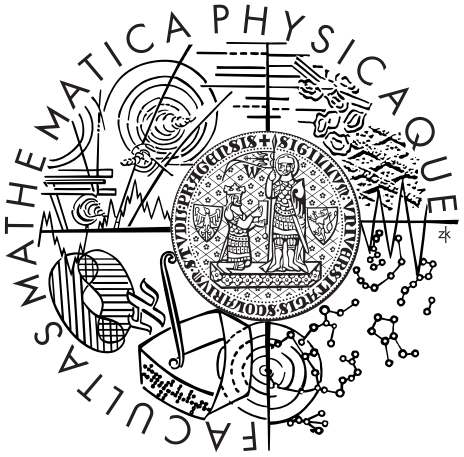
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Filip Matzner

Tracking of 3D Movement

Department of Theoretical Computer Science and
Mathematical Logic

Supervisor of the bachelor thesis: prof. RNDr. Roman Barták, Ph.D.
Study programme: Computer Science
Specialization: General Computer Science

Prague 2014

Hereby, I would like to thank my supervisor Roman Barták for his priceless advice and suggestions, for keeping me motivated through the whole process and for his interest in the success of his students.

Furthermore, I am very grateful to my colleagues Adam Blažek and Adam Vyškovský for the extraordinary amount of time they reserved for me. I shall also mention Zdeněk Rafaj, whose skills in physics helped me to overcome the initial obstacles.

Last but not least, I would like to thank my family for their support during my studies.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Sledování pohybu v 3D prostoru

Autor: Filip Matzner

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: prof. RNDr. Roman Barták, Ph.D.

Abstrakt: V této práci navrhujeme a testujeme metodu pro krátkodobé sledování trojrozměrného pohybu a orientace zařízení, využívající pouze integrované sensory - akcelerometr, gyroskop a magnetometr. Je zde představena přímočará metoda sledování, která je popsána jak z teoretického pohledu, tak pomocí praktického algoritmu. Tuto metodu vylepšujeme stabilizačním systémem, který opravuje chyby způsobené nepřesností sensorů, a to pokaždé, když je zařízení v klidové poloze. Účinnost navržené metody a kvalita zmíněných vylepšení je měřena v několika experimentech se dvěma mobilními zařízeními. Součástí práce je kompletní softwarové řešení umožňující experimentování se sensory v chytrých telefonech pomocí uživatelsky přívětivého rozhraní.

Klíčová slova: inerciální navigace, kombinování sensorů, 3D pohyb

Title: Tracking of 3D Movement

Author: Filip Matzner

Department: Theoretical Computer Science and Mathematical Logic

Supervisor: prof. RNDr. Roman Barták, Ph.D.

Abstract: In this thesis, we propose and evaluate a method for tracking short-term movement and orientation of a device using only its on-board sensors - accelerometer, gyroscope and magnetometer. A straightforward method of motion tracking is described from the theoretical perspective and afterwards transformed into a practical algorithm. To improve its performance, we enhance the method with a stabilization system, which corrects the bias caused by sensor inaccuracies every time the device stands still. The effectiveness of the proposed method and the merits of the enhancement are evaluated in several experiments with two mobile devices. Furthermore, a complete software solution is included, which allows experimentation with smartphone sensors in a user friendly interface.

Keywords: inertial navigation, sensor fusion, 3D movement

Contents

1. Introduction

The technological progress has provided extremely small, lightweight and low-power microchips and thanks to bulk manufactoring and low material consumption the price of these chips has significantly decreased. These days various micro sensors are becoming widely available not only inside modern electronics, but even in cheap low-end devices. The size and weight of these sensors allow them to be mounted in mobile phones, watches, RC models[1] and their more sophisticated versions can be found in airplanes and car airbag systems.

The usual combination of motion sensors present in smartphones consists of an accelerometer, a gyroscope, and a magnetometer. In theory, having a perfect accelerometer and either gyroscope or magnetometer in optimal conditions, it would be possible to track the movement of a device without the need of GPS or similar external positioning systems. The navigation would be working even inside buildings and underwater. However, cheap sensors are far from perfection and the goal of this thesis is to maximize the short interval during which the tracking is reasonably accurate. There exist several projects that track the full 3D motion for longer intervals, but they depend on advanced proprietary or experimental hardware. In addition to the techniques improving the accuracy, we will explain the problematics of inertial navigation from theoretical basics to a full implementation.

