Development of a Bluetooth controller for mobile VR headsets

Tommy Holmberg
Abstract

Development of a Bluetooth controller for mobile VR headsets

Tommy Holmberg

Mobile virtual reality (VR) headsets have been becoming more and more popular. However, the cheapest headsets do not come with any controllers and the ones that do include controllers only uses sensors for rotation, not translational movement. This thesis project aims to develop a prototype of a Bluetooth connected controller for the mobile VR headsets. The controller is based on a MetaMotionC board produced by mbientlab Inc., which comes with Bluetooth Low Energy (BLE), an ARM M4 microcontroller, an miniature inertial measurement unit (IMU) sensor (containing a 3-axis accelerometer, a 3-axis gyroscope, and a 3-axis magnetometer and a barometer), a thermometer and other sensors. The only sensors used in this project are the accelerometer, gyroscope, and magnetometer.

As a finished prototype, the MetaMotionC is placed on a glove together with five Aruco markers; a 3D model of a hand intended to use as an avatar of the glove was made with Blender and MakeHuman; and a VR room to use the controller with was created in Unity. The 3D hand responds to rotational and translational movements via Bluetooth connection to the IMU sensor on the MetaMotionC. The smartphone camera is used to detect the glove's position with Aruco markers, and the 3D hand is moved to a corresponding location in the VR room. The OpenCV library is used for image processing. The sensor data is filtered with low-pass, median, and thresholding to improve the measurement accuracy. Zero velocity update is used to reset the drift of the integrated accelerations. To reduce the integration error, Romberg's method with a floating window is implemented in Matlab. However, it did not reduce the error enough to make a difference. Thus, the result was unreliable.
Sammanfattning


Länkade handkontroller

For att interagera med VR-världen finns en mängd olika handkontroller. Till de dyrare systemen, som kopplas till en kraftfull spel dator, har handkontroller med rörelsesensorer kommit. Men till de billigare systemen som en smartphone används i, finns bara kontroller som påminner om tv-spelskontroller, så kallade gamepads.


För att kunna testa kontrollen kommer även ett VR testrum att utvecklas i Unity. En 3D hand kommer att användas som avatar för handsken.
Acknowledgments

Special thanks to my subject reader, Ping Wu, who patiently reviewed the report several times and helped improve the quality of it significantly.

I am grateful to the people at Syntronic in Gävle, who let me do this thesis work in their facilities. Thanks to Håkan Sjörling and Kent Zetterberg for encouragement and valuable feedback during the project.
# Contents

1 Introduction .................................................. 7
   1.1 Background ............................................... 7
   1.2 Purpose and project specifications .................... 9
   1.3 Tasks ..................................................... 10
   1.4 Outline .................................................. 11

2 Working principles and theory ............................... 12
   2.1 Overview of the system .................................. 12
   2.2 MetaMotionC Board ....................................... 13
   2.3 Multiprotocol system-on-chip nRF52832 ................. 14
   2.4 Motion sensors (Inertial measurement unit) .......... 14
      2.4.1 Accelerometer ...................................... 14
         2.4.1.1 Sensor construction and noise sources .......... 14
         2.4.1.2 Mathematical operations ...................... 15
         2.4.1.3 Translational acceleration ................. 16
      2.4.2 Gyroscope .......................................... 17
      2.4.3 Sensor fusion ..................................... 17
         2.4.3.1 Quaternions .................................... 17
      2.5 Discrete integration .................................. 22
         2.5.1 Trapezoid rule .................................... 22
         2.5.2 Romberg’s method ................................ 23
      2.6 Filters ................................................ 24
         2.6.1 Low-pass .......................................... 24
         2.6.2 Median ............................................ 24
         2.6.3 Zero velocity update ................................ 24
   2.7 Image capture and analysis ......................... 25
      2.7.1 Basics about images ................................ 25
      2.7.2 Camera .............................................. 28
      2.7.3 OpenCV ............................................... 29
         2.7.3.1 Fiducial markers ............................... 30
         2.7.3.2 Aruco markers ................................... 30
         2.7.3.3 Whycon markers .............................. 32
      2.8 Communications: Bluetooth and Internet .......... 35
         2.8.1 Bluetooth Low Energy (BLE) .................... 35
         2.8.2 Internet protocol ................................ 35

3 Implementation .............................................. 36
   3.1 Development tools ..................................... 36
      3.1.1 Unity ............................................... 36
3.1.2 Blender .............................................................. 37
3.1.3 MakeHuman ...................................................... 37

3.2 Methods ............................................................. 38

3.2.1 Testing room .................................................. 38
3.2.2 User interface ................................................ 39
3.2.3 Player .......................................................... 39
3.2.4 Hand model ................................................... 40
3.2.5 Coordinate indicators ...................................... 41
3.2.6 Coordinate system .......................................... 42

3.3 Movement .......................................................... 45

3.3.1 UDP Client ..................................................... 45
3.3.2 Sensor data collector ....................................... 46
3.3.3 File Management ............................................. 47
3.3.4 Filters ........................................................ 47
3.3.5 Acceleration to position .................................. 48
3.3.6 Implementation of movement in Unity ............... 48

3.4 Pattern recognition .............................................. 52

3.4.1 Camera ........................................................ 52
3.4.2 Aruco markers .............................................. 52
3.4.3 Whycon markers .......................................... 54

3.5 Zero velocity update ............................................ 57

3.6 Implementation and debugging .............................. 57

3.6.1 Android ...................................................... 57
3.6.2 Matlab and Scilabs ....................................... 62
3.6.3 Processing .................................................. 62

4 Results and discussions ......................................... 63

4.1 Prototype ........................................................ 63
4.2 Romberg’s algorithm with a floating window .......... 64
4.3 Linear acceleration .......................................... 64
4.4 Translational acceleration .................................. 73
4.5 Fiducial markers .............................................. 73
4.6 Sensor fusion .................................................. 73
4.7 Bluetooth ........................................................ 74
4.8 Pattern detection .............................................. 74
4.9 Cost .............................................................. 75

5 Conclusions and suggestions for future work .......... 76

5.1 Conclusions ...................................................... 76
5.2 Future work ..................................................... 77
1 Introduction

1.1 Background

Virtual reality (VR) systems have a growing market. A variety of headsets are available, ranging from high-end products (with high performance and high prices) to low-end ones (with low performance and low prices). On the high-end side, for instance, there are HTC VIVE, Oculus Rift, and Samsung Odyssey that are the leaders in their field. These high-end headsets are equipped with motion-tracked handheld controllers for users to interact with the environment. The headsets contain specialized hardware and are tethered, which means that they will be connected to a gaming PC through a cable. The PC runs a VR world. The headset relays information about the headsets orientation and keeps track of hand controllers. It is called mixed reality when images from the real world are used to map objects into the VR world.

HTC VIVE uses two base stations called ”lighthouses” which are placed in opposite corners of a room, and can be used in a room up to 4.6x4.6m. The lighthouse sends out a ray of invisible light that hits the hand controller and is reflected on to the headsets light sensor. The HTC VIVE has 37 IR sensors that are used to accurately track the controller.

Oculus Rift, on the other hand, has IR LEDs strategically placed on the headset and controllers. An external camera is used to track both the headset orientation and the controllers. This camera can only track an area that is 1.5x1.5m accurately.

Samsung Odyssey uses a different approach. Instead of relying on external units that are placed in a room, they use two cameras mounted on the headset. The cameras resulting field of view of is 110°. To accomplish this wide FOV they are pointed slightly outward to the sides. However, it cannot track the controllers when they are moved behind the headset. The controllers have inertial measurement units (IMUs) like the previously described headsets, so the rotation of the controller can still be tracked.

The low-end VR headsets are often made by the leading smartphone manufacturers. Among them are Samsung Gear VR, Google Daydream View, and Cardbox VR. The main differences between them is appearance and materials. Some of them, like the Daydream and Gear VR, also have controls. The controllers contain an IMU to sense rotation, some buttons, and a touchpad to control menus and cursors. But they have no way to track motion other than rotation. These headsets use a smartphone as the main computer. This means that these types of headsets are not tethered, which allows for more freedom of movement. Games or movies can be installed on a smartphone,
which is mounted on the front of the headset. The display of the smartphone is magnified through binocular-like lenses in order to fill a large portion of the user’s field of view.

A smartphone headset is shown in Figure 1.1. These headsets are opened in the front to expose the smartphone’s camera. This could be used to detect objects in front of the camera, in a similar fashion that Samsung Odyssey uses.

All of the smartphone manufacturers are trying to add special features to their headsets. For example, the Gear VR can be tapped to interact with the object that the user is looking at. The accelerometer in the smartphone is used to detect the tap.

Low-end VR headsets were a popular Christmas present in 2016, so they have become increasingly common on the market. However, there is still no cheap controller for translational motion to accompany the headset. The controllers used with Daydream and Gear VR do not utilize the depth of the VR world, they can only sense rotation. Perhaps this is the reason why VR has not made its huge breakthrough yet, since the high-end VR systems

Figure 1.1: Low-end VR headset (smartphone VR headset) that uses a smartphone as the main computer [1].
cost several hundreds of dollars and a high-end PC is needed to run the VR world.

1.2 Purpose and project specifications

The goal of this project is to develop a prototype of a controller for the low-end VR headsets. The purpose of this controller is to be able to map hand movements in the real world into the virtual reality (VR) world. The controller is made of a glove on which system consisting of a microcontroller, an IMU and a Bluetooth communication module is mounted (Figure 1.2). It allows for movement in the depth dimension in conjunction with horizontal and vertical movements so as to increase the immersion of the VR experience. The controller needs to be cheap since it is aimed for the low-end headsets. The Bluetooth controller is a MetamotionC board that is intended for motion, health, fitness and gaming, and consists of a microcontroller, an accelerometer, gyroscope, and magnetometer to detect rotations. The accelerometer is used to roughly determine movements, and the smartphone’s camera in the VR headset is used to make a precise position estimation. The schematic of the controller is illustrated in Figure 1.2, where the user wearing the controller and puts it in front of the VR headset so that the smartphone camera can see it.

![Figure 1.2: VR headset using the smartphone camera and a sensor unit to track motion.](image)
1.3 Tasks

- Literature study
  - Study sensors.
  - Study microcontrollers and sensor units.
  - Study signal processing methods and numerical methods.
  - Write a report for each study.

- VR development
  - A simple VR room.
  - A 3D model of a hand.
  - Use sensor data to move the hand in the VR room.
  - Implement IP camera viewer.

- Filters
  - Write C# programs for digital filters (low-pass, median and threshold).
  - Implement pattern recognition with OpenCV.

- Android development
  - Write an Android app to communicate with the MetaMotionC.
  - Create an activity to override Unity’s Android activity.

- Processing
  - Write a graphing program to visualize signals from the MetaMotionC.

- Matlab
  - Write an implementation of Romberg’s method and test performance.
  - Write programs to process data from the Bluetooth controller and decide filter properties experimentally.

- Prototype
  - Attach markers and MetaMotionC to a glove.
1.4 Outline

The report will be outlined as follows. Section two deals with the detailed theory of all the parts and processes used in this project. Mathematical equations that describe each process that was used in the project will be presented. The third section is concerned with the implementation. Each part of the implemented software is described and pseudo code is used when feasible. Then the results of the measurements will be presented in section four. The section contains graphs and tables with comparisons, and the results are subsequently discussed. The last part, section five, contains conclusions regarding the goals of the project and suggestions for future work.
2 Working principles and theory

2.1 Overview of the system

The schematic drawing of the Bluetooth controller is shown in Figure 1.2. The controller is used with a VR headset. In order to achieve an accurate position of the controller, two separate processes will be run (Figure 2.1). The sensor process on the left side, and the marker detection on the right side. The process on the left is highly inaccurate at estimating the hand position. When the accelerations that the sensors measure are integrated to get velocity, and then again to get position, it will lose its precision rapidly. However, the camera can not take a sharp image when the hand is moving, so it is only the sensor data measurements that can be used to estimate the position. The process on the right is highly accurate. When the hand stands still, the pattern recognition algorithm will be able to produce a precise position of the hand. When this happens, the velocity integral can be reset. The velocity integral needs to be reset often since the accuracy is reduced rapidly, which will be explained in further detail later in this report.

![Flowchart](image)

**Figure 2.1:** Flowchart that shows an overview of the program. This loop is executed each frame of the game.
2.2 MetaMotionC Board

MetaMotionC from mbientlab is a coin-sized sensor unit. It is meant to be used for prototyping. It measures approximately 25 mm in diameter. The main component is the SoC nRF52832, which has an ARM Cortex-M4F MCU and a BLE module [2]. All the onboard components can be seen in Figure 2.2. The most important for this project is the nRF52832 and the 9-Axis IMU. The 9-Axis sensor fusion consists of a Bosch BMI160 6 Axis Accelerometer/-Gyroscope and a Bosch BMM150 3-Axis Magnetometer. Bosch also provides a sensor fusion software called Bosch Sensortec FusionLib, which generates absolute orientations from the two sensor units. The other sensors on the MetaMotionC are irrelevant for this project. A price chart containing the cost of MetaMotionC and the individual parts important to this project can be seen in Table 2.1. The components used in this project cost approximately $6.3 if they were to be purchased individually.

![Figure 2.2: MetaMotionC with the most important components mapped out.](image)

<table>
<thead>
<tr>
<th>Component</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>MetaMotionC</td>
<td>$80</td>
</tr>
<tr>
<td>nRF52832</td>
<td>$3</td>
</tr>
<tr>
<td>BMI160</td>
<td>$2.5</td>
</tr>
<tr>
<td>BMM150</td>
<td>$0.8</td>
</tr>
</tbody>
</table>

Table 2.1: Cost of components at the time of writing. Prices are given in USD.
2.3 Multiprotocol system-on-chip nRF52832

The SoC nRF52832 consists of an ARM Cortex-M4 32-bit 64 MHz CPU and a 2.4 GHz transceiver that supports Bluetooth low energy. The antenna can be used for other protocols as well, such as ANT of other proprietary protocols. When it is used for Bluetooth, its sensitivity is at -96 dBm. The sensitivity level is how strong the signal that the transceiver needs to receive to reach a bit error rate of $10^{-3}$. The product specification claims to support 2 Mbps transfer rate [3], which was introduced with Bluetooth version 5. Before version 5, only 1 Mbps was supported. That means that if the device that connects to the nRF52832 has a Bluetooth version lower than 5, it will use the lower transfer rate. The nRF52832 is also equipped with I2C and a 200 ksps 12-bit ADC, and several other components which are less important to this project.

