on individual lines that are prefixed with the letter 'I' followed by a white space. For example, `I <osg/Vec3d>` will make sure that the `osg/Vec3d` header is included in the generated code.

It is the user's responsibility to handle the parsing of custom data types. This is done inside the switch-clause inside the ConfigReader's constructor. In the configuration description file, lines that begin with the letter 'T' followed by a data type and a label, e.g. `T osg::Vec3d VEC3D`, describe what label belongs to what data type.

B.2 Reading and Writing Custom Data Types

Parsing an osg::Vec3d is done as follows,

```
case VEC3D:
{
    osg::Vec3d v;
    ss >> v.x() >> v.y() >> v.z();

    osg::Vec3d* paramDataStr =
        static_cast<osg::Vec3d*>(m_parameterRegister[key].data);
    *paramDataStr = v;
}
break;
```

`ss` is the file stream of the configuration file. `m_parameterRegister[key].data` is a void pointer to the location where the object should be stored.

Writing custom data types to the configuration file requires the same label to be added to the switch clause in ConfigWriter's write function. The write implementation an osg::Vec3d is presented here,

```
case VEC3D:
{
    osg::Vec3d& v = *static_cast<osg::Vec3d*>(parameter.data);
    file << v.x() << " " << v.y() << " " << v.z();
}
break;
```

`parameter.data` is a void pointer to where the data is stored. `file` is the output file stream.

Bibliography

[1] P. Bhagwat. Bluetooth: technology for short-range wireless apps. *Internet Computing, IEEE*, 5(3):96 –103, may/jun 2001.

[2] A. Birgisson and B. E. Kristjánsson. Immersion through head-tracking. Technical report, Reykjavík University, 2008.

[3] G. Bradski and A. Kaehler. *Learning Opencv*. Twayne Publishers, Boston, first edition, September 2008.

[4] GlovePie. ”`http://glovepie.org/`”, 05 2010.

[5] R. Hartley. *Multiple View Geometry in Computer Vision*. Twayne Publishers, Boston, 2003.

[6] S. Hay, J. Newman, and R. Harle. Optical tracking using commodity hardware. In *ISMAR '08: Proceedings of the 7th IEEE/ACM International Symposium on Mixed and Augmented Reality*, pages 159–160, Washington, DC, USA, 2008. IEEE Computer Society.

[7] J. Heikkila and O. Silven. A four-step camera calibration procedure with implicit image correction. In *Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on*, pages 1106 –1112, jun 1997.

[8] J. Lee. Hacking the nintendo wii remote. *Pervasive Computing, IEEE*, 7(3):39 –45, july-sept. 2008.

[9] P. Martz. *OpenSceneGraph Quick Start Guide*. Skew Matrix Software LLC, 2007.

[10] Microsoft. Project natal. ”`http://www.xbox.com/en-us/live/projectnatal/`”, 05 2010.

[11] NaturalPoint. Trackir. ”`http://www.naturalpoint.com/trackir/`”, 04 2010.

[12] OpenCV. ”`http://opencv.willowgarage.com`”, 05 2010.

[13] OpenSceneGraph. ”`http://www.openscenegraph.org/`”, 05 2010.

[14] Qt. ”`http://qt.nokia.com/`”, 05 2010.

[15] RationalCraft. Winscape. ”`http://www.rationalcraft.com/Winscape.html`”, 04 2010.

[16] Rocks. Rocks clusters. ”`http://www.rocksclusters.org/`”, 05 2010.

[17] D. Scherfgen and R. Herpers. 3d tracking using multiple nintendo wii remotes: a simple consumer hardware tracking approach. In *Future Play '09: Proceedings of the 2009 Conference on Future Play on @ GDC Canada*, pages 31–32, New York, NY, USA, 2009. ACM.

[18] D. Shreiner and T. K. O. A. W. Group. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1 (7th Edition)*. Addison-Wesley Professional, 2009.

[19] L. D. Smith. Cwiid. "`http://abstrakraft.org/cwiid/`", 04 2010.

[20] Sony. Playstation move. "`http://us.playstation.com/ps3/playstation-move/`", 05 2010.

[21] D. Stødle. *Device-Free Interaction and Cross-Platform Pixel Based Output to Display Walls.* PhD dissertation, University of Tromsø, Department of Computer Science, June 2009.

[22] WiiBrew. Wiibrew wiki. "`http://wiibrew.org`", 04 2010.

[23] Wiiuse. "`http://http://www.wiiuse.net/`", 05 2010.