1.1 Thesis Outline

At first, we will explain what data does each sensor produce. Then we will describe sensor fusion using pure mathematics and afterwards we will design an algorithm based on this mathematical model. Having the algorithm for perfectly accurate sensors, we will deal with real-world inaccuracies and common pitfalls. Afterwards, we will propose a simple implementation design of this theoretical algorithm. Along with the thesis a software project was developed and it is included on the attached CD. The project consists of a pure C++11[2] library performing the computations, Qt front-end [6][3] used for data visualization and Android [8][4] (Java) application used to read and stream sensor data from a smartphone in real time. At the very end, we include the user documentation of the Qt front-end and the Android application. API[5] documentation of the library itself is distributed with the source code in an HTML form.

Throughout the text we will present data acquired from Samsung Galaxy S III and Huawei U8600 Honor smartphones. We have chosen these two specific devices because the Galaxy phone represents one of the Samsung flagships and the Honor phone represents one of the cheaper devices on the market.

[1] Radio control models such as cars and quadcopters

[2] C++11 is a recent standard of the C++ programming language

[3] Qt is a C++ framework with tools for graphical user interface programming

[4] Android is an operating system from Google for smartphones, tablets and similar devices

[5] Application Programming Interface

1.2 Solution Approaches

There are two possible approaches of fusing sensor data. The first one is to ignore the behaviour of the device (i.e., the way it moves) and completely trust the data received from the sensors. It is called a non-model method. The second one is to create a model of the device behaviour and trust more the data in accordance with the model than the data contrary to the model. It is called a model method. Model methods are particullary useful for objects with a limited means of movement, such as cars and airplanes. For example, when an airplane is flying forwards, it is highly unprobable that it stops in place and starts moving backwards.

Probably the most famous example of the model method is the Kalman filter algorithm. It plays a key role in the inertial navigation of airplanes, vehicles and signal processing in general. It operates on sensor data containing noise and inaccuracies and produces a statistically optimal estimate of variables such as velocity, position, and orientation. The Kalman filter is covered in [7] and [9], however, understanding of this thesis does not require its deeper knowledge.

Our solution focuses on smartphones which can be moved in all directions and arbitrarily rotated, thus not complying to any reasonably strong model. Additionally, in the author's experience, the noise of the sensor data is weak and symetrically distributed and therefore not significantly influencing the result. Because of these observations, we decided to implement a straightforward non-model method based on physics and linear algebra. To eliminate some kinds of inaccuracies, such as sensor drift, we will propose a stabilization method based on the fact that gravity and magnetic field keep pointing the same direction no matter the orientation and position of the device.

1.3 Related Works

As mentioned earlier, most of the recent smartphones (example of such device is depicted in Figure 1.1) are equipped with an accelerometer, a gyroscope, and a magnetometer. The minimal requirements for these sensors are low because they are almost exclusively used for detecting the orientation of the device and not its position. The orientation is afterwards used in games, mobile applications supporting both landscape and portrait view etc. Those sensors in smartphones usually produce hundreds of events per second.

Figure 1.1: Samsung Galaxy S4 Zoom (picture by Samsung Belgium on Flickr)

A recent revolutionary invention (September 2012) is the Oculus Rift [10]

headset (as depicted in Figure 1.2). It produces two separate video feeds for user's eyes and gives him a vision of a 3D virtual reality. More importantly, it traces the orientation of the user's head and changes the view accordingly. To be able to confuse the human brain, the sensor sampling rate is about thousand events per second [12]. Oculus VR has recently (March 2014) been acquired by Facebook [13] for two billion dollars [14] and we might expect a new era of gaming and communication.

Figure 1.2: Oculus Rift (picture by Sebastian Stabinger on Wikipedia)

The headset utilizes the Kalman filter which processes signal from the sensors and, in this specific case, produces an estimate of the orientation of the user's head. In case of the Oculus Rift the model of the device behaviour is called "head-on-a-stick".

Those were examples that only trace orientation. There are a few projects in development that track the full 3D motion of the device, but unfortunately we have not found anything but a vague information available for public (May 2014). For example, Google develops the Project Tango (2013) [11] (a screenshot of the software is depicted in Figure 1.3) and U.S. DARPA develops (April 2013) an inertial navigation microchip for the U.S. Military [15]. The sensors from the project Tango should produce about a quarter million events per second.