2.4 Motion sensors (Inertial measurement unit)

2.4.1 Accelerometer

2.4.1.1 Sensor construction and noise sources

A micro electro-mechanical system (MEMS) accelerometer is a microscopic capacitor that changes capacitance when it is subjected to movement. A capacitor in its simplest form is two conductive plates, called electrodes, that are separated by an isolator. The MEMS accelerometer can be seen as two combs, where the comb’s teeth are the capacitor electrodes. One of the combs is a static structure, the other is attached on one side and has a proof mass on the other side. When the structure is moved, the proof mass’ inertia will make the teeth of the two combs separate slightly, which changes the capacitance. An illustration of this is seen in Figure 2.3. The accelerometer suffers from several noise sources. They are thermal noise, Brownian noise, and mechanical imperfections. Thermal noise affects the electrical transmission, and Brownian noise happens when air molecules collide with the sensing mass. Mechanical imperfections mean that the three axes inside the gyroscope are not perfectly orthogonal. The axes can also have different scaling, where the same acceleration gives different readings along different axes. Also, the axes may be misaligned with respect to the body frame, which is the structure holding the sensing unit. Mechanical errors are deterministic and can be compensated for by calibrating the accelerometer. The thermal, and Brownian noise are random noises, usually modeled as white noise.
2.4.1.2 Mathematical operations

If the acceleration is integrated it becomes velocity.

\[ v(t) = v_0(t) + \int_{t_0}^{t_1} a(t) dt \]  \hspace{1cm} (2.1)

However, because the noise is also integrated the velocity is subject to a random walk. The discrete integration also contributes to this problem, as mentioned in the previous section. A random walk is a mathematical object which is used to describe a process in which the variance increases for each added value to a time series. Since the variance increases it is not possible to predict the end result. To get from velocity to position, it is integrated in the same way.

\[ p(t) = p_0(t) + \int_{t_0}^{t_1} v(t) dt \]  \hspace{1cm} (2.2)

Since this also integrates the random walk, the variance will be proportional to \( t^2 \), while the velocity random walk is only proportional to \( t \).

To calculate the tilt angles, the accelerations are projected onto the gravity grid and the following equations can be acquired. Use Figure 2.4 for reference.
\[ \theta = \arctan\left( \frac{a_x}{\sqrt{a_y^2 + a_z^2}} \right) \]
\[ \phi = \arctan\left( \frac{a_y}{\sqrt{a_x^2 + a_z^2}} \right) \]
\[ \psi = \arctan\left( \frac{a_z}{\sqrt{a_x^2 + a_y^2}} \right) \]

\( \theta \), \( \phi \), and \( \psi \) are the tilt angles calculated from the accelerations. The MEMS accelerometers give an output in the positive direction for negative accelerations. As seen in Figure 2.4, the gravity direction is along the negative \( z \). This means that if the gravity is considered a right-hand system, the output of the accelerometer will be a left-handed system since all axes are reversed. This turns out to be the case with the accelerometer used in this project.

2.4.1.3 Translational acceleration

The accelerometer measurements are both from rotational and translational movements. In order to move the virtual hand with the MetaMotionC, it is necessary to try to separate the rotational and translational accelerations.

When the MetaMotionC is rotated without being moved, it will experience an acceleration perpendicular to the center of rotation. These are the
rotational accelerations. When the center of the MetaMotionC is moved, it is called translational movement and these are the accelerations that should be used to move the hand. The linear accelerations that the MetaMotionC outputs does not differentiate between the two.

2.4.2 Gyroscope

A gyroscope measures angular rate. If the angular rate is integrated once, the angular distance is obtained. To get the current angle, simply add the initial angle to the result.

\[ \theta(t) = \theta_0 + \int_{t_0}^{t_1} \omega(t) \, dt \]  

(2.3)

As in the case with the accelerometer, this is subject to a random walk. The gyroscope produces a low noise signal for the angular velocity, but because of the random walk, it will start to drift over time.

2.4.3 Sensor fusion

A popular method to get an accurate reading of the inclination angle is to use sensor fusion, where the gyroscopes short-term accuracy is joined with an accelerometers long-term accuracy, to obtain the best properties from both [4]. Modern algorithms typically use Kalman filtering for that kind of fusion, in which the gyroscopes measurement is used as the control input and the accelerometer as measurement input [5]. A simpler method is called the complementary filter. It uses a low-pass filter on the accelerometer and a high pass filter on the gyroscope and adds the signals together. This yields a drift-free signal with reasonable accuracy. A more common algorithm for sensors that also have a magnetometer is the Madgwick algorithm. It outputs absolute directions in quaternions [6]. It is able to output absolute directions since the magnetometer can be used to find the direction of north.

2.4.3.1 Quaternions

Quaternions started out as an extension of complex numbers during the 17th century, and have become useful in modern graphics computing. It’s a convenient way to calculate rotations. Instead of just one imaginary number, two more are added to the number system. Hence it’s a four-dimensional object. The rules that govern a quaternion are as follows.

\[ i^2 = -1, \quad j^2 = -1, \quad k^2 = -1 \]
\[ ij = k, \quad jk = i, \quad ki = j \]
\[ ji = -k, \quad kj = -i, \quad ik = -j, \quad ijk = -1 \]

So the general format of a quaternion is
\[ q_1 = a + bi + cj + dk \]
which can also be represented as a scalar part and a vector part
\[ q_1 = (r, \vec{v}) \tag{2.4} \]
Where \( r = a \) and \( \vec{v} = (bi, cj, dk) \). The formulas for addition will then be
\[ q_1 + q_2 = (r_1 + r_2, \vec{v}_1 + \vec{v}_2) \tag{2.5} \]
and for multiplication
\[ q_1q_2 = (r_1r_2 - \vec{v}_1 \cdot \vec{v}_2, r_1\vec{v}_2 + r_2\vec{v}_1 + \vec{v}_1 \times \vec{v}_2) \tag{2.6} \]
It should be noted that quaternions are not commutative \((q_1q_2 \neq q_2q_1)\), which follows from the rule outlined in Equation 2.6. Assume that \( \vec{v} \) is rotated around a unit vector \( \hat{n} \) which is perpendicular to \( \vec{v} \). Rotation around the vector can be described by the following equation
\[ \vec{v}' = \cos(\theta)\vec{v} + \sin(\theta)(\hat{n} \times \vec{v}) \tag{2.7} \]
Where \( \vec{v}' \) is the rotated vector.

![Figure 2.5: Rotation around axis \( \hat{n} \) in the plane. Visualisation of Equation 2.7.](image)

The rotation can be made with quaternions instead. The vector \( \vec{v} \) will be rotated around \( \hat{n} \). First, the vectors will be rewritten as quaternions.
The dot product $\hat{n} \cdot \vec{v} = 0$ since $\hat{n}$ is perpendicular to $\vec{v}$. Now according to Equation 2.7, $q_2$ can be written as

$$q_2 = \cos(\theta)q_1 + \sin(\theta)q_3q_1 = (\cos(\theta) + \sin(\theta)q_3)q_1 \quad (2.8)$$

Since $\hat{n}$ is a unit vector, it also has the following property

$$\hat{n}^2 = (0, \hat{n})(0, \hat{n}) = (-\hat{n} \cdot \hat{n}, \hat{n} \times \hat{n}) = (-|\hat{n}|^2, 0) = (-1, 0)$$

So $\hat{n}^2 = (-1, 0)$. This is similar to the complex number $i$. The Euler quaternion is $e^{bn}$ where $n$ is the quaternion around which the rotation is taking place. If $e^{bn}$ was expanded it would be $\cos(\theta) + \sin(\theta)nxi + \sin(\theta)nxj + \sin(\theta)nxk$ and using the same notation as before, it would be $q_n = (\cos(\theta), \sin(\theta)\hat{n})$.

The Equation 2.8 can be written as

$$q_2 = e^{bq_3}q_1 \quad (2.9)$$

if the vector $q_3$ is a unit vector. A unit quaternion has magnitude 1 and is also called a rotation quaternion.

Quaternions can also be used to rotate a 3D vector. Consider a vector that is rotated around an axis $\hat{n}$. The vector $\vec{v}$ is not rotated on the plane, as in Figure 2.5, instead its direction is away from the plane as seen in Figure 2.6.

Figure 2.6: Rotation of $\vec{v}$ around axis $\hat{n}$ off the plane.
The vector $\vec{v}$ consists of two parts, one part is parallel to the $\hat{n}$ vector and the other is perpendicular. They are found by projecting $\vec{v}$ onto $\hat{n}$ and onto the plane. These vectors will be called $\vec{v}_\parallel$ for the parallel part and $\vec{v}_\perp$ is the perpendicular part.

$$\vec{v} = \vec{v}_\parallel + \vec{v}_\perp$$  \hspace{1cm} (2.10)

When $\vec{v}$ is rotated around $\hat{n}$, the parallel part $\vec{v}_\parallel$ will not change, only $\vec{v}_\perp$. Then the change of $\vec{v}_\perp$ is the same as in Equation (2.7) which is the rotation in the plane. The rotation of $\vec{v}$ will be

$$\vec{v}' = \vec{v}_\parallel + \cos(\theta)\vec{v}_\perp + \sin(\theta)(\hat{n} \times \vec{v}_\perp)$$ \hspace{1cm} (2.11)

Where $\vec{v}'$ is the resulting rotated vector. A few observations can be made. The first is that $\vec{v}_\perp = \vec{v} - \vec{v}_\parallel$. The second observation is that

$$\hat{n} \times \vec{v} = \hat{n} \times (\vec{v}_\parallel + \vec{v}_\perp) = \hat{n} \times \vec{v}_\parallel + \hat{n} \times \vec{v}_\perp = \hat{n} \times \vec{v}_\perp$$

Where $\hat{n} \times \vec{v}_\parallel = 0$ since the vectors are in the same direction. The third observation is that $\vec{v}_\parallel$ is the projection of $\vec{v}$ onto $\hat{n}$ so the projection formula $\vec{v}_\parallel = (\vec{v} \cdot \hat{n})\hat{n}$ can be used. Making these three substitutions in Equation (2.11) yields

$$\vec{v}' = (1 - \cos(\theta))(\vec{v} \cdot \hat{n})\hat{n} + \cos(\theta)\vec{v} + \sin(\theta)(\hat{n} \times \vec{v})$$ \hspace{1cm} (2.12)

Which is called the Rodriguez rotation formula. With this knowledge, the formula for rotating 3D vectors with quaternions can be derived. First, a 3D quaternion is defined by setting $a = 0$

$$q_3 = (0, \vec{v}_3)$$

Using the same reasoning as before, the quaternion $q_3$ can be divided into a parallel and a perpendicular part, and only the perpendicular part will change during rotation

$$q_3' = q_3\parallel + e^{q_\theta q_\hat{n}}q_3\perp$$ \hspace{1cm} (2.13)

where $q_n$ is the quaternion around which the rotation is taking place. And the vector that is going to be rotated is $\vec{v}_3$. Multiplying a quaternion from the left is the equivalent of rotating the inverse of the same quaternion from the right, provided that the vector being rotated is perpendicular to the rotation quaternion.

$$e^{q_\theta q_\hat{n}}q_3\perp = q_3\perp e^{-q_\theta q_\hat{n}}$$ \hspace{1cm} (2.14)

Using the definition of the Euler quaternion yields

$$(\cos(\theta), \sin(\theta)\hat{n})(0, \vec{v}_3\perp) = (0, \vec{v}_3\perp)(\cos(\theta), -\sin(\theta)\hat{n})$$
Which according to Equation 2.6 is
\[(0, \cos(\theta)\vec{v}_{3\perp} + \sin(\theta)\hat{n} \times \vec{v}_{3\perp}) = (0, \cos(\theta)\vec{v}_{3\perp} - \sin(\theta)\vec{v}_{3\perp} \times \hat{n})\]
The real part is zero because the real part of \(q_3\) is 0 and the vector part of \(q_3\) is perpendicular to \(q_n\). On the right-hand side, the vector part turns out to be \(-\sin(\theta)\vec{v}_3 \times q_n\). The \(-\sin(\theta)\) is simply a scalar so it is factored out in front of the cross product, but the order of the cross product is the opposite from the left-hand side and the cross product does not commute. If the order of the cross product is changed, the expression becomes negative.
\[(0, \cos(\theta)\vec{v}_{3\perp} + \sin(\theta)\hat{n} \times \vec{v}_{3\perp}) = (0, \cos(\theta)\vec{v}_{3\perp} + \sin(\theta)\hat{n} \times \vec{v}_{3\perp})\]

Then the equality holds. In the case of the parallel part \(q_{3\parallel}\) it turns out to be commutative in this special case
\[e^{\theta q_n} q_{3\parallel} = q_{3\parallel} e^{\theta q_n}\]  (2.15)
Using the definition in 2.6 yields
\[(0 - \hat{n} \cdot \vec{v}_{3\parallel}, \cos(\theta)\vec{v}_{3\parallel} + \hat{n} \times \vec{v}_{3\parallel}) = (0 - \vec{v}_{3\parallel} \cdot \hat{n}, \vec{v}_{3\parallel} \cos(\theta) + \vec{v}_{3\parallel} \times \hat{n})\]
The cross product of the vectors are in the same direction and goes to 0. The dot product is, in fact, commutative so they yield the same result. That equality also holds. Equation 2.13 can also be written as
\[q_3' = e^{\frac{\theta}{2} q_n} q_{3\parallel} e^{-\frac{\theta}{2} q_n} + e^{\frac{\theta}{2} q_n} e^{\frac{\theta}{2} q_n} q_{3\perp}\]
Using Equation 2.14 and 2.15 the equation can be rewritten as
\[q_3' = e^{\frac{\theta}{2} q_n} q_{3\parallel} e^{-\frac{\theta}{2} q_n} + e^{\frac{\theta}{2} q_n} q_{3\perp} e^{-\frac{\theta}{2} q_n}\]
Which is the same as
\[q_3' = e^{\frac{\theta}{2} q_n} (q_{3\parallel} + q_{3\perp}) e^{-\frac{\theta}{2} q_n}\]
And using Equation 2.10 the equation can be written as
\[q_3' = e^{\frac{\theta}{2} q_n} q_3 e^{-\frac{\theta}{2} q_n}\]  (2.16)
This is the equation that can be used to rotate a 3D vector by a quaternion. The first part \(e^{\frac{\theta}{2} q_n} q_3\) rotates the vector half the distance and usually adds a real part to \(q_3'\), but that is removed when it is multiplied by the conjugate which rotates the vector the remaining half.
2.5 Discrete integration

A computer can only process discrete time (digital) signals, while a hand motion is continuous time (analog). If a computer is to make calculations on an analog signal, it needs to be discretized. That means that instead of having an infinite amount of changing data, the output is measured with a certain time interval $t_i$. The sample rate is then $\frac{1}{t_i}$, the rate at which new samples are received. When a signal is discretized, the information between the time interval is lost. Therefore any calculations done with a discrete signal is an approximation of a calculation with the real signal, provided that the original signal was not already discrete. The MetamotionC discretizes the accelerations through the ADC of its MCU. When a continuous signal is discretized, all information between the samples is lost. Therefore the motion of a hand can never be perfectly replicated from a discrete signal. However, there are many methods to integrate discrete time series. They are approximations of the continuous time integral, but some methods have less error than others.

2.5.1 Trapezoid rule

A common and simple method for discrete integration is the trapezoid rule, it uses the least amount of computations, and can calculate an integral with only two points. This is convenient in real-time systems that need to find the final result instantly. It is also a very crude approximation, assuming that the function value is constant between each step.

$$\int_a^b f(x)dx = (b - a)\frac{f(a) + f(b)}{2} + \epsilon$$

Where $\epsilon$ is the error. The trapezoidal method can also be written as a composite of many steps.

$$\int_a^b f(x)dx = \frac{h}{2} \left[f(a) + 2 \sum_{j=1}^{n-1} f(x_j) + f(b)\right] + \epsilon$$

Where $h = (b - a)/n$, $x_j = a + jh$ and the $\epsilon$ is the error compared to the continuous time integral. The summations can be split up into odd and even numbers of $j$, and can then be simplified to

$$I_{j,1} = \frac{1}{2} I_{j-1,1} + h_j \sum_{j=1}^{2j-1} f(a + (2j - 1)h_j)$$  \hspace{1cm} (2.17)$$

where $h_j = \frac{1}{2j-1} (b - a)$ and $I_{j,1}$ is the resulting discrete integral.
2.5.2 Romberg’s method

Romberg’s method is used to estimate the definite integral by applying Richardson extrapolation repeatedly on the trapezoid rule. A previous study showed that using the method could reduce the error significantly \[7\]. The general expression of Richardson extrapolation is

\[
I_{j,k} = \frac{4^{k-1}I_{j,k-1} - I_{j-1,k-1}}{4^{k-1} - 1} \tag{2.18}
\]

Where \(j\) is the order of extrapolation and \(k\) is the higher accuracy integral and \(k - 1\) is the lower accuracy \[8\].