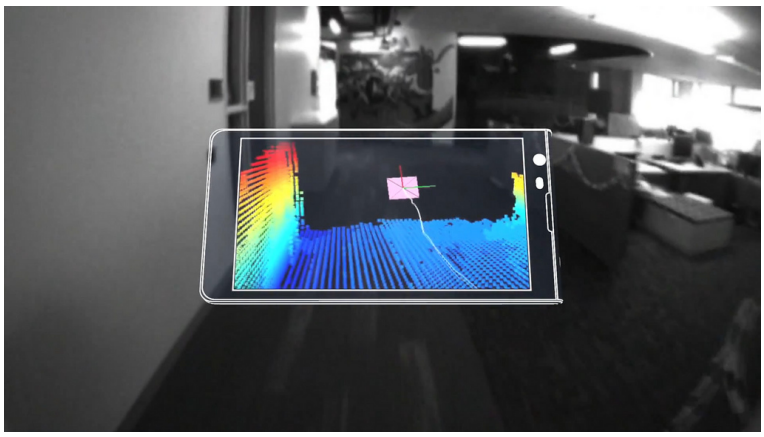


Figure 1.3: Project Tango screenshot (by Ruslan Rugoals on Flickr)

A huge demand for motion tracking devices arises from the gaming industry. The gaming accessory does not usually rely on the inertial sensors only, but uses an external mechanism to continuously calibrate the position and the orientation of the device. A well known example is the Nintendo Wii [18] Remote controller (as depicted in Figure 1.4). It is equipped with an accelerometer, a gyroscope,

and an infrared sensor [19][20]. The Nintendo console box, which has to be visibly placed close to the Remote, has several infrared LED's[6] and the Remote controller uses these LED's to continuously calibrate by a fixed point. The controller has a limited functionality even without the visibility of the LED's. For instance, gesture recogition in sports games can be used to play tennis and golf without the need of pointing the sensor in the direction of the console box.

Figure 1.4: Nintendo Wii Remote (by user Alphathon on Wikipedia)

Less related to this thesis are gaming devices entirely based on the non-inertial technology. For instance the Microsoft Kinect [17], which uses an infrared camera, and the Razer Hydra [16], which uses a generator of a magnetic field.

The last example and the most related to this thesis is the sensor fusion integrated in the Android operating system [8]. Along with access to hardware-based sensors the system provides a few software-based sensors that fuse data from other sensors. For example, Android's orientation sensor uses the accelerometer, the gyroscope, and the magnetometer to determine the orientation of the device. There is also a linear acceleration sensor that attempts to estimate the acceleration of the device uninfluenced by orientation and gravity. That is very similar to what we are trying to achieve and a comparison of our technique and Google's implementation will be presented later (Chapter 5).

[6]Light Emitting Diode

2. Sensors

In this chapter we discuss the principles of sensors and the data they produce. The technology and the manufacturing process will be described very briefly as it is not neccessary for understanding the thesis. The images and concepts of the sensors are strictly orientational.

The most widely used technology of smartphone sensors is called micro-electro-mechanical systems (usually denoted as MEMS). The size of these sensors is of the order of micrometers (10^{-6} meters) and the previously mentioned triple of an accelerometer, a gyroscope, and a magnetometer is usually manufactured on a single chip or at least the sensors are placed very close to each other [5]. For the sake of simplicity of this thesis, the distance between the sensors inside a smartphone will be omitted and we will assume they are in the very same place. The position of the chip inside the smartphone is also omitted and by motion of a device we mean motion of the sensors.

The coordinate axes we use (as depicted in Figure 2.1) are the same as in the Android documentation and the Android API. The order of axes used throughout the thesis will always be x, y, and z.

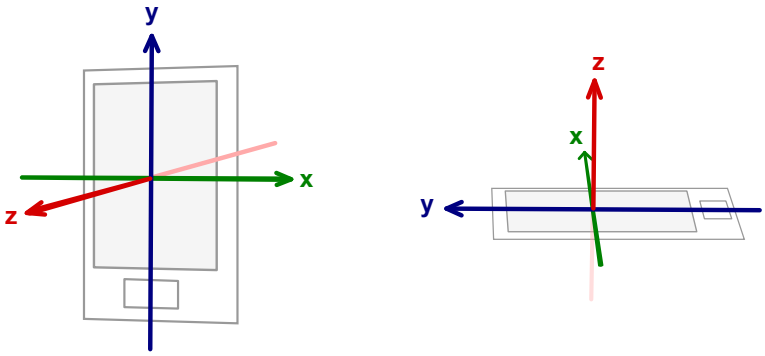


Figure 2.1: Coordinate axes

2.1 Accelerometer

The accelerometer is a sensor that measures acceleration of the device along each of the three coordinate axes. It can be imagined as a heavy ball on springs attached to a frame (a 2D example of such accelerometer is depicted in Figure 2.2). There is no ball in the actual smartphone, but the idea stays the same.