The Romberg algorithm using Equation 2.17 and 2.18 will be

\[
I_{1,1} = \frac{(b-a)f(a) + f(b)}{2}
\]

for \(j = 2, 3,...\)

\[
h_j = \frac{b-a}{2^{j-1}}
\]

\[
I_{j,1} = \frac{1}{2}I_{j-1,1} + h_j \sum_{j=1}^{2^{j-1}} f(a + (2j - 1)h_j)
\]

for \(k = 2, 3,...\)

\[
I_{j,k} = \frac{4^{k-1}I_{j,k-1} - I_{j-1,k-1}}{4^{k-1} - 1}
\]

end

end

The integrals \(I_{j,k}\) can be seen in Table 2.2. The error is reduced in columns further to the right by using Equation 2.18 for \(j, k \geq 2\).

<table>
<thead>
<tr>
<th>Step</th>
<th>(O(h^2))</th>
<th>(O(h^4))</th>
<th>(O(h^6))</th>
<th>(O(h^8))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(h)</td>
<td>(I_{1,1})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(h/2)</td>
<td>(I_{2,1})</td>
<td>(I_{2,2})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(h/4)</td>
<td>(I_{3,1})</td>
<td>(I_{3,2})</td>
<td>(I_{3,3})</td>
<td></td>
</tr>
<tr>
<td>(h/8)</td>
<td>(I_{4,1})</td>
<td>(I_{4,2})</td>
<td>(I_{4,3})</td>
<td>(I_{4,4})</td>
</tr>
</tbody>
</table>

Table 2.2: Integrals further to the right have more precision. This is achieved by using the integrals in the previous column using Equation 2.18 for \(j, k \geq 2\).

This can then be extended to run over a moving window with a step size
The window step size must be chosen such that it can be divided into 8 segments in order to be applicable to the algorithm above. Thus \( w \) must be chosen as \( 8n + 1 \) where \( n \) is an integer. The integral will be with the least precision, which is just a trapezoid from \( a \) to \( b \). The next integral will use two segments, from \( a \) to \( \frac{a+b}{2} \), and then to \( b \). This way the loop acquires integrals with different precisions over the region. Then the inner loop can use these integrals of increasing precision to get an even more precise result.

2.6 Filters

2.6.1 Low-pass

Low-pass filters remove frequencies above a given cut-off frequency. There are many ways to accomplish a discrete low-pass filter, one common method is the moving average filter. The filter can be described mathematically by

\[
y_m(n) = \frac{1}{n} \sum_{i=0}^{n-1} x_{m(i-n)}
\]

(2.19)

where \( n \) is the size of the window. This filter has slow roll off and very bad stopband attenuation. However, the moving average filter is a good smoothing filter with low computation time [9].

2.6.2 Median

The median is defined as the middle value of an ordered series of \( n \) elements if \( n \) is odd. If \( n \) is even, it is the mean value of the two middle values. The median filter takes the \( n \) previous samples in the signal and returns the middle value. This removes sharp peaks in the signal.

2.6.3 Zero velocity update

When a noisy acceleration is integrated to the velocity it is subjected to a random walk. This means that the acceleration of a movement that starts with zero velocity and ends with zero velocity will still have some residual velocity from the noise. Therefore it is necessary to add a detector for zero velocity. There are two common approaches for this [10]. The first is used when the motion is cyclic, which makes the acceleration predictable. The
other is to detect when both acceleration and the angular rate is zero, and then assume that the velocity is zero. The drawback of the latter method is that it will assume that constant velocity is 0 since the acceleration, in that case, is 0. Neither of these methods will be applicable to this project since a slow hand motion could have close to constant velocity. Since this project is aimed at making a controller for smartphones and most of them have an optical sensor, it will be used to detect zero velocity. The drawback of this method is that it is computationally heavy and the view angle of the optical sensor may not be wide enough. There are accessories that extend the view angle, called fisheye lenses, but they degrade image quality and the edges are out of focus.

2.7 Image capture and analysis

2.7.1 Basics about images

Computer images can be represented in many different color spaces. A color space consists of numbers that are mapped to colors. There are many color spaces of different sizes, which means that some contain more nuances than others. If a larger color space is converted to a smaller, the colors are mapped to the closest representation of the nuance. The result is often a less vibrant image. For most algorithms in this project, 8-bit RGB will be used. The color space is additive and each channel is 8-bit, which means that red, green and blue can have values ranging from 0 to 255. When all the values are 255 the color is white, and when all are 0 it is black. The color space can be visualized as a cube in a space where the axes are red, green and blue values, as seen in Figure 2.7.
The intensity of a color is the mean value of the three channels. This means that the intensity is linear, but in reality, different colors contribute more to intensity than others, at least in the way that human eyes perceive intensity. Therefore weights are often used when adding the channels together. In addition to color space, some images also have an alpha channel assigned to them. The alpha channel is used for image transparency.

A commonly used tool when doing image analysis is looking at the histogram. A histogram is a bar chart that visualizes how many instances of defined properties that occur in a set. For example, the intensity can be used to classify pixels into distinct bars, as seen in Figure 2.8.
This technique is used in a lot of image processing algorithms. For example, thresholding, when the image is divided into black and white pixels. That could be illustrated with a histogram with only two defined classes, with the threshold being between the two classes.

\[ g(x, y) = \begin{cases} 
1, & f(x, y) > T \\
0, & f(x, y) \leq T 
\end{cases} \]

When the pixels have been classified, all the pixels of the class with containing pixels of lower intensity are set to black and the pixels in the other to white. Image analysis is usually computationally heavy. Even a relatively simple operation such as applying a threshold where pixels are sorted into two classes may take many thousands of CPU cycles. A fairly low-resolution image of 320x240 pixels would take more than 320 * 240 = 76800 cycles. That’s why dedicated hardware is often used for image processing.
2.7.2 Camera

A camera is usually modeled as a pinhole camera in computer graphics. An imaginary cone can be drawn from the pinhole. The pixels that are inside the cone will be mapped to the image plane that is displayed on the screen by doing a perspective transform and camera view transform. Perspective transform is used to make closer objects look larger than far away objects, and camera view transform turns world coordinates into camera coordinates.

\[ \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \]

where \((X, Y, Z)\) are world coordinates, \((u, v)\) are image coordinate, \(A\) is the camera matrix that contains information about principal point and focal length, \((c_x, c_y)\) is called the principal point which is often the image center, \(f_x, f_y\) are focal lengths in pixel units. The image is scaled by \(s\). The matrix containing \(r_{ij}\) and \(t_i\) is the joint rotation-translation matrix. That matrix is called the camera view transform. When the world coordinate \((X, Y, Z)\) is multiplied by that matrix it turns into the cameras coordinate system \((x, y, z)\).

Figure 2.9 shows an illustration of the perspective transform.
Equation 2.20 can be rewritten as

\[
\begin{bmatrix}
  x \\
  y \\
  z
\end{bmatrix}
= R \begin{bmatrix}
  X \\
  Y \\
  Z
\end{bmatrix} + t
\]

\[
x' = \frac{x}{z}
\]

\[
y' = \frac{y}{z}
\]

\[
u = f_x \times x' + c_x
\]

\[
v = f_y \times y' + c_y
\]

Objects with higher z-coordinates will appear smaller and vice versa.

2.7.3 OpenCV

OpenCV is a library for computer vision. It was started 1999 by Intel Research in an effort to improve CPU-intensive tasks. The library is open source with BSD license and can be used in academic or commercial endeavors. Scientists all around the globe help to develop this library. It is designed for efficiency and focused on real-time applications. It is fast due to its usage of pointers instead of having to copy large arrays in the memory. It also detects if the system has a compatible GPU, which is used rather than CPU.
Many of the algorithms are also optimized to run on multiple cores in parallel. OpenCV uses Mat objects to store images as matrices in the memory. It consists of a header and a pointer. The header contains information about the image size, memory address, storing method and other meta information. The pointer is used to access the pixel data of the image. This is useful because functions often only need the header information, which makes it unnecessary to move the whole image matrix. If the image is to be cropped, for example, a new header with smaller boundaries than the original can be created. The image data is not moved.

2.7.3.1 Fiducial markers

A fiducial marker is an object in an image that can be used as a point of reference. Any object can be used, such as a ruler or more commonly, a barcode.

2.7.3.2 Aruco markers

One of the more common objects to use in augmented reality is Aruco markers. These are black squares with white areas inside. The square can be divided into a grid of many smaller squares, which can then be read as binary numbers, where white areas are 1 and black 0. The grid size depends on how many numbers that need to be detected. The simplest marker that OpenCV supports is 4x4. Such a marker is illustrated in Figure 2.10.

![Figure 2.10: A 4x4 Aruco marker. The red lines illustrate the 4x4 matrix. This particular marker represents the number 1.](image)

The first step of the detection process is thresholding. This sorts pixels as black or white by an adaptive threshold value. Then it detects contours and discards concave contours and shapes that are not square, or which are the wrong size or too close to each other. The second step of the algorithm is bit extraction. A canonical form of the marker is obtained by applying a perspective transform. Then a threshold algorithm known as Otsu’s method is used. Otsu’s method maximizes the variance between the defined classes. Otsu’s method assumes that the image is bimodal,
which means that most pixels belong to two different intensities. If the image is not bimodal, Otsu’s method won’t be able to find a good threshold.

**Otsu’s method:**

With two defined classes, 1 and 2, the probability of a pixel belonging to either of the classes is

\[ P_1(T) = \sum_{i=0}^{T} p(i), \quad P_2(T) = \sum_{i=T+1}^{I-1} p(i) \]

Where \( T \) is the threshold, \( p(i) \) is the probability that a pixel is of intensity \( i \) and \( I \) is the number of intensity levels on the histogram. The mean value of the classes are

\[ \mu_1(T) = \sum_{i=0}^{T} \frac{ip(i)}{P_1(T)}, \quad \mu_2(T) = \sum_{i=T+1}^{I-1} \frac{ip(i)}{P_2(T)} \]

And the variances are

\[ \sigma_1^2(T) = \sum_{i=0}^{T} (i - \mu_1(T))^2 \frac{p(i)}{P_1(T)}, \quad \sigma_2^2(T) = \sum_{i=T+1}^{I-1} (i - \mu_2)^2 \frac{p(i)}{P_2(T)} \]

The between class variance is

\[ \sigma_b^2(T) = P_1(T)(1 - P_1(T))(\mu_1(T) - \mu_2(T))^2 \]

Maximizing the between-class variance is the same as minimizing within-class variance. The within-class variance weighted by the class probability is

\[ \sigma_w^2(T) = P_1(T)\sigma_1^2 + P_2(T)\sigma_2^2 \]

In a computer, it is easier to maximize the between-class variance, since it can be done in an iterative fashion without complicated calculations such as derivatives.

If the detected shape is indeed a marker, it should only contain black and white pixels, which makes it bimodal and a good candidate for Otsu’s method. After the marker has been thresholded it is divided into a grid. The black and white pixels are analyzed to see if they match an id in a specific dictionary. Error correction algorithms like Hamming can also be applied to increase accuracy.
2.7.3.3 Whycon markers

Whycon was developed to be an efficient way to localize concentric black and white circles of a known diameter [14]. The outer circle is black and the inner is white. A Whycon marker is illustrated in Figure 2.11. The marker is an ellipse when viewed from any angle that is not straightforward. When a marker is detected, a line can be drawn through the longest diameter and the shortest. When the cross product of these two vectors is calculated, a normal vector is acquired and thus also the marker’s rotation.

![Figure 2.11: A Whycon marker consisting of an outer black circle and a concentric smaller white circle. When the marker has been detected, a line is drawn on the longest and shortest distance respectively.](image)

In the first step, the image is thresholded with an adaptive threshold value in order to be more robust against uneven illumination. When the detection fails, the threshold is updated to \([1/2, 1/4, 3/4, 1/8, 3/8, 5/8...]\) up to a predefined granularity where the threshold is reset to its initial value [15]. On the other hand, if a successful detection is made, the threshold is set to

\[
T = \frac{\mu_{\text{outer}} + \mu_{\text{inner}}}{2}
\]

Where \(\mu\) is the mean brightness of the segments. This means that the algorithm assumes that the next marker will have roughly the same illumination as the last marker.

Then it looks for segments of continuous black pixels using the flood-fill method. It is then tested for circularity. If the segment is composed of \(n\) pixels with width \(w\) and height \(h\) and the circle inner and outer diameters are \(d_i\) and \(d_o\) respectively, first a ration between the black and white pixels is obtained by

\[
\text{ratio} = \frac{\pi\left(\frac{d_o}{2}\right)^2 - \pi\left(\frac{d_i}{2}\right)^2}{\pi\left(\frac{d_o}{2}\right)^2} = > \frac{d_o^2 - d_i^2}{d_o^2} \quad (2.21)
\]

The circle will most likely look like an ellipse because of camera distortion and pattern orientation and deformation. The area of an ellipse of width \(w\)
and height $h$ is

$$A = \frac{\pi w h}{2} = \frac{\pi}{4} wh$$

(2.22)

Now Equation 2.21 and 2.22 can be combined to get the number of black pixels

$$n = \frac{\pi}{4} wh \frac{d_o^2 - d_i^2}{d_o^2}$$

With some algebra it becomes

$$0 = \frac{\pi}{4n} wh \frac{d_o^2 - d_i^2}{d_o^2} - 1$$

And instead of 0, a tolerance level can be introduced

$$p_{tol} > \left| \frac{\pi}{4n} wh \frac{d_o^2 - d_i^2}{d_o^2} - 1 \right|$$

The algorithm then proceeds to see if the center segment is full of white pixels with the same flood-fill technique, and a roundness test is also done for the white pixels, but this time the test is

$$p_{tol} > \left| \frac{\pi}{4n} wh - 1 \right|$$

Since the bounding box now only contains the white pixels, no ratio is needed. The next task is to find the semi-axes $e_0$ and $e_1$. The covariance matrix is calculated

$$C = \frac{1}{n} \sum_{i=0}^{s-1} \begin{bmatrix} u_i u_i & u_i v_i \\ v_i u_i & v_i v_i \end{bmatrix} - \begin{bmatrix} uu & uv \\ vu & vv \end{bmatrix}$$

Where $u$ and $v$ are the center coordinates of the segment, which is the mean of the pixel positions. The variables $u_i$ and $v_i$ are the coordinates of each pixel in the segment. Then the eigenvalues $\lambda_0$, $\lambda_1$ and eigenvectors $v_0$, $v_1$ of the covariance matrix are calculated. They are used to find the semi axes

$$e_0 = 2\sqrt{\lambda_0} v_0$$
$$e_1 = 2\sqrt{\lambda_1} v_1$$

The covariance matrix $C$ is real and symmetric and can be written as

$$C = U \Lambda U^T$$
Where $U$ is orthonormal so that $UU^T = I$. $\Lambda$ is a diagonal matrix with eigenvalues. So it follows that

$$C^{-1} = U^{-T}\Lambda^{-1}U^{-1} = U\Lambda^{-1}U^T = \sum_{i=1}^{d} \frac{1}{\lambda_i} u_i u_i^T$$

And then Mahalanobis distance, which is the Euclidean distance of $x$ from $\mu$ in a rotated and scaled coordinate system, is used so that

$$D^2 = (x - \mu)^T C^{-1} (x - \mu) = (x - \mu)^T \left( \sum_{i=1}^{d} \frac{1}{\lambda_i} u_i u_i^T \right) (x - \mu)$$

Factor it and it can be written as an ellipse, which is illustrated in Figure 2.12

$$D^2 = \sum_{i=1}^{d} \frac{1}{\lambda_i} (x - \mu)^T u_i u_i^T (x - \mu) = \sum_{i=1}^{d} y_i^2 \frac{y_i}{\lambda_i}$$

Since $(x - \mu)^T u_i$ and $u_i^T (x - \mu)$ are the same, namely the distance $y_i$.

![Figure 2.12: Ellipse with the quadratic form $Q = \frac{y_0^2}{\lambda_0} + \frac{y_1^2}{\lambda_1}$.

Then the last test is applied to determine if the number of pixels in the segment matches the area of an ellipse

$$t_{tol} > |\pi |e_0||e_1| n - 1|$$

This tolerance can be much lower than $p_{tol}$, since $e_0$ and $e_1$ are calculated from the covariance matrix which means that they have sub-pixel precision [13]. The covariance matrix is the most intensive task for the CPU because it involves integer arithmetic. The other calculations are relatively simple in this algorithm. The semi-axes can be used to determine the spatial position of the marker, provided that the camera has been calibrated and the size of
the marker is known. First, the ellipse covertices coordinates are converted to
camera canonical coordinates, with the parameters obtained during camera
 calibration. Then the center is determined in camera coordinates and a conic
$O$ is determined so that all ellipse covertices $u', v'$ satisfy

$$\begin{bmatrix} u' \\ v' \\ 1 \end{bmatrix}^T O \begin{bmatrix} u' \\ v' \\ 1 \end{bmatrix}$$

Then eigenvectors $\lambda_0, \lambda_1, \lambda_2$ and eigenvectors $q_0, q_1, q_2$ of the conic $O$ are
calculated. The position is then determined by

$$p = \frac{d_o}{\sqrt{-\lambda_0 \lambda_2}} \left( q_0 \lambda_2 \sqrt{\frac{\lambda_0 - \lambda_1}{\lambda_0 - \lambda_2}} + q_2 \lambda_0 \sqrt{\frac{\lambda_1 - \lambda_2}{\lambda_0 - \lambda_2}} \right)$$

## 2.8 Communications: Bluetooth and Internet

Bluetooth is used to connect the controller to the smartphone. However, due
to limitations in the Unity editor, the Bluetooth connection cannot be used
to connect the MetaMotionC to the PC. Therefore the internet protocol is
used to connect the PC to the smartphone, which in turn is connected to the
MetaMotionC. The smartphone relays the information from MetaMotionC
to the PC.

### 2.8.1 Bluetooth Low Energy (BLE)

BLE is a version of Bluetooth that is optimized for energy efficiency, as the
name suggests. Classic Bluetooth was designed to stream data continuously
at short ranges, for example, file transfers or audio transmissions. In contrast,
BLE was designed to send information in short bursts and stay in sleep
mode in between. This is useful in applications that only send information
periodically. A small battery can last for years if the communication is kept
to a minimum. The data rate of BLE is typically 2-300 kbps, while classic
can reach up to 3 Mbit/s. The send time for messages is limited to 6 ms in
BLE, while it is 100 ms in classic.

### 2.8.2 Internet protocol

Computers and smartphones can communicate with each other using a network.
The network uses a communication protocol called internet protocol (IP). There are two higher level protocols that use IP. The most commonly used is transmission control protocol (TCP). The TCP synchronizes
the sender with the receiver and makes sure the network packets are received in the correct order. If the receiver does not send back a confirmation that the packet was received, it is considered lost and is resent after a set period of time. The other protocol is user datagram protocol (UDP). It is a connectionless protocol, meaning that the sender does not bother if the packets are received or which order they are received in. UDP is used when high speed is the priority, such as with games or live stream. The packet loss is normally very low with UDP, less than 0.01% under normal circumstances.

3 Implementation

3.1 Development tools

In order to be able to test the controller, some kind of testing environment will have to be used. There are many free game development platforms available. They are also called game engines because it is where all the parts come together. A game engine can include a rendering engine that generates graphics and lighting, a physics engine that emulates natural laws of physics, an audio engine that is responsible for playing sounds etc.

3.1.1 Unity

One of the more popular free game development platforms is Unity. It has all the necessary functions to create both 2D and 3D games. It supports C# scripting and can export the games to multiple platforms. Most importantly for this project, it can export to Android and there is an SDK to support VR. Unity also has an asset store, in which plugins and models can be downloaded and installed directly in Unity.

Everything in Unity is considered a game object unless the designer specifies otherwise. Each game object is derived from a base class called MonoBehaviour. This contains several functions, of which the most important two are Start() and Update(). In the Start() function the designer can initialize variables and settings before the game actually starts. Update() is called once on each displayed frame. So if the frame rate is set to 30 fps, the Update() will be called 30 times per second. Unity uses a left-handed coordinate system. Each game object also contains a transformation script, which holds translation, rotation, and scale. The rotation is stored as a quaternion but is presented in the user interface as Euler angles. The quaternion rotation prevents gimbal lock. Each game object has a local and a global coordinate system. The global coordinates use the world origin as reference. The game
objects can be arranged in a hierarchical order so that the game objects local coordinate system uses its parent’s global coordinates as reference.

Unity’s UI consists of five important windows. The first is the Hierarchy, in which all the current scene’s game objects are shown and hierarchical order can be changed. The scene window shows a visual representation of the current scene. It can be used to move, rotate and scale objects. The Inspector window shows properties and scripts that are attached to the currently selected game object. Variables in scripts that have been declared public, can be initialized to a value that can be set by the inspector. The physical folder structure can be seen in the Project windows. Files and folders can also be moved and renamed in this window. There is also a Console window, which outputs information from Unity’s Debug.Log() function and error messages. The game can be run in Unity’s editor without compiling for a specific console. This is useful for testing since compiling for a console can take several minutes, while the editor can run the game in seconds.

Unity also uses a special component that can be added to the game objects called colliders. These colliders can have different shapes, but their most important property is that they will report if they have connected with another collider in each frame.

3.1.2 Blender

Blender is a free 3D graphics software. It can be used to create animated movies, 3D models and even games. In this project, it will be used to create the hand model that the player will be able to control with the Bluetooth controller. It has a rigging system which means that bones can be applied to a model to emulate a skeleton. The skeleton can be used to animate the model.

A drawback when using Blender models with unity is that they use different coordinate systems. Unity uses right-handed XYZ and unity uses left-handed. Both consider Y axis to be upward. But this also means that the models from blender will be mirrored when they are imported to Unity.

3.1.3 MakeHuman

Making humanoid models in 3D is difficult and time-consuming. It requires great knowledge about the topology of body parts. A free software that simplifies this process is MakeHuman. It is specifically used to create humanoid 3D models. It uses one base mesh and four morphing targets which can then be linearly interpolated to achieve all intermediate shapes. The UI in make human consists of several sliders to change body shape and characteristics.
In 3D-modeling there is a concept called rigging, where the model is attached to a bone structure which allows for easier animation. When a bone is moved, the modeled muscle that is attached to it will follow. MakeHuman also has an option to export the model together with a bone structure rigged to it.

### 3.2 Methods

Everything is implemented in Unity, using C# to make custom functionality. The variables defined in this document have different names than the ones used in the code. The names in the code are more descriptive, whereas the names in this report are purely for theoretical demonstration.

#### 3.2.1 Testing room

A testing room consisting of a room with four walls, a ceiling and floor and some furniture was made in Unity. Colliders were attached to the walls in order to be able to stop the VR hand from moving through them. This room was placed in the top layer of Unity’s hierarchy. Then an SDK for Android VR provided by Google was installed. The SDK contains a custom main camera that can be used as the eyes of VR character. It connects to the accelerometer and gyroscope in the Android device so that the VR characters head rotates along with the device. Then a Unity asset called OpenCV for Unity was installed. With that, most of OpenCV’s functions and objects can be used from custom C# scripts in Unity.

There was no way to connect to the MetamotionC directly from Unity. In the Android OS, all apps use something called activities and have at least one main activity. Unity’s activity is called UnityPlayerActivity. A compiled app has the extension apk, and is therefore called an apk-file. Android also uses a manifest file, which describes an app’s activities, permissions etc. The manifest is the first file that the Android OS reads and is used to decide which of the app’s activities to start first.

A custom app was written in Android studio, extending the UnityPlayerActivity. Methods were added to connect to the MetaMotionC. Then a jar file was compiled and placed in a special folder in Unity together with a modified manifest file in which the new activity was defined. Unity’s compiler will read the new jar file and the manifest when compiling the app for Android. The details of the methods in this jar file will be covered in Section 3.6.1.
3.2.2 User interface

Unity’s Input.GetKeyUp() only reacts to keyboard input. In order to be able to press keys on the Android device, game objects were added and placed to the left of the screen to act as buttons (Figure 3.1). Unity has a button script that makes it easy to add OnClick() methods, which will be executed once the button is pressed. The V button hides all buttons except itself, and the Cam button will be used to switch camera. For example, an Android device usually has a front-facing camera and a back-facing camera. The switching of the camera is done in WebcamTextureToMatHelper, explained in Section 3.4.1. The rest of the buttons will be linked to methods that will be explained in Section 3.3.6 and Section 3.6.1.

![Figure 3.1: Buttons to the left can be used to change settings when the app is running.](image)

3.2.3 Player

A game object called Player was added to represent the user’s body in the VR world. It was also placed in the top layer of Unity’s hierarchy since it is independent of all other objects’ orientations. A C# script called Player-ControlScript was attached to the Player. A camera game object was also added and placed below the Player in the hierarchy. This is the only camera in the scene and will, therefore, be the main camera. It is possible to have several cameras in Unity, but only one main camera. In this project, this camera is the equivalent of the players head and eyes. When the Players
body moves, the head and eyes should follow. By placing the camera below the Player in the hierarchy, it will use the Player object’s coordinate system as reference. This means that it will treat the Player object’s current coordinates as the origin. PlayerControlScript has one input, that is the quaternion rotation of the camera. A reference for the camera was added to PlayerControlScript so that the Player object can access the camera’s rotational values. Googles VR SDK will control the orientation of this camera when it is compiled for Android. The SDK provided game object called GVREditorEmulator which controls the camera when running the game in the editor, by pressing CTRL or ALT on the keyboard, for tilt and rotation respectively. In the Update() method, Unity’s Input.GetAxis(“Horizontal”) and Input.GetAxis(“Vertical”) methods were used to read control input from a keyboard or controller. They were multiplied by Time.deltaTime, which holds the difference in time between the last frame and the current frame. This way the movement can be scaled so that higher or lower FPS still yields the same movement speed. These two inputs were used to create a movement vector for x- and z-axis movement. Then the method transform.Translate() is used to move the player, but the movement is first multiplied with the rotation of the camera around the y-axis. This makes it so that the body follows the head rotation. When the forward key is pressed, the player will move in the direction that the eyes are looking. Except up or down, that is why only rotation around the y-axis is used. Otherwise, the player would be able to levitate by pressing forwards when looking at the sky.

3.2.4 Hand model

In this project, a glove will be made for the left hand. The VR-headset that is used has an opening for the smartphone camera on the left side, so it is the most appropriate hand to start with. The default character in MakeHuman was exported with a bone structure and imported to Blender. There were several bone structures to choose from, the most detailed was selected in order to have access to each bone in the fingers. In Blender, everything was removed except for the left hand. The vertices at the wrist were merged together so that the hand did not look hollow. The hand was then imported into Unity, and its bones were mapped to Unity’s internal rigging system so that they were exposed to Unity’s animation controls and coordinate system. This makes it possible to move individual bones instead of the whole hand. Colliders were added to the thumb and index finger joints so that the user will be able to interact with the VR world. A collider was also added to the palm of the hand.

Since the hand model is not dependent on any of the other objects rota-
tions or positions, it was placed at the top of Unity’s hierarchy. This makes it a sibling to the Player object, which means that they are independent of each other. This is not realistic since a hand should remain in close proximity to the body. But this hierarchy results in fewer calculations for the rotation of the linear accelerations that will be explained in Section 3.2.6. The hand will not be programmed to stay in close proximity to the Player object in this project, because it is not relevant, but it is easy to implement by adding the position coordinates of the Player object to the VR hand.

### 3.2.5 Coordinate indicators

Four game objects were created and placed in the scene. Two of them will represent the VR coordinate system, the other two will represent the rotation of the Bluetooth controller. A script called CoordinateSystem was attached to each of the game objects. The objects that represent the Bluetooth controller will use its rotation as input. All four objects were placed below the camera in the hierarchy so that they will follow the camera’s movements. Two of the objects, one of each, were placed in the top left corner on top of each other. The other two were placed in the right corner, also on top of each other. For the ones that represent the coordinate system, Debug.DrawRay() was used to draw a red, green, and blue line in world coordinate x, y, z direction respectively. The other two will do the same, but then the object’s rotation will be multiplied by the Bluetooth controller quaternion and \( q \), described in section 3.2.6. That means that the lines will represent the hand’s rotation. These objects can be used to find the difference between Unity’s world coordinate system and the Bluetooth controller. By aligning the lines of the objects, the offset can be measured. This is useful since the Bluetooth controller outputs absolute quaternions which represent the orientation in the real world. With the offset, the rotation in the real world can accurately be mapped to the VR world. The reason that two of these objects are used, one in each corner, is that the perspective can sometimes be deceiving. It may look like the lines of the systems align, but it could also be that the line’s rotation causes its perspective projection to align only on a 2D plane. An example of this is shown in Figure 3.2, where the green lines that are pointing straight upwards correspond to Unity’s y-axis, the red lines that are pointing to the right is the x-axis and the blue lines pointing forward is the z-axis. The other lines reflect the Bluetooth controller rotation. On the right side, it looks like the green line is lining up with Unity’s system. But the left indicator shows that it is only the projection in this perspective that makes it look like that. All the lines with the same color must be aligned on both indicators to find the Bluetooth controller’s rotation offset from Unity’s
coordinate system.

Figure 3.2: Coordinate system indicators. One indicator in each corner is static and shows Unity’s world coordinate system. The other indicator is rotating with the VR hand rotation. When all colored lines are aligned with each other, the offset rotation of the Bluetooth controller has been found.