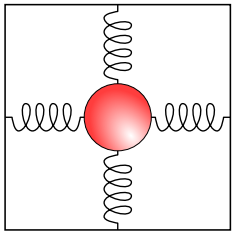


Figure 2.2: 2D accelerometer concept

The ball thanks to it's momentum tends to stay on the same place while the

frame moves, which causes a tension of the springs. In Figure 2.3, there is an example of 2D accelerometer where the frame (i.e., the device) accelerates up and the ball stays down.

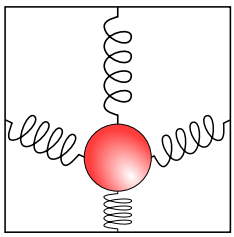


Figure 2.3: Accelerometer accelerating upwards

The tension of each spring is measured and the sensor produces a vector of the device acceleration, usually given in metres per second squared ($\mathrm{m\,s^{-2}}$). In case of the usual 3D (3-axis) accelerometer the vector is, of course, three dimensional. Example of raw accelerometer data is plotted in Figure 2.4

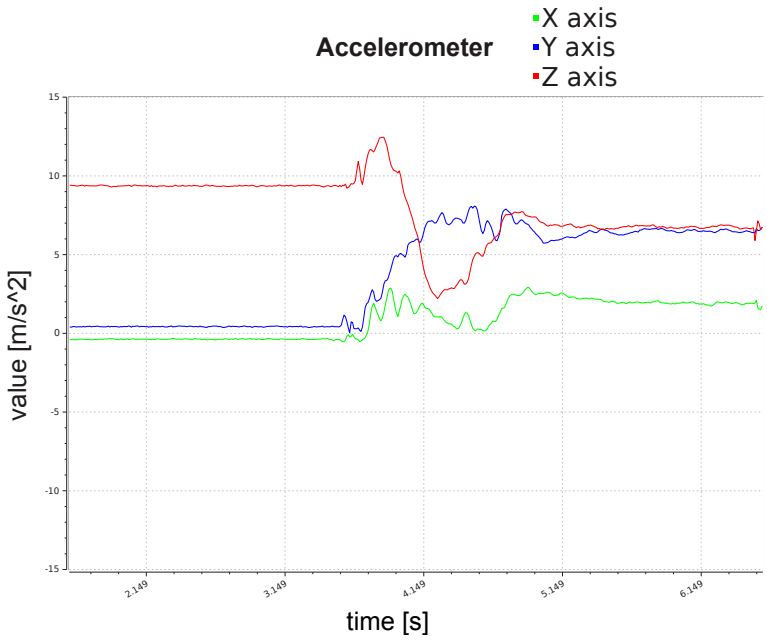


Figure 2.4: Raw accelerometer data from the Samsung Galaxy S III

2.1.1 Gravity

Every accelerometer is affected by a permanent gravity force. The non-intuitive thing is, the force is the same as if the device would be flying upwards (away from the planet). The accelerometer cannot distinguish the ball pulled down by the gravity from the frame accelerating up with no gravity pulling it down (as shown in Figure 2.5).

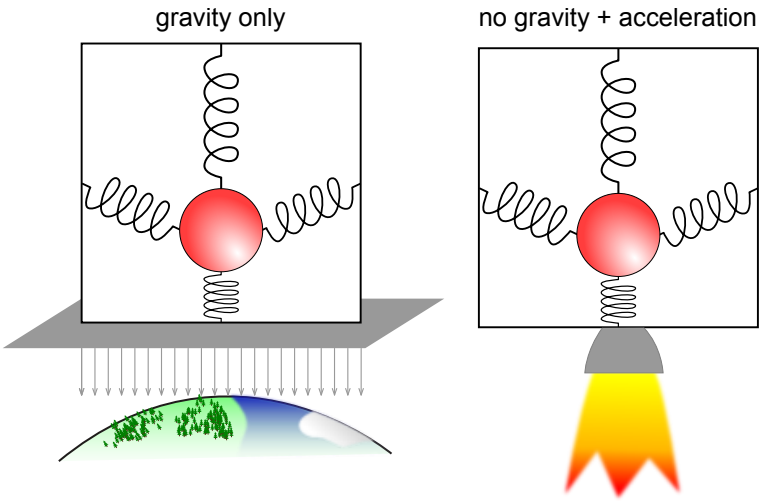


Figure 2.5: Equality of gravity and upwards acceleration

2.2 Gyroscope

The historical concept of a gyroscope, depicted in Figure 2.6, is composed of a spinning disc inside a system of rings. The disc, thanks to its angular momentum, remains rotating around the spin axis regardless of the orientation of the outermost ring (i.e., gyroscope frame). The orientation of the device could be determined from the difference of the spinning disc orientation and the orientation of the rings.