3.2.6 Coordinate system

The quaternion values from the Bluetooth controller are given with a left-handed coordinate system. Since Unity is also using a left-handed coordinate system, all that needs to be done is aligning the axes of the Bluetooth controller to Unity’s axes. This is done by a rotation. The hand model in Unity has the system (x,y,z) and it corresponds to the Bluetooth controller (-y, -z, x). Therefore the values received Bluetooth controller needs to be rotated as seen in Figure 3.3. This is done by rotating -90° around z and subsequently -90° around x. All rotations are done in a left-handed system, which means that they rotate the opposite way compared to the right-hand rule which is often used in mathematics.
With quaternions, it is possible to create a rotational quaternion that can make this rotation in one step. First, a rotation quaternion \( q_0 \) is defined as the identity quaternion, \( q_z \) is defined as the rotation of \(-90^\circ\) around \( z \) and \( q_x \) as the rotation around \( x \). Using the notation explained in Equation 2.4, the quaternions are

\[
q_0 = (1, (0, 0, 0)), \quad q_z = \left(\frac{1}{\sqrt{2}}, (0, 0, -\frac{1}{\sqrt{2}})\right), \quad q_x = \left(\frac{1}{\sqrt{2}}, (-\frac{1}{\sqrt{2}}, 0, 0)\right)
\]

First, rotate around \( z \)

\[
q_{01} = q_0 * q_z = \left(\frac{1}{\sqrt{2}}, (0, 0, -\frac{1}{\sqrt{2}})\right)
\]

Then rotate around \( x \)

\[
q_r = q_{01} * q_x = (-0.5, (0.5, -0.5, 0.5))
\]

And \( q_r \) is the resulting rotation quaternion. The quaternions received from the Bluetooth controller will be right hand multiplied with \( q_r \) to rotate it to the VR hand coordinates.

Since the Bluetooth controller output absolute quaternions, it needs to be aligned with the VR hand. This is done by holding the hand in any position and pressing the C-key on the keyboard. This position will represent the VR-hands’ default position, which is that the VR-hand is pointing straight-forward and the \((x,z)\) plane being level with the cameras \((x,z)\)-plane. Since every hand and glove are slightly different, the default position quaternion will be different in each case. The program stores the quaternion as \( q_c \). The subsequent quaternions \( q_i \) from the Bluetooth controller are multiplied from the left with the inverse of \( q_c \), since all new rotations should use \( q_c \) as the reference point. However, \( q_c \) also has to be rotated to Unity’s coordinate
system. This is done by multiplying with $q_r$ from the right. In total, the resulting quaternion is $(q_c * q_r)^{-1} * q_i * q_r$, where $q_c * q_r$ is the reference and $q_i * q_r$ is the current rotation. In order to reduce the number of quaternion multiplications, $(q_r * q_r)^{-1}$ is stored in a separate quaternion $q_p$.

While rotating the quaternion output from the Bluetooth controller is quite straightforward, the same approach cannot be used for linear accelerations. The linear accelerations are received as a 3D vector from the Bluetooth controller. The linear accelerations are given in a right-handed coordinate system from the Bluetooth controller. This means that no matter how it is rotated, one axis will be in the opposite direction compared to Unity’s system. This is solved simply by making the z-axis negative to make the system left-handed.

Unity uses Equation 2.16 to rotate 3D vectors by quaternions. The notation in C# is simply $q_w * \vec{v}$. To rotate the opposite direction $q_w^{-1} * \vec{v}$ can be used. In order to know how much the vector should be rotated, it is necessary to know the quaternion that matches Unity’s system to calculate the difference. This is done by holding the Bluetooth controller such that the four coordinate indicators, described in 3.2.5, are aligned. Then the quaternion can be stored by pressing the x-key. The aligned quaternion is then multiplied with $q_r$ from the right, to bring it into Unity’s coordinate system. This quaternion will be called $q_a$ and represents the quaternion where the Bluetooth controller is aligned with Unity’s system.

The linear acceleration vector is called $\vec{v}_a$. It also has to be converted into Unity’s coordinate system. This is done in a similar fashion as the rotation quaternions, but the coordinate system of the linear accelerations was slightly different. It requires a $90^\circ$ around the z-axis, where the rotation quaternions required $-90^\circ$. This is after making the z-axis negative. This rotation should be multiplied from the right with $\vec{v}_a$ and is therefore inverted. The resulting quaternion is $q_{ra} = (0.5, 0.5, 0.5, -0.5)$. In order to obtain the correct accelerations in Unity coordinates, it is necessary to rotate the accelerations from the Bluetooth controller to Unity’s. Therefore $(q_a)^{-1}$ is multiplied from the right. Then the current VR hand’s rotation also needs to be applied. This is done by $\vec{v}_a$ with $q_h^{-1}$ from the right, where $q_h$ is the quaternion that represents the VR hand’s rotation. Lastly, $\vec{v}_a$ is multiplied with $q_{ra}$ from the right, to convert from the Bluetooth controller to Unity coordinates. The complete calculation to rotate $\vec{v}_a$ to Unity’s system is $\vec{v}_a * q_a^{-1} * q_h^{-1} * q_{ra}$. Since Unity can only multiply quaternions from the left with 3D vectors, the final result used in Unity is $q_a * q_h * q_{ra}^{-1} * \vec{v}_a$. 

44
3.3 Movement

The movement of the hand is made with a combination of the accelerometer and camera image input. The accelerometer is only accurate for a short period of time, within seconds it becomes unreliable because of the random walk error. The camera image has millimeter precision, but can only be used while the hand moves very slowly. If the hand moves fast the image will have too much motion blur to detect any markers. Figure 2.1 is a flowchart of the program. In the following sections, the smallest parts of the program concerning accelerations are described, followed by Section 3.3.6 where all the parts come together in one script.

3.3.1 UDP Client

A game object called UDPRceiver was created in the scene and a script named UDPserver was attached to it. This script has a reference to the VR hand object, in order to be able to update the current quaternion. It has only one setting, and that is the port number that it should listen to. In UDPserver’s Start() function, createUDPClient() is called. An object called udpReceive is created by calling UDPRceive.CreateInstance(). UDPRceive is a special kind of script in Unity, which is called scriptable object. It does not use the Start() and Update() methods that game objects use, but has other properties. More information about the scriptable object can be found in the Unity documentation. UDPRceive.CreateInstance() takes one input, which is port number. This can be any available port on the device. Port 18111 was arbitrarily chosen for this project. Three variables are defined in the UDPRceive.CreateInstance() method. The first is an IPEndPoint object, which takes an IP address and a port number as input. In this case, 0.0.0.0 is used which means that it will listen to all of the device’s assigned IP addresses. The other variable is called udpc and is a UdpClient object which is a class provided by C#. It takes a port number as input. The third variable is an AsyncCallback object, which takes a delegate method as input. Using AsyncCallback will make the UDP connection run in a separate thread, so that the game can continue without waiting for the UDP communication to finish. After the three variables are defined, StartUdpReceive() is called. This method contains only one line of code, udpc.BeingReceive(), which tells the UDPClient to begin to listen for messages. It takes two inputs, the AsyncCallback and a C# object which is used to hold the received information. Normally, the receive method blocks the thread until a message is received. But with an AsyncCallback, when a message is received, it is sent to the delegate method, which in this case writes it to a static variable
called ReceivedMsg, and runs StartUdpReceive() again in order to begin listening for the next message. The static variable can be accessed from any object. In UDPServer’s Update() function, ReceivedMsg is read and split at the commas, since the expected messages contain comma separated values. The message is split by using C# String.Split() method, which takes one character as input, which is the separator which will be used to split the string. It returns an array of separated values. Then float.Parse() is used to convert strings to float numbers. All of the received numbers will be floats, except for the time which will be a long int. So the last value is parsed with long.Parse() instead. The VR Hand is then updated with the new quaternion, accelerations and time values.

3.3.2 Sensor data collector

A game object called SensorDataCollector was added to the scene. A C# script called MetaMotionText was attached to it. It has one setting, and that is which Bluetooth address to connect to. The script holds a reference to the VR hand, just like the UDP client, to be able to update the VR hand’s rotation and position. The Start() method begins by defining two variables. The first variable is an object that is created with Unity’s AndroidJavaClass class. It takes the name of the Android package as input. In the Android OS, each app has a unique package name in which the app’s activities can be identified with. In this case, ”com.unity3d.player.UnityPlayer” was used as the package name. The variable was called unityClass.

The other variable holds a reference to the current activity in the package, which is obtained by calling unityClass.GetStatic <AndroidJavaObject> (“CurrentActivity”). This variable is called unityActivity.

When the B button (Section 3.2.2) is pressed, the method OnBluetootClick() is executed. It checks if the board is connected, if it is not then unityActivity.retrieveBoard() is called. It takes one which is the Bluetooth address to connect to. Then it checks if the sensors are started and if data is being collected. If it is not, then the method unityActivity.ConnecToBoard() is called. The reason it checks if the board is connected and collecting data, is so that it will not reconnect if the B button is pressed by mistake. If the D button is pressed, the sensors are deactivated and the device is disconnected. If the app is shut down without first disconnecting the Bluetooth controller, its sensors will keep running and the battery will be drained. These methods will be described in greater detail in Section 3.6.1.

In the Update() method, the data of the VR hand is updated by calling unityActivity.Get<>(). It takes a string as input, which corresponds to a variable defined in the OverrideActivity (Section 3.6.1). These variables are...
the four numbers of the quaternion, the accelerations in x, y, and z, and the time.

3.3.3 File Management

In order to be able to analyze the same movement several times, some way of recording was needed. C# provides two objects that are useful for writing and reading files on the hard drive, StreamWriter, and StreamReader. A file management class was written to handle the creation, opening, and closing of files. It has only one setting, that is the folder path in which the files should be stored. It holds two lists, one of all files that are being read and one of all files that are being written. The read file list is called readFiles and the other is called writeFiles. They are created with the Dictionary class provided by C#, which uses an index key and an object as input. The object can later be retrieved by specifying the correct index. The FileManagement class has similarly named methods as StreamWriter and StreamReader, but they all use a filename string as input. The file name is used as the index in the list. To start writing to a file, the method RecordFile() is called. It creates the file with StreamWriter in the previously defined file path. If the file already exists it is deleted first. Then it is added to the list writeFiles. The filename is used as an index in the list. To read a file, the method readFile() is used. It opens the file with a StreamReader and adds it to the readFiles list. An exception can be thrown by the StreamReader if the file does not exist, but Unity does not crash from it so the exception can be ignored. The other details of this class will not be part of this report, since it uses the same methods as StreamReader and StreamWriter, but uses the filename as input for the methods so that the appropriate object can be accessed from a list. The main purpose of this class is to reduce lines of code in the other classes that use files.

3.3.4 Filters

The smoothing filter was implemented by using a list of floating point numbers to store the window history. It uses three inputs, a key, a value, and window length. In this case, a window length of 10 is used. Each time the filter receives a new input, the value at position 0 in the list is removed and the new input value is added to position 9. A C# object called Dictionary is used to store the list. The key input is used as the index in the Dictionary. This means that any number of filters can be used. In this case, only three are needed, one for each axis. The filter output is a summation of all the values in the list of the corresponding input key, divided by the length of the
list.  

The threshold filter was implemented such that if the absolute value of the filter input is smaller than the threshold, the filter outputs zero. Otherwise, it returns the input value. It uses the absolute value because the negative values also need to be thresholded.

The median filter was implemented in the same way as the smoothing filter, using a key and value as input, so that window history for each axis could be kept. This window has a length of 3. The window is sorted and the middle value is returned as the filter output. A helper class was created so that the main program only need to call one method to filter the three axes. This method takes four inputs: three values, and window length. The method uses different keys for the three inputs and calls the method described at the beginning of this section.

### 3.3.5 Acceleration to position

The acceleration vector \( \vec{a} \) is in the unit \([g]\), so it is multiplied by earth’s gravity constant 9.81. The trapezoid method is then used to calculate the velocity

\[
\vec{v}_i = \vec{v}_{i-1} + \frac{\vec{a}_i + \vec{a}_{i-1}}{2} \times (t_i - t_{i-1})
\]

The distance is calculated in almost the same way, but only the difference in distance from the last sample is needed, so the previous distance is not added

\[
\vec{d}_i = \frac{\vec{d}_i + \vec{d}_{i-1}}{2} \times (t_i - t_{i-1})
\]

### 3.3.6 Implementation of movement in Unity

A script called CustomMovement is attached to the VR hand model. It holds references to two Bluetooth controller communication objects, one is a UDP receiver and the other is for Bluetooth. The inputs to the CustomMovement class are eight numbers. Four of the numbers make up the quaternion rotation, three of them are accelerations in x, y, and z and the last input is the timestamp for when the data was collected. These inputs come from either the SensorDataCollector or the UDPReceiver. The settings that can be changed via the inspector regarding the Bluetooth controller data are

- Booleans (True or False)
  - Use UDP client
  - Record UDP Messages
– Use Recorded Movements
– Stop Game After One Iteration

• Strings (Text)
  – UDP Filename (text string)
  – Recorded Movements File Name (text )

If Use UDP Client is true, a game object called UPDReceiver is enabled and the object SensorDataCollector is disabled, and vice versa if it is false. This is done to ensure that the VR Hand does not receive rotation data from more than one source.

If Record UDP Messages is true, every time the hand is rotated, the quaternion, accelerations and time values will be stored in a text file. The name of the text file will be the value of UDP Filename.

Use Recorded Movements will read a text file that was saved before. The name of the file that it will try to open is the value of Recorded Movements File Name. The option Stop Game After One Iteration will stop the game when the program reaches the end of the file. This is useful when debugging rotations and filters.

Regarding the accelerations the settings are

• Floats (numbers)
  – Threshold
  – Movement speed (Vector of three floats)

• Booleans
  – Activate Acceleration To Motion
  – Output Filtered Movements

• Strings
  – Output Filename

The threshold is the value that is used for the threshold filter. Activate Acceleration To Motion simply disables or enables movement that is caused by Bluetooth controller values. This is useful when working with other scripts that also moves the VR Hand, such as the pattern recognition algorithm. Movement Speed is a 3D vector containing scaling factors in x,y, and z direction. Output Filtered Movements writes the x, y, z and time values of the rotated and filtered accelerations to a file. The name of that file is the value
in Output Filename. With these variables in place, it is possible to record a set of movements, read them, output the filtered and rotated data and stop at the end of the recorded file.

The \texttt{Start()} function begins by checking of a stored value for $q_a$ is available. This is only supported in the editor, because of a permission issue in Android that prevents Unity from storing files. It is useful to store this quaternion so that the user does not have to calibrate the Bluetooth controller to Unity’s coordinate system each time. Then a \texttt{FileManagement} object, described in Section 3.3.3, is created to record and read movements. The FileManagement is also available only in the editor, because of the same permissions problem as mentioned before. After that, the UDP Client or the Sensor data collector is enabled as described above. Then all the text files that will be used for reading or writing are initialized with the FileManagement object. Lastly, the quaternions needed to translate the Bluetooth controller’s coordinate system into Unity’s are initialized.

The \texttt{Update()} function of the CustomMovement script begins by checking if Use Recorded Movements was checked. If it is, the program calls \texttt{NextMovements()} which reads data from the specified file and updates the values quaternion, accelerations and time. After that, it calls \texttt{RotateHand()}, which rotates the hand with the new values if the quaternion has changed since last time. This is also where UDP messages are recorded if that option was checked. Then it checks if Activate Acceleration To Motion was checked. If it was then \texttt{MoveHand()} is called.

\texttt{MoveHand()} begins by checking if the time since the last movement has been updated. If it has not been updated the distance integral would not change, since no time has passed, so no calculations would have to be made. If the time has changed it creates a new \texttt{Vector3} object, which is Unity’s way to represent a 3D vector. The vector contains the linear accelerations from the Bluetooth controller, but the z-axis is negated to make the coordinate system a left-handed system. Then the quaternion that was calculated in 3.2.6 is used to rotate the vector. The rotated vector is filtered with the threshold, median3, and low-pass which are described in Section 3.3.4. Then the time difference since the last movement is calculated and the trapezoid rule is used to update the velocity integral by adding the trapezoid to the last value. Only the last trapezoid of the distance integral is calculated, it will be added to an integral later. The current values of accelerations and velocities are saved to other variables to be used as starting points for the next trapezoid element. Then the distance trapezoid is scaled by the Movement Speed vector mentioned earlier. After that, the VR hand is moved by adding the distance trapezoid to transform.position. Since the script is attached to the VR Hand, transform is a reference to its orientation, position, and scale.
in the VR world. The transform.position is effectively the distance integral. Lastly, the function checks if Output Filtered Movements is checked and if it is, writes the accelerations and time to the file name specified in Output Filename.

The last step of the Update() function in CustomMovement is to call HandleKeyInput(). This method uses Unity’s Input.GetKeyUp() function. It takes a string input which corresponds to a specific key or mouse button. It returns a boolean true if the key or mouse button was released, between the last frame and this frame. That means a button was pressed since it must have been pressed in order to be released.

If the button ”m” is released, the motion from accelerations is toggled by calling toggleAccelerationToMotion(). This method sets a boolean called ”activateAccelerationToMovement”, which is the same variable that is toggled when ”Activate Acceleration To Motion” is set to true by the inspector. This is useful when developing the pattern recognition, since it may be confusing if the VR hand is controlled from two sources. The method also resets the Recorded Movements file, so that the first line in the file corresponds to the beginning of the movement.

Next, if the button ”x” is released, the method setAlignedRotation() is called, which saves the current quaternion from the Bluetooth controller. This quaternion is used as an offset to align Unity’s coordinate system with the Bluetooth controller’s.

If ”c” is released, the method setPreferredRotation() is called. It saves the inverse of the quaternion \((q_c * q_r)^{-1}\) explained in Section 3.2.6, which is used to align the controller with the user’s preferred coordinate system. It also stores the quaternion \(q_a\) mentioned in Section 3.2.6.

Lastly, the method checks if the ”r” button was released. If it was, the VR hand is reset to its original position and the velocity integral is set to zero.

The methods previously described are also executed by pressing the button objects with the same letter (Section 3.2.2).

There is also a method called ResetVelocity() in CustomMovement, which can be called from objects that hold a reference to the VR hand. This is useful for the pattern recognition script, which will reset the velocity integral when a marker is detected.
3.4 Pattern recognition

3.4.1 Camera

A game object called ExternalCamera was added to the scene. It will show the image from the real world provided by the camera. The image will be treated as the perspective transform of the real world. A script called WebCamTextureToMatHelper was attached to the game object. This script will let the camera image replace the default texture of ExternalCamera. UnityEngine provides an object for handling camera input called WebcamTexture. This class holds static references to all of the current device’s cameras. It holds references to a PC’s connected web cameras and a smartphone’s front and back camera, so either one can be used. When a new image is available it is converted to an OpenCV Mat object by WebcamTextureToMatHelper.

The camera matrix is defined using data from the image. The focal length in x and y are set to the image pixel width or height, depending on which is larger. The principal point \((c_x, c_y) = (\frac{\text{width}}{2}, \frac{\text{height}}{2})\). Then the focal length is scaled by distance, so that the following matrix is acquired

\[
\begin{bmatrix}
    f_x \ast z_c & 0 & c_x \\
    0 & f_y \ast z_c & c_y \\
    0 & 0 & 1 \ast z_c
\end{bmatrix}
\]

Where \(z_c\) is the distance of the ExternalCamera object from the main camera. The camera matrix and the image are then passed to an OpenCV method called calibrationMatrixValues. The method calculates the sensor or film size in mm, which is what a real world camera would project the image onto. It also calculates the field of view \((x_{FoV}, y_{FoV})\) in degrees, the focal length of the lens and the aspect ratio \(\frac{f_y}{f_x}\). These are values that Unity can use to adjust the main camera properties, for example field of view, so that the image fills the screen. This information can then be used to calculate positions from the image and map them into the VR world.

An IP camera server for Unity was also used. It was copied from a Unity project found on Github called SampleUnityMjpegViewer. It has one input which is the URL to the IP camera. An IP camera app can be downloaded to the Android device from Google play store. The app will present the URL from which the stream is available. This made it possible to acquire images from the Android device’s camera via WiFi.

3.4.2 Aruco markers

A script called CameraScriptWithAruco was attached to the ExternalCamera object. Most of the code was borrowed from an example project in the
OpenCV for Unity asset, called ArucoExample, from the file ArUcoWebCamTextureExample. Some parts of the code from ArUcoWebCamTextureExample had to be changed. The image was always expected to be positioned at $z = 1$. Unity's standard field of view is set to 60 degrees. In order to maintain this field of view and have the image fill the whole screen, the image distance to the camera will have to be changed. To adjust for the distance in $z$, $z_c$ (Section 3.4.1) was implemented. This was done in the Start() method. A reference to the VR hand was also added so that the script can update its position and reset its velocity integral.

In the Update() of CameraScriptWithAruco, it first checks if a new image is available. This is done with WebCamTextureToMatHelper. If there is a new image, it is converted from RGBA to RGB by omitting the alpha channel. It is then sent to the Aruco object to be analyzed by using Aruco.detectMarkers(). The method takes five inputs: image, Aruco matrix size, detector parameters, camera matrix and distortion coefficients. The detector parameters are contained in an object provided by the ArucoExample project, the standard values were used in this case. Camera matrix is the matrix mentioned in 3.4.1, which is defined in the Start() method. Distortion coefficient is a matrix that can be used if, for example, a fisheye lens is used. In this case, it was set to zero for no distortion. The method outputs corners, ids, and rejected points. Corners is a list of matrices with the positions of corners of identified Aruco markers. Ids is a matrix containing the ids of identified markers. Rejected is a list of matrices with corners that turned out to not belong to a marker.

The detectMarkers() method uses the steps mentioned Section 2.7.3.2. Once a marker is detected, the corner positions and camera matrix is used to estimate the position and rotation of the marker with Aruco.estimatePoseSingleMarkers(). This method takes four inputs: corners, marker length, camera matrix and distortion coefficients. These are the same matrices and lists mentioned before, except for marker length which corresponds to the length of one side of the marker. In this project, a size of 13mm was used. The method outputs rotation vectors and translational vectors. The information can be used to position the VR hand accordingly. The coordinate system has to be corrected, since OpenCV uses a right-handed system and Unity uses a left-handed. The orientation matrix in (2.20) is a 3x4. The Y axis will be inverted from the left, to change the direction of rotation. Then the Z-axis will be inverted from the right, in order to align it with Unity's system. The marker will be facing toward the user, which means negative z-direction in Unity's coordinates. Therefore it has to be multiplied from the right. To make the rotation from the right, a 4x4 orientation matrix is needed because a 3x4 cannot be multiplied with a 3x3 from the right. Therefore a homogeneous
The orientation matrix is created

\[
[R \ t]_h = \begin{bmatrix}
  r_{11} & r_{12} & r_{13} & t_1 \\
  r_{21} & r_{22} & r_{23} & t_2 \\
  r_{31} & r_{32} & r_{33} & t_3 \\
  0 & 0 & 0 & 1
\end{bmatrix}
\]

And the matrices that will invert the Y and Z axis respectively

\[
Y_{inv} = \begin{bmatrix}
  1 & 0 & 0 & 0 \\
  0 & -1 & 0 & 0 \\
  0 & 0 & 1 & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix},
Z_{inv} = \begin{bmatrix}
  1 & 0 & 0 & 0 \\
  0 & 1 & 0 & 0 \\
  0 & 0 & -1 & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix}
\]

\[
Y_{inv}[R \ t]_h Z_{inv} = \begin{bmatrix}
  r_{11} & r_{12} & -r_{13} & t_1 \\
  -r_{21} & -r_{22} & r_{23} & -t_2 \\
  r_{31} & r_{32} & -r_{33} & t_3 \\
  0 & 0 & 0 & 1
\end{bmatrix}
\]

The rotation is not used to rotate the VR hand in this case, since the rotation is accurately supplied by the MetamotionC on the glove. The \( t \) vector is used to position the VR hand. A scale vector is also acquired by calculating the magnitude of each row in the rotation vector.

To move the VR hand, only the translational vector is used. The rotational vectors have less accuracy than the MetamotionC, so it is more reliable to only use the MetamotionC for rotations. Then Unity’s transform.TransformPoint() is used to convert from the main camera coordinate system to world coordinates. TransformPoint() takes a position vector as input and outputs a vector in world coordinates relative to the transform from which the method was called. In this case, it was the main camera’s transform since it represents the user’s eyes in the VR world, which is the point of origin in the pinhole camera model. Some offsets were used to map different markers to appropriate parts of the hand. They were found by approximating the distance from the VR hands origin to the respective marker and adding the difference to the translation vector.

Lastly, the alpha channel is added again to the image. The image is then rendered on the ExternalCamera object with the transparency setting. This is useful for debugging, because the VR hand will be visible through the partially transparent image of the real world.

### 3.4.3 Whycon markers

The inventors of Whycon markers made their source code available on Github. It is written in C++ and uses OpenCV library. Since Unity can’t compile
C++, the application was translated to C#. First, an object called Many circles detector is created, which holds references to as many Circle detectors as the number of circles the user wants to track. Each Circle detector can only detect one circle, but the Many circles detector keeps track of the position of the detected circles so that the next detector starts from the previously detected circle’s position. Each detector follows the steps mentioned in the theory section in order to detect circles.

A class named ManyCircleDetector, CircleDetector, Circle, and DetectorParameters was created with C#. The ManyCirclesDetector takes four inputs; number of circles, width, height and parameters. Number of circles is how many circles the detector will be able to track, width and height is the width and height in millimeters. DetectorParameters does not take any inputs and has no methods but several fields of floats and ints that represent settings for the detector. There are eleven settings, for example inner and outer diameter, roundness tolerance and minimum size to look for.

A list of circle detectors and a list of valid circles is initialized in the constructor or the ManyCirclesDetector. One circle detector will detect only one circle, Two more lists are initialized for all circles and last valid circles. These will be used for failed circles and a history of valid circles, so that the detector knows where to start. The only important method in this class is detect(). It takes four inputs; an image, reset, max attempts and refine max step. Image is the image that is to be analyzed, reset is to empty the list of past valid circles, max attempts can be used to run the detector several times if no circle was found with different thresholds each time, refine max step is for trying different thresholds even if a circle was found earlier, since not all circles are guaranteed to be found with the same threshold. It is highly dependent on the lighting conditions that the image was shot in. This method will run all circle detectors in its list one by one. Each circle detector keeps track of exactly one circle.

The circle detectors in the list are created with the class CircleDetector. It takes the same inputs as ManyCircleDetector, but also an additional called context. This is an object that keeps track of detected continuous segments, during the flood fill procedure. This class also has a detect() method. The method takes three inputs, the image matrix, a clean up flag and the previous circle that was detected. It starts by defining three pixel categories for black, white and unknown colors. Unknown is the category for pixels that have not yet been classified. A variable called queue is also defined, it will hold the cloud of pixels that the current circle constructed with. This queue will later be used to calculate the covariance. Then the position of the previous valid circle is determined which used as a starting point for the analysis. If the starting point is at the beginning of the image, it means
that the pixels have not yet been copied from memory into the byte array which is needed to do the analysis. It is done by creating a byte array called \texttt{pixels}, which size is determined by calling image.total() multiplied by image.channels(), where image is the OpenCV Mat object of the image from the camera. Image channels are the colors red, green and blue. Image total is the width multiplied by the height. The pixels are retrieved by calling image.get(), which takes three inputs. The first is which row to start from, the second is the column, and the third is the byte array that should be populated with the data. The output is, therefore, the filled byte array called pixels mentioned earlier. After that, the method begins to classify each pixel one by one by calling threshold\_pixel(). It takes two inputs, the index of the pixel being analyzed and the pixel array. The method multiplies the index by three, since each pixel consists of three colors and the array contains the color information for each pixel. Then it sums the three following colors from the array and compares the sum with the threshold value. If it is higher than the threshold, it is classified as white, and black if it is lower. As explained in Section 2.7.1 a mean value of the colors corresponds to pixel intensity. In this case, the threshold value is set to a value in the range \([0, 768]\). This makes the calculation of the mean value redundant, since the sum of all pixels can only ever reach \(255 \times 3 = 765\). The threshold is calculated by a random number in the range \([0, 48]\) multiplied by 16. It is unclear why the original developers chose to do it this way. The calculation is put in a method called change\_threshold().

When a black pixel is found, examineCircle() is called. The method takes four inputs, the image, an empty circle object, index in the pixel array, and outer area ratio. The outer area ratio is calculated from values in the parameter object, but the details of its use will not be covered in this report. Then the flood fill method begins by analyzing and classifying nearby pixels and filling up the queue. The edges of the segments are also stored while executing this process. Lastly, the size and roundness of the segments is determined by the edges, to see if it is within the tolerances defined with parameters.

If all the conditions are met, the method then analyzes the center of the circle. The outer circle was black, so the pixel in the middle must be white in order to be a Whycon marker. If it is white, the flood fill algorithm is used for white pixels, the same way as was done with black pixels. If the white segment also passes the check, the two circles are compared for ratio, which is the last test to confirm that it is indeed a Whycon marker. After that, the size is computed with the covariance matrix as explained in Section 2.7.3.3 and the position is calculated with the edges of the circle. This information is then stored in the circle object.
This way the hand can be moved by calling the list of detectors, then check if one of the detectors holds a valid circle. If it does, move the hand to that circle’s position. In order to use more than one circle, some kind of system to identify each individual circle must be implemented, such as Whycode [14].

3.5 Zero velocity update

As soon as the marker is detected, the velocity is set to zero by calling ResetVelocity() on the VR hand object. This means that the integration of the acceleration along with the integrated noise is reset. This is what the theory section referred to as zero velocity update. Since an acceleration of zero means constant speed, there is no way for the MetamotionC alone to detect if it is moving or not. If the zero velocity update happens at a high enough frequency, the velocity integral will be reasonably accurate.

3.6 Implementation and debugging

3.6.1 Android

The MetamotionC manufacturer provides API (application programming interface) for several operating systems, most important for this project is Android and Windows. The controller prototype will be made for Android, and Windows will be used as the development platform. The API provided is for Universal Windows Platform (UWP), which makes it possible to create apps for several types of devices that are running Windows 10. However, the API for UWP was not working. Because of this, there was no way to get sensor data directly to the PC within the limited time frame of this project. Therefore an Android UDP server app was written in Android Studio. The app can connect to the MetamotionC via Bluetooth and send data over IPv4 to the PC. The data is a comma separated text string containing a quaternion and linear accelerations. The app was written so that a UDP packet is sent as soon as new values are received from the MetaMotionC. The steps are illustrated in Figure 3.4. A UDP client was then created for Unity, so that packets can be received directly to Unity’s editor. Theoretically, this method adds a couple of milliseconds of delay and packets may be lost or arrive in the wrong order. This is not a big problem since this will only be used during development and the amount of erroneous packets is very small. This can be considered a worst-case scenario, it will work better with the finished product.
In order to create the app, a new project was opened in Android Studio. This generates many files, of which the most important is content_main.xml, MainActivity.java, and AndroidManifest.xml. There is also a file called build.gradle, which is used for automatic download of libraries and changing elements of the manifest file.