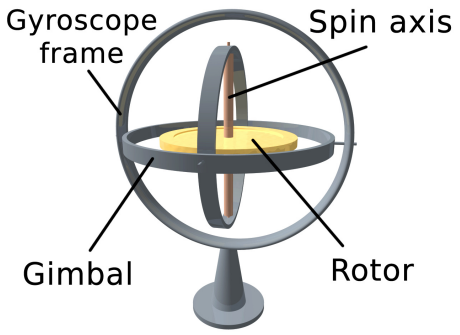


Figure 2.6: Historical gyroscope by Lucas Vieira Barbosa on Wikipedia.com

MEMS gyroscopes present in smartphones are based on a slightly different principle. Instead of a spinning disc, there is a high-frequency vibrating structure which, in contrary to the historical concept, rotates with the frame, but slightly counteracts to the rotation. The strength of the counteraction is measured and interpreted as angular speed of the rotation [2]. It should be noted that in the historical concept, the orientation of the device could be deduced directly from the position of the rings, however, with MEMS gyroscope we only know the current angular speed of the ongoing rotation.

Figure 2.7 is a plot of data produced by a MEMS gyroscope. The three values are rotation rates around the three coordinate axes given in radians per second ($\mathrm{rad\,s^{-1}}$).

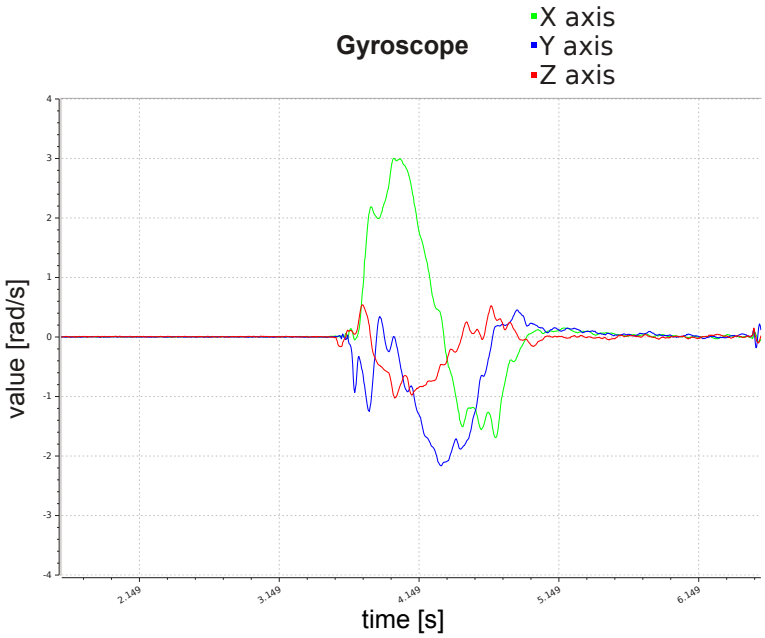


Figure 2.7: Raw gyroscope data from the Samsung Galaxy S III

2.3 Magnetic Field Sensor

There are many technologies of MEMS magnetometers. Widely used are so-called anisotropic[1] magnetoresistant (usually just AMR) compasses which utilize a ferromagnetic[2] material whose electrical resistance depends on its orientation relative to the magnetic field [5].

The magnetometer measures magnetic field vector, i.e., magnetic field strength for each of the three coordinate axes, usually represented in microtesla (µT). It is very sensitive to an interference caused by WiFi, electrical wires etc. An example of the magnetometer data is plotted in Figure 2.8.

[1] properties of anisotropic materials vary with orientation
[2] material which is attracted to magnets