The content_main.xml file is used to build a user interface. Android Studio has a built in layout editor which includes a preview of the final result. The first element in content_main is of the type constraint layout, which means that its child objects will be able to align themselves with constraint properties. A spinner element was added, it is a list of items which opens as a drop-down menu when tapped. This list is populated with Bluetooth addresses to chose from. Two MetamotionCs were available in this project, so both addresses were added. These address was found by using the smartphone to scan for nearby Bluetooth devices. It was given the properties toLeftOf and toTopOf and they were set to parent. This means that the spinner will be placed at the top left corner of the constraint layout. Then a start button element was added. It will be used to start the connection to the MetaMotionC and the UDP server. It was placed below the spinner and aligned to the left. The next element is an autocomplete text view. This will be used to input the IP address of the receiving device. Autocomplete means that it will suggest IP addresses that were previously used, if it matches the current input in the text field. This text view was placed to the right of the start button and below the spinner. Then three text views were added and placed in a column below the start button. The first was called service_text, which will indicate if the app has access to Android’s Bluetooth service. The other text view was called sensor_text, and is used to indicate if the MetaMotionC is connected. The last text view is called info_text, and it is used to display general information about the state of the app, for example if a connection problem occurred or if everything is fine and it is sending messages.

The MetaMotionC connection will be run in a separate thread from the UI thread. Therefore it will not be possible to update the text views because different threads have by default no way of communicating with
each other. Because of that, three more files were created. The first was named MetaWearCommunication and it will hold all the methods that are needed to communicate with the MetaMotionC. The other file was named MetaWearCommunicationListener, which is defined as an interface class and will be used to pass messages between the threads. The last class is called UpdateTextViewAsync and is an extension of Androids AsyncTask class, which is used to update text field asynchronously. It is not clear why the text field needs to be updated asynchronously, but the app stopped working when any other way was tried. The only text field that could be updated directly was service_text.

The MetaWearCommunicationListener class contains three methods. Two of them will be used to update the text fields sensor_text and info_text respectively. The last method is for debug messages. All of them takes one input which is the text message.

UpdateTextViewAsync takes a view and an activity as input. A view is a general class from which text view is the subclass. UpdateTextViewAsync has two methods, doInBackground() and onPostExecute(). The first is not used in this project since this threads only purpose is to update a text view. They are defined as abstract in the superclass AsyncTask, so both must be present in order for the program to compile. The AsyncTask also has an execute() method, which takes one input. When execute() is called, it first calls doInBackground() and then onPostExecute(). The input used when execute() was called will be passed to onPostExecute(). In this project, the text string that the text view will be updated to is used as input. The text is changed by calling TextView.setText(), which takes the text string as input.

MetaWearCommunication is the class where the sensor data is obtained. It uses three inputs, the current activity, a metawear board object and a metawear communication listener, mentioned above. The metawear board is a C# object provided by the library from the MetaMotionC manufacturer. This class has three important methods, one that initiates the MetaMotionC, one that stops it and one that reads the data that the MetaMotionC acquires. When StartSensor() is called, a request is sent to the MetaMotionC to start gathering quaternions and linear acceleration data. When StopSensor() is called, the MetaMotionC is requested to go into sleep mode. The third method is ConnectToBoard, which uses an IP address as input. The first thing this method does is to create the UDP object needed to relay the data of the MetaMotionC. The IP address is used as input to that object. It will be described in detail later in this section. After that the method establishes a connection to the MetaMotionC. It uses the Bolt library to create a series of asynchronous chain linked tasks. If the board connects successfully, the next task is run which is to configure the board. The board is configured to use
sensor fusion, with an accelerometer range of 2 g and gyro sensitivity of 250 dps. This is the lowest range that the accelerometer supports and the lowest resolution gyro. For a motion, 2 g is very fast. If the range was to be increased, the accuracy of the slower motions would be lessened. The gyroscope is kept at 250 dps because there is no noticeable improvement with higher sensitivity. Then two asynchronous tasks are created to read the data that is gathered, one for quaternions and one for accelerations. When accelerations have been received, both quaternion data and accelerations are assembled in a comma separated string together with the time in milliseconds and sent via UDP. The time is obtained with Android’s System.currentTimeMillis().

The UDP server is also created with an AsyncTask. It is recommended by Android’s documentation to run time-consuming tasks in separate threads. Communication protocols typically block the thread often, for example when waiting to receive a message. Even though this app is only used to send data, the AsyncTask will be used since it is common practice. The UDP server takes an IP address, port number and a message as input. In the doInBackground() method, it opens a new datagram socket, converts the message into a byte array and a datagram packet is created with the DatagramPacket() class. It takes four arguments, bytes to send, length of the byte array, IP address and port number. Then it is sent by calling DatagramSocket.send(), which takes the datagram packet as input. The onPostExecute() method is not used in this case.

MainActivity.java is the file that is first read when the app is opened. It is also the file that will use all the previously described files to make everything work together. The onCreate() method is the first method that the OS calls. It takes one input which is the previous state of the app, which can be used to restore a previous session. The method begins by setting the content xml as layout. Then it creates a service connection, which is how Android lets the app use peripherals like Bluetooth. The service is connected by using a binder, which is provided by the MetaMotionC’s manufacturer. The binder is a subclass of Android’s class IBinder. This will create the connection between Android and the MetaMotionC when bindService() is called, which uses the binder and service connection object as inputs. After that, a metawear communication listener object is created, and the methods are overridden so that the text views are updated asynchronously through the task described earlier. The listener is then passed to the metawear communication object, so that metawear communication can use it to update the text views on the UI thread.

The start button described earlier is what triggers the connection event on the service connection object. When the service connects successfully, a binder is automatically assigned to the app. This binder is an IBinder
object. Therefore the binder is cast into the class of the binder provided by the manufacturer. Android’s BluetoothManager is used to connect to the device by using getRemoteDevice(), which takes an input that is the address of the Bluetooth device. This address is the current address that is selected in the spinner object in the UI. The device object is then passed as an input to getMetaWearBoard(), which is a method in the binder class. The board object that is returned is then used as input in the metawear communication object. Then the metawear communication object’s ConnectToBoard() method is called.

The stop button disconnects the MetaMotionC and stops the Android service so that the app returns to its initial state.

Android has a special class for storing preferences, called SharedPreferences. It is called shared because all the activities in the app can read them, which is not important for this app since it only contains one activity. When an IP address is put in the autocomplete text field, it is stored in the preferences. All the IP addresses that have been used are stored as a comma separated string in the preferences. When the app is started, the preferences are loaded and the string is separated into an array. When a new IP address is put in the field, it compares all the addresses in the array and suggests those that still match. For each number of the IP address that is put in, fewer suggestions will match.

Lastly, a library was made with similar methods as previously described. This library contains only one class called OverrideActivity. It will be used in Unity to connect to the board. In the oncreate(), the service connection is created the same way as it was when the start button was pressed in the app.

A method called retrieveBoard() is added in this activity. It takes a Bluetooth address as input, and creates the metawear board object the same way as prepareBluetoothConnection() did in the app. If the object is successfully created, it means the board is connected and a boolean flag is set to indicate this. Then the Unity application can track the progress of the connection state, if needed.

Then a method called ConnectToBoard() was added, but it does not take an IP as input as the method in the app did. This is because this class will be used by the SensorDataCollector object, so no UDP server is needed, as the app did. The purpose of this method is to start the MetaMotionC and collect data, so the name may be a bit misleading. When the MetaMotionC is successfully started and data is collected, a flag is set to true so that the app created in Unity can track the progress. Then variables corresponding to the quaternion, accelerations and time stamp will be updated while data is collected from the MetaMotionC. These are the variables that can be accessed
from the Unity app.

### 3.6.2 Matlab and Scilabs

Matlab was used to plot sensor data and confirm that the filters were correctly implemented. Also, a Romberg’s algorithm with a floating window was written in Scilabs.

Romberg’s method was implemented as a function with two inputs \((x,y)\) and the Romberg integral table as output, where \(I_{4,4}\) is the most accurate. The inputs lengths must always be nine data points because the function is going to create eight trapezoid segments. A loop calculates the integrals, in the first column of the table mentioned in Section ??, using the trapezoid method. The step size is changed each iteration of the loop and is set to

\[
h = \frac{8}{2^i - 1}
\]

Where \(i\) is the number of iterations the loop has run. This means that the step size is halved each iteration. Inside the loop is another loop that will fill out the rest of the columns, as soon as values in the first column become available. The window works so that when nine data points have been collected, they are sent into the Romberg’s algorithm, then the most accurate result is saved. Then the program waits until nine more have been collected, applies Romberg’s algorithm on them and add the most accurate result to the previously calculated Romberg integral. This way should be equivalent of summing more accurate trapezoids than just using the trapezoid method.

### 3.6.3 Processing

In order to verify the MetaMotionC output, it was necessary to plot sensor data in real-time. A graphing library was used to make a real time plot. A UDP client was written in Java, and implemented in Processing to be able to receive the UDP packets sent from the Android device. The real-time plot can be used to make sure the calibration process on the MetamotionC is complete. The calibration process of the sensors needs the MetamotionC to be turned in different directions, and stay still for a couple of seconds, to map the gravity correctly. When this is done, the offsets will be close to zero. Without the calibration, offsets over 0.3928 \(m/s^2\) were measured.

62
4 Results and discussions

4.1 Prototype

A prototype of a Bluetooth controller has been developed (4.1). It can be used to move a 3D model of a hand in a VR world on an Android device (One-plus One). The tracking is done with the help of a microcontroller mounted on the glove together with an accelerometer, gyroscope, magnetometer, and a BLE module. The camera on the Android device is also used to visually track patterns on the controller. Since the sensor provides an inaccurate estimation that sometimes makes the VR hand move in unpredictable ways, the pattern detector is used to provide a very precise position.

Figure 4.1: The completed prototype of the Bluetooth controller.
4.2 Romberg’s algorithm with a floating window

Five functions with different frequencies and area were tested with Romberg’s algorithm. The tested functions and their result can be seen in Table 4.1. All functions were integrated from 1 to 10000. In all cases, Romberg’s method was around 1000 times slower per sample than the trapezoid method.

<table>
<thead>
<tr>
<th>Function</th>
<th>Analytical solution</th>
<th>Trapezoid method</th>
<th>Romberg’s method</th>
</tr>
</thead>
<tbody>
<tr>
<td>$6 \cdot \sin(x + 5 \cdot \pi)$</td>
<td>-9.00927</td>
<td>-8.9975373</td>
<td>-8.9989831</td>
</tr>
<tr>
<td>$\sin(2 \cdot \pi \cdot x \cdot 50)$</td>
<td>0</td>
<td>0.1266197</td>
<td>0.0986149</td>
</tr>
<tr>
<td>$\sin(2 \cdot \pi \cdot x \cdot 1005) + \sqrt{x}/100$</td>
<td>66666.6</td>
<td>66666.618</td>
<td>66666.655</td>
</tr>
<tr>
<td>$\sin(2 \cdot \pi \cdot x \cdot 777 \cdot 10^6)$</td>
<td>0</td>
<td>0.0139667</td>
<td>0.0107178</td>
</tr>
<tr>
<td>$\frac{x^2 \cdot \sin(2 \cdot \pi \cdot x \cdot 1005)}{10^2}$</td>
<td>-0.184097</td>
<td>11.211810</td>
<td>34.040523</td>
</tr>
</tbody>
</table>

Table 4.1: Functions tested to see if Romberg’s method in a floating window provides a more accurate result over trapezoid method.

Romberg showed an improvement on the third decimal when it was better than the trapezoid method. However, in cases with high frequencies or difficult functions, Romberg’s method gave a worse result. There may be a problem with the implementation. It may also be that Romberg’s method does not work with a floating window. It assumes that the integral that it is improving on is the full integral, not the partial integral which is the case with the floating window. Even if it were more accurate in all cases it would not be suitable for this project. Since the measurement is [m/s], Romberg improved the millimeters on the measurement. However, the accelerometer measurements are inaccurate by several centimeters per second, so the improvement on millimeters is irrelevant. Romberg was also about $10^3$ times slower per sample. Therefore Romberg was not used in the final implementation.

4.3 Linear acceleration

Several measurements of the accelerations were done to prove the theory that there is indeed a random walk. If the conditions were ideal and the sensors were perfect, the graphs should look like Figure 4.2 when the MetaMotionC is moved in the x-direction 0.4 m over a period of approximately 2.5 seconds. The peak acceleration is 0.4 m/s² and the peak velocity is 0.32 m/s. Figure 4.2 shows graphs the ideal motion.
Figure 4.2: Simulated acceleration to position that proves the theory.

Five measurements were taken when the MetaMotionC traveled 0.4 m along the x-axis. Figure 4.3 through 4.6 shows the unfiltered and filtered sensor data and the velocity and position calculated from that data respectively.
Figure 4.3: Simulated MetaMotionC movement 0.4 m along the x-axis.
Figure 4.4: Velocity and position from unfiltered measurements. X represents the velocity. Y represents the position.

<table>
<thead>
<tr>
<th>Graph</th>
<th>Max acceleration ([m/s^2])</th>
<th>Min acceleration ([m/s^2])</th>
<th>Max velocity ([m/s])</th>
<th>End velocity ([m/s])</th>
<th>End position ([m])</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.3937</td>
<td>-1.5546</td>
<td>0.3284</td>
<td>-0.2627</td>
<td>0.1151</td>
</tr>
<tr>
<td>2</td>
<td>2.3372</td>
<td>-1.4098</td>
<td>0.4807</td>
<td>0.0277</td>
<td>0.6516</td>
</tr>
<tr>
<td>3</td>
<td>2.1290</td>
<td>-1.9680</td>
<td>0.5592</td>
<td>0.3310</td>
<td>1.1076</td>
</tr>
<tr>
<td>4</td>
<td>2.0092</td>
<td>-2.0323</td>
<td>0.5230</td>
<td>0.2841</td>
<td>0.7546</td>
</tr>
<tr>
<td>5</td>
<td>1.2767</td>
<td>-1.1379</td>
<td>0.2880</td>
<td>0.0335</td>
<td>0.3402</td>
</tr>
<tr>
<td>(\sigma)</td>
<td>0.4678</td>
<td>0.3781</td>
<td>0.1207</td>
<td>0.2382</td>
<td>0.3830</td>
</tr>
<tr>
<td>(\sigma^2)</td>
<td>0.2188</td>
<td>0.1430</td>
<td>0.0146</td>
<td>0.0567</td>
<td>0.1467</td>
</tr>
</tbody>
</table>

Table 4.2: Important values from the graphs with unfiltered values. The last two rows are standard deviation and variation.
Figure 4.5: The filtered signal of a motion 40cm along the x-axis. The filtered signal looks more like the simulated motion than the unfiltered signal did.
Figure 4.6: Velocity and speed of the filtered signal a motion 40cm along the x-axis. X represents the velocity. Y represents the position. The final position when the acceleration goes down to zero is more consistent than with the unfiltered values, but it is not an accurate representation of the real motion.
Table 4.3: Important values from the graphs with filtered values. The last two rows are standard deviation and variation.