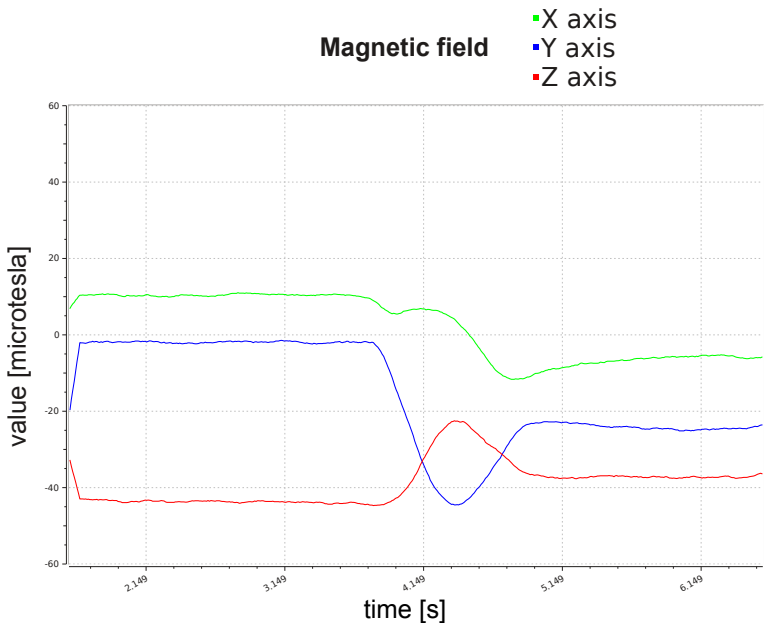


Figure 2.8: Raw magnetic field from the Samsung Galaxy S III

3. Sensor Fusion

The goal of this chapter is to describe the process of fusing the sensor data into the position of the device. The sequence of positions with their corresponding timestamps specifies the movement of the device. At first, we assume that all the sensors produce continuous data and are perfectly accurate (Section 3.1) and after that we will use this knowledge to propose a practical algorithm for discrete sensors (Section 3.2). Having the algorithm, we will discuss real-world difficulties and we will present a stabilization system which improves the accuracy of the algorithm (Chapter 4).

For the sake of simplicity the initial position of the device is $\langle 0, 0, 0 \rangle$ and time begins at 0. Additionally, we assume that at the beginning the device was standing still, i.e., the only force measured by the accelerometer in time 0 was gravity.

Before we begin, we have to define the two coordinate systems we will be working with. A global coordinate system will represent the system the device was in at the very beginning of the process. A device coordinate system, on the other side, is fixed to the device case and rotates with it. Comparison of the two systems is depicted in Figure 3.1. The data produced by the sensors are always relative to the device coordinate system, because the sensors are mounted to the smartphone case.

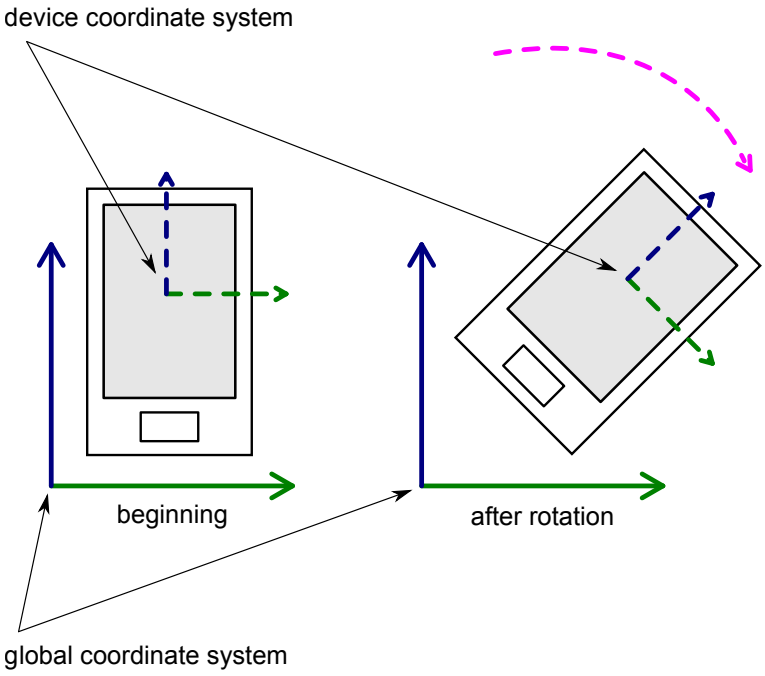


Figure 3.1: Global and device coordinate systems

The data produced by sensors will be defined as following:

- **Accelerometer data**, $\text{acc}(t) : \mathbb{R} \to \mathbb{R}^3$ specifies the acceleration vector in $\text{m}\,\text{s}^{-2}$ in time t including the gravity force. Additionally, $\text{acc}(0)$ denotes gravity only.

- **Gyroscope data**, $\text{gyro}(t) : \mathbb{R} \to \mathbb{R}^3$ specifies the rotation rates (i.e., angular speeds) in $\text{rad}\,\text{s}^{-1}$ around the three coordinate axes in time t.
- **Magnetometer data**, $\text{mag}(t) : \mathbb{R} \to \mathbb{R}^3$ specifies the magnetic field strength in µT along the three coordinate axes in time t.