<table>
<thead>
<tr>
<th>Graph</th>
<th>Max acceleration [m/s²]</th>
<th>Min acceleration [m/s²]</th>
<th>Max velocity [m/s]</th>
<th>End velocity [m/s]</th>
<th>End position [m]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.5576</td>
<td>-0.8065</td>
<td>0.2294</td>
<td>-0.1354</td>
<td>0.2426</td>
</tr>
<tr>
<td>2</td>
<td>0.5694</td>
<td>-0.6785</td>
<td>0.4175</td>
<td>-0.03087</td>
<td>0.5681</td>
</tr>
<tr>
<td>3</td>
<td>0.4438</td>
<td>-0.1886</td>
<td>0.3616</td>
<td>0.3020</td>
<td>0.6708</td>
</tr>
<tr>
<td>4</td>
<td>0.3965</td>
<td>-0.3789</td>
<td>0.3078</td>
<td>0.1858</td>
<td>0.3595</td>
</tr>
<tr>
<td>5</td>
<td>0.4622</td>
<td>-0.4563</td>
<td>0.2789</td>
<td>0.04452</td>
<td>0.3614</td>
</tr>
<tr>
<td>σ</td>
<td>0.0749</td>
<td>0.2446</td>
<td>0.0729</td>
<td>0.1733</td>
<td>0.1741</td>
</tr>
<tr>
<td>σ²</td>
<td>0.0056</td>
<td>0.0598</td>
<td>0.0053</td>
<td>0.0300</td>
<td>0.0303</td>
</tr>
</tbody>
</table>

As seen from the Table 4.2 and Table 4.3, the variance of the filtered distances is much lower. That is the expected result, because of the smoothing and median filter. The lowest calculated distance was 0.24 m on the filtered values compared to 0.12 m. The largest distance of the filtered values was 0.67 m and for the unfiltered it was 1.10 m. Even though the accuracy is nowhere near usable for a Bluetooth controller, the filtered values are a lot more consistent than the unfiltered values. The most important role of the accelerometer is to provide a general direction in between the optical pattern recognition signals. The frequency response of the signals was also plotted.
with Matlab, the results can be seen in Figure 4.7 and 4.8.

![Frequency plot for unfiltered signal](image)

**Figure 4.7:** Frequency content of the unfiltered signal. It is evenly distributed with a bit more weight on the lower frequencies. This means that the signal is almost indistinguishable from noise.
The frequency response of the ideal signal would result in a peak at 0.4 Hz. There is a peak in both the filtered and unfiltered signals at 0.9 Hz. This is most likely the frequency of the motion, and the rest of it is noise. It is at 0.9 Hz since the MetaMotionC was moved manually by hand. If it had been moved by a machine it would possibly have given a frequency response closer to 0.4 Hz and with much less noise. However, the controller will be used by humans and not machines so the noise profile is accurate for a normal use case. In the filtered values plot, there is also two more peaks above 5 Hz. Those could be signs of pathological tremors present in all humans. These tremors are usually in the 8-12 Hz range [16]. It is difficult to assess how the smoothing filter affects these specific frequencies’, it may just be regular sensor noise. These frequencies could be eliminated by using an Adaptive Linear Fourier Combiner. It is a filter that uses LMS and assigns weights depending on previous frequencies in order to predict which frequencies should be eliminated [17]. However, that could also reduce the
Bluetooth controllers’ ability to record fast motions. To use the filter, in this case, would probably do more harm than good since the magnitude of those frequencies is so small.

4.4 Translational acceleration

During the project, no way to isolate the translational acceleration was found. Bosch’s algorithm outputs a value for linear acceleration, but it reacts to rotational motions as well. The sensor fusion algorithm uses the accelerometer values to find the downward direction. If a translational movement is done, the downward direction is considered to be changed and the hand rotates. While it should be possible to do the calculations by using sensor fusion for the rotations and restrict the accelerometer vector sum to always equal the earth’s gravitational pull, this should be done in the MetaMotionC software. However, it could not be fit within the time frame of this project.

4.5 Fiducial markers

In order to measure the performance of Aruco and Whycon markers, the average time per frame spent on pattern detection was measured. The marker was placed approximately 20 cm from the camera. The machine that the measurements were performed on had an AMD Phenom II X4 955 CPU clocked at 3 GHz and an AMD Radeon 6870 GPU. The Android device used was equipped with a Quad-core 2.5 GHz Krait 400 CPU and an Adreno 330 GPU.

<table>
<thead>
<tr>
<th>Device</th>
<th>Aruco</th>
<th>Whycon</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>7 ms</td>
<td>130 ms</td>
</tr>
<tr>
<td>Android</td>
<td>20.7 ms</td>
<td>749 ms</td>
</tr>
</tbody>
</table>

Table 4.4: Time spent on detecting patterns per frame. The values are average over a set of 100 samples.

4.6 Sensor fusion

The sensor fusion API sets the accelerometer, gyroscope, and magnetometer to sample at 100, 100 and 25 Hz respectively. It is unclear if this is a limitation on Bosch’s sensor fusion algorithm or its implementation in MetaMotionC. The noise density is measured in dB/√Hz. This means that a higher sampling rate would lead to less noise.
4.7 Bluetooth

Bluetooth low energy is a protocol made for Internet of things products in particular. It is meant to be in sleep mode until it gets a wakeup call and starts to transmit data. When the data has been transmitted it goes into sleep mode again. This is not a suitable protocol for a constant stream of information, as is the case with the controller. It reduces the transmission speed to go into sleep mode after each transmission and it is equal in energy expenditure as a Bluetooth classic device when the BLE is constantly ordered to wake up and transmit information. While the MetamotionC can log data to its internal memory at 400Hz, streaming over BLE is limited to 100 Hz. This means that the more sensors that the user wants to read, the less sample rate each sensor is going to have. In this project, a sampling rate of 50 Hz was used for linear accelerations and sensor fusion data respectively, adding up to a total of 100 Hz.

4.8 Pattern detection

The pattern detection worked very well. It is difficult to measure how precise the markers map into the 3D world, but it looks very accurate to the human eye. Since games and virtual reality is more about how natural it feels, accurate measurements of the mapping are not the most important aspect of this controller. The user’s virtual character may, for example, have arms that are three meters long. Then the mapping would have to be scaled accordingly. The Whycon marker calculations turned out to be much slower than the Aruco markers. This is contradictory to the theory, but because Aruco markers were much faster they were used on the final prototype. The studies that this project was based upon claimed that Whycon was two magnitudes faster than Aruco. The Whycon implementation used in this project was copied from the creator’s source code and translated into C#. While the author says it is using the OpenCV library, it seems that the only part of OpenCV that is used is the Mat object to store images in memory. This means that the flood-fill and other algorithms that are used are still CPU based. This makes sense since the Whycon markers are aimed toward small MCUs that are often not equipped with a GPU. On the other hand, Aruco’s code uses the full capability of OpenCV and uses the systems GPU to run processes in parallel. OpenCV does support the flood-fill algorithm and would probably speed up Whycon significantly. The Whycon code also uses pointers in order to not have to move information in the memory. C# does not support pointers, which means that the byte array first has to be copied to another part of the memory where C# can access it. The process of copying
the byte array alone took 21ms on the PC for a 640x480 image. This means that even if the flood-fill algorithm was used, doing any image processing in C# is not going to work. The Aruco library solved this by having all the image processing in C++ and compiling dynamic link libraries (DLL) that can be used in C#. Since DLLs contain OS specific code, different libraries has to be compiled for each system. This could also be done with Whycon and this could possibly lead to the two magnitudes improvement over Aruco that Whycon’s authors found.

A drawback with the camera is its field of view, which is very narrow. That makes it so that the user must hold the controller very near eye level, which is uncomfortable and unnatural. The field of view could be expanded by attaching a fisheye lens to the camera. There are cheap fisheye lenses on the market, the cheapest being around $1 which claims to expand the field of view up to 170°. However, the edges are very blurry when using such a lens, and the pattern recognition fails even in areas where it was successful without the lens. A better lens or sharpening techniques in software are needed in order to increase the field of view with a fisheye lens.

4.9 Cost

The goal of this project was to make a prototype of a Bluetooth controller that would be reasonably cheap since the VR headsets are only around $10. The critical components of the MetamotionC are around $6 and a fisheye lens will make it $7. The materials for the glove plus the hardware should not cost much more than $10. It is difficult to predict how much more work is going to be needed before the controller will be usable and marketable. A profitability study would have to be conducted before continuing this project. The cheapest usable controller that resembles this Bluetooth controller on the market today is Leap Motion which costs $70. It uses USB connectivity and has to be attached to the VR headset with some kind of adhesive. It uses a unit with several cameras to detect the hands.

Designing the Bluetooth controller for Unity makes it easy to make it accessible to developers through Unity’s asset store. A free, easy to use library can be made for the controller to increase the chances of developers adopting this system.
5 Conclusions and suggestions for future work

5.1 Conclusions

A prototype of a Bluetooth controller for mobile VR headsets has been developed. It consists of a glove, with MetaMotionC board and five Aruco markers mounted on the back of it. MetaMotionC has a nRF52832 SoC and an IMU sensor (a 3-axis accelerometer, a 3-axis gyroscope, and a 3-axis magnetometer), and other sensors that are not used in this project. The nRF52832 has an ARM Cortex-M4 MCU and a BLE module. The IMU is used to track motion and the BLE module is used to connect to a smartphone. The Aruco markers are used to obtain a precise position of the controller, by using the camera of the smartphone.

A VR room had to be created to test the controller. Unity was used to create the room, MakeHuman and Blender were used to create a 3D hand model to be used as an avatar of the Bluetooth controller. The VR room was compiled for Android and the prototype was tested with the mobile VR headset. Unity has support for many different operating systems, including Windows, iOS, and Android which was used in this project. Unity’s Android activity was overridden in order to get access to the MetaWear API, which allows Unity communicate with the MetaMotionC.

Digital filters were implemented for the accelerometer data and they were written in C# so that Unity could use them. Both one-dimensional and three-dimensional filters were written. When the accelerometer is used for position estimation, the 3D hand responds with a significant time lag. This could be caused by the Bluetooth transmission and sensor data processing. The rotational data has a bit less, but noticeable lag.

A game object for communicating with the device’s camera and one for using an IP camera was created. The game object represents the film or a sensor in a pinhole camera. The IP camera object allows a PC version of the VR room to use the Android device’s camera via Wifi. Using a PC version saves a lot of time, since compiling for Android is very slow. OpenCV was used to process the images from the camera. When markers were detected, the pinhole camera model was then used to calculate where the marker was placed if it would have been located in the VR space. Subsequently, the 3D hand was moved to that location. Both Aruco and Whycon markers were tested, and Aruco was faster. According to the inventors of Whycon markers, it should be two magnitudes faster than Aruco. More research should be done to find out under which conditions Whycon is faster. Detection rate should also be taken into consideration. Since Aruco has a more complicated pattern it should be easier to detect Whycon markers. An Android app that can relay
the MetaMotion data via Wifi to the PC version of the VR room was also created.

Processing was used to create a real-time graph. The data is received via Wifi, so the same Android app that is used with Unity can be used with the real-time graph. This is useful for analyzing accelerations and could easily be extended to visualize the effects of filters.

Romberg’s method for estimation of integrals was implemented in Matlab and it did show some improvements, but too small to be significant in this project. The inaccuracy of the accelerometer is too large to estimate the position of the hand, even with Romberg’s method. Therefore it is not used. Instead only the trapezoid method is used for integration of the accelerations, which is used to estimate the position of the 3D hand. While the position is inaccurate, the direction is often correct. Then the camera can make a precise estimate when the hand is moving slow enough to capture a sharp image.

A motion of 40 cm in the x-direction was analyzed. Matlab was used to process the accelerometer data with filters. The filters for linear acceleration did reduce the standard deviation significantly, but the difference of the end position was still over 40 cm for the same motion. The motion was performed manually. A better approach would have been to use a mechanical motion, for example a cart on a tilted rail. That way the motion could have been more consistent. Also, different motions should be analyzed to see if other filter parameters could be better for a broader set of motions.

Most of this project’s time has been spent creating tools and an environment to test the controller. While making a Bluetooth controller with cheap parts is certainly possible, the results of this project were not satisfactory for real-world usage. With the tools created in this project, further research can be made in the details of the controller. With more advanced signal processing, more accelerometers mounted on the controller, better smartphone cameras and simpler markers the performance could be improved.

5.2 Future work

The Bluetooth controller developed in this project could be further improved in many ways. One way is to develop better accelerometers with less noise, but this is very difficult and a lot of research is already going into it. More accelerometers with the same noise levels could also be used to lower the random noise. Using the RMS value would improve the measurement proportional to the square root of the number of accelerometers used, which means that four accelerometers would have half the random noise compared to one. The Bluetooth should be replaced from a low energy into a classic.
This will allow greater transfer speeds without costing much more energy. At the time of writing, Bluetooth 5 was recently released on the market, which allows for both LE mode and classic mode.

The controller could also benefit from better filtering techniques. This is quite difficult since there is no obvious way to design a predictor for hand movements, and trying to fuse the data from the accelerometer and camera would only make sense if the camera was unreliable as well. In this case, the camera is very accurate and if it was not for motion blur it would be enough with just the camera input. If some kind of predictor could be designed, a Kalman filter could be used. The accelerations should in most cases be sinusoidal since each motion must have a beginning and an end. If the recognized motions were restricted to straight lines with set slopes, the predictor might be easier to design. A study of common hand movements expressed in mathematical functions would be useful for future development of the controller.

Using Whycon instead of Aruco would bring the energy efficiency up, provided that the implementation of OpenCV was used more extensively than it is currently. By installing LEDs on the glove or making custom markers that emit light could allow the cameras shutter speed to be lowered. A lower shutter speed reduces motion blur, which will let the pattern detector find its target more often. Since the camera provides more reliable input than the accelerometer, this would be a good way to improve the controller. Aside from the improvements that can be made on the glove itself, better cameras on the smartphones would also improve the performance of the controller. More recent smartphones have been equipped with a black and white camera for better low light performance. Huawei, one of the leading smartphone manufacturers at the time of writing, claims the B&W camera can capture up to 300% more light. This inevitably means that the camera could use a faster shutter speed in normal lighting conditions than the traditional camera would. Since both Aruco and Whycon markers only use black and white images, this kind of smartphone could work better with the controller. Neural networks could probably improve the marker detection, as well as aid in the estimation of the position of a hand in a blurry image.

More research could be done on the algorithms that limit the sample rate. Finding a way to improve the MetaMotionC to support higher than 100Hz transmissions and find a way to increase the sensor fusion output speed would help lower the noise. The BLE connection supports speeds up to 2 Mbps, so the bottleneck should be found in MetaMotionCs software, or the Bosch Sensortec. Increasing the sample rate would make it possible to use the controller on higher FPS games than 50 Hz. While it can still be used for applications with a framerate higher than 50 Hz, the calculations would
be more precise with a higher sample rate since the current implementation assumes new readings each frame. The 100 Hz limitation also prevented flex sensors to be installed for the fingers. Transmitting the flex sensor data over BLE would mean that the linear accelerations and sensor fusion data rate would have to be limited further.

Colliders will have to be attached to all joints, which was not necessary for this early stage of development. Since the flex sensors could not be used, adding colliders for the finger joints was not a priority.

Finding a thinner and more breathable material for the glove would also be required. Even with better materials, hygiene will become an issue after some time. The glove should be made washable, either by encapsulating the electronics or by making the electronics removable.
References


81