In this chapter we use integration of vector functions. The result of such operation is again a vector function where the integration is performed piecewise, as shown in Formula 3.1.

$$\begin{aligned} f &: \mathbb{R} \to \mathbb{R}^3 \\ f(x) &= \langle f_1(x), f_2(x), f_3(x) \rangle \\ \int f(x)\mathrm{d}x &= \langle \int f_1(x)\mathrm{d}x, \int f_2(x)\mathrm{d}x, \int f_3(x)\mathrm{d}x \rangle \end{aligned} \tag{3.1}$$

3.1 Theoretical Model

In this section we begin with the simplest possible model where the sensors are perfectly accurate and produce continuous data. This theoretical model will be analyzed using pure mathematics. In Subsection 3.1.1 we assume that the device is not rotating during its motion, Subsection 3.1.2 will define what the rotation is and Subsection 3.1.3 will combine the knowledge together to create a formula for the position of the device.

3.1.1 Position

We will begin with the simplest possible model, where the device is not rotating during its motion. Because the orientation of the device does not change, the device coordinate system will have the same orientation as the global coordinate system for the entire motion. Therefore, we can process the sensor data as if they were relative to the global coordinate system.

Let us remind some basic physics. Velocity is defined as derivative of position with respect to time (Formula 3.2) and acceleration is defined as derivative of velocity with respect to time (Formula 3.3).

$$\begin{aligned} \text{linacc} &: \mathbb{R} \to \mathbb{R}^3 \text{ is acceleration vector in time } t \text{ without gravity} \\ \text{vel} &: \mathbb{R} \to \mathbb{R}^3 \text{ is velocity vector in time } t \\ \text{pos} &: \mathbb{R} \to \mathbb{R}^3 \text{ is position in time } t \end{aligned}$$

$$\text{vel}(t) = \frac{\partial \text{pos}(t)}{\partial t} \tag{3.2}$$

$$\text{linacc}(t) = \frac{\partial \text{vel}(t)}{\partial t} \tag{3.3}$$

However, in our case, the only known variable is the acceleration, thus we will use the definition the other way around. That is, the velocity is the integral of acceleration (Formula 3.4) and the position is the integral of velocity (Formula 3.5).

$$\text{vel}(t) = \int_0^t \text{linacc}(u)\text{d}u \tag{3.4}$$

$$\text{pos}(t) = \int_0^t \text{vel}(u)\text{d}u \tag{3.5}$$

Because the accelerometer sensor also measures gravity, we have to subtract it before the sensor data can be used in the formulas above. We have assumed that the device stands still at the beginning, thus the only value measured by the accelerometer at time 0 is gravity, thus the gravity can be denoted as acc(0). By putting Formulas 3.4 and 3.5 together and removing the gravity from the accelerometer data (Formula 3.6) we will express the position of the device using only the data from the accelerometer (Formula 3.7).

$$\text{linacc}(t) = \text{acc}(t) - \text{acc}(0) \tag{3.6}$$

$$\text{pos}(t) = \int_0^t \int_0^t \text{acc}(u) - \text{acc}(0)\text{d}^2u \tag{3.7}$$

Once we take the rotation of the device into account, the device coordinate system will rotate with the device, thus accelerometer data from different time can be relative to different coordinate systems. Therefore, the data have to be converted to the global coordinate system and after the conversion we can integrate the transformed data the same way we have done in Formula 3.7. To be able to calculate this conversion, we need to know the orientation of the device.

3.1.2 Orientation

Before we define our representation of the orientation, we have to define a 3D rotation matrix [4]. The 3D rotation matrix is used to represent and perform arbitrary rotation in 3D space. If we have a rotation matrix $A \in \mathbb{R}^{3\times3}$ and we want to apply the rotation it represents to a vector $u \in \mathbb{R}^3$, we can simply pre-multiply the vector by the matrix, i.e., Au. The construction of such matrix is, however, a little bit more difficult.

To make the construction easier, we will introduce a known Formula (3.8) for a rotation matrix representing a rotation around an arbitrary axis through an arbitrary angle [4]. For our purposes the slightly simplified version where the axis is a unit vector is sufficient. This formula hardly has an intuitive explanation. However, the procedure that derives it is not that difficult and is clearly explained in [4].

$$R : \mathbb{R}^3 \times \mathbb{R} \to \mathbb{R}^{3\times3}$$
$$v \in \mathbb{R}^3$$
$$R(v,\theta) \text{is a rotation matrix through the angle } \theta \text{ around the unit vector } v$$
$$\text{where } v =< a, b, c > \text{ and } \sqrt{a^2+b^2+c^2} = 1$$
$$R(\langle a, b, c\rangle, \theta) = \tag{3.8}$$

$$\begin{pmatrix} a^2+(1-a^2)\cos(\theta) & ab(1-\cos(\theta))-c\sin(\theta) & ac(1-\cos(\theta))+b\sin(\theta) \\ ab(1-\cos(\theta))+c\sin(\theta) & b^2+(1-b^2)\cos(\theta) & bc(1-\cos(\theta))-a\sin(\theta) \\ ac(1-\cos(\theta))-b\sin(\theta) & bc(1-\cos(\theta))+a\sin(\theta) & c^2+(1-c^2)\cos(\theta) \end{pmatrix}$$

The direction of the rotation is specified by a so-called "right-hand rule", which very informally reads, "wrap your right hand around the axis you want to rotate around with the thumb pointing in direction of the axis. The wrapped fingers show the direction of the rotation". Figure 3.2 demonstrates an example where the x coordinate axis was pre-multiplied by the matrix $R(y, \frac{3\pi}{2})$ and the result is the z axis.

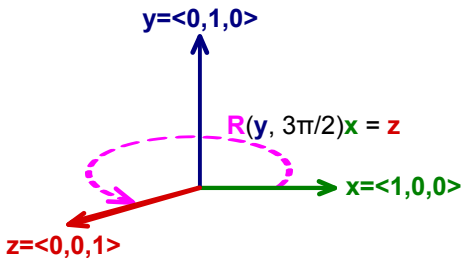


Figure 3.2: Rotation example for $R(\langle 0,1,0\rangle, \frac{3\pi}{2})$

Now, we can define the orientation of the device as the rotation matrix representing the rotation of the device coordinate system relative to the global coordinate system (i.e., relative to the initial orientation). This section will describe how to construct such a matrix using the gyroscope data.

It might be tempting to integrate the raw gyroscope data (i.e., angular speeds) the same way we have integrated the accelerometer as shown in Formula 3.9.

$$\int_0^t \text{gyro}(t)\text{d}t = \ ? \tag{3.9}$$

The result of such integration are three rotation angles along the axes x, y and z. Unfortunately, the rotation composition is not commutative in the 3D space (a counterexample is depicted in Figure 3.3), so we cannot deduce the orientation of the device using just those three rotation angles, thus the result is effectively useless.

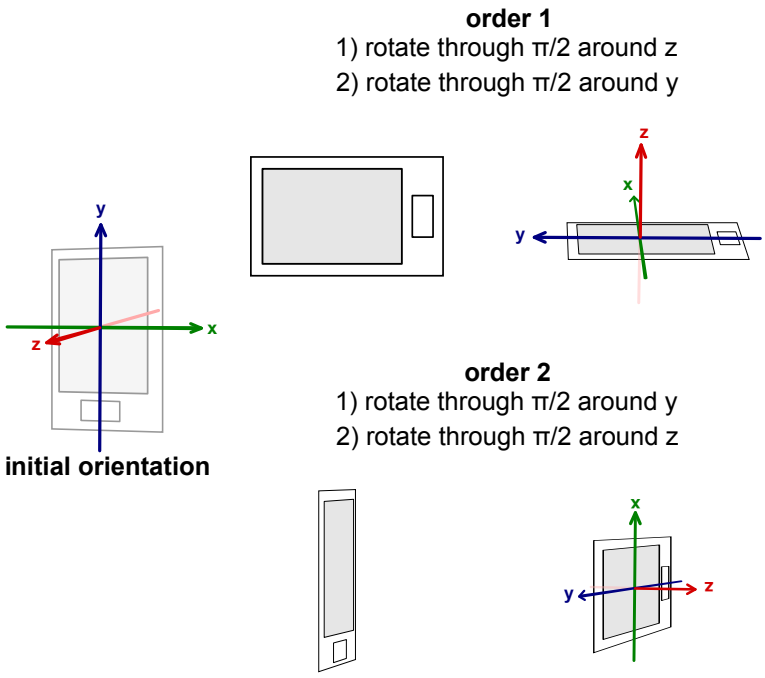


Figure 3.3: Counterexample for rotation commutativity

We will solve this problem by sampling the rotation of the device into a lot of rotation matrices representing rotations during very short time intervals. More formally, the length of the time intervals is getting to zero, thus the number of matrices is getting to infinity. The rotation matrices can be composed by multiplication, that is, when we have a matrix representing a rotation A and a matrix representing a rotation B, matrix AB represents the rotation A followed by the rotation B. Therefore, we can simply multiply the sampled matrices to create a matrix representing the rotation from the beginning to the time t as depicted in Figure 3.4. This composed matrix will be the orientation matrix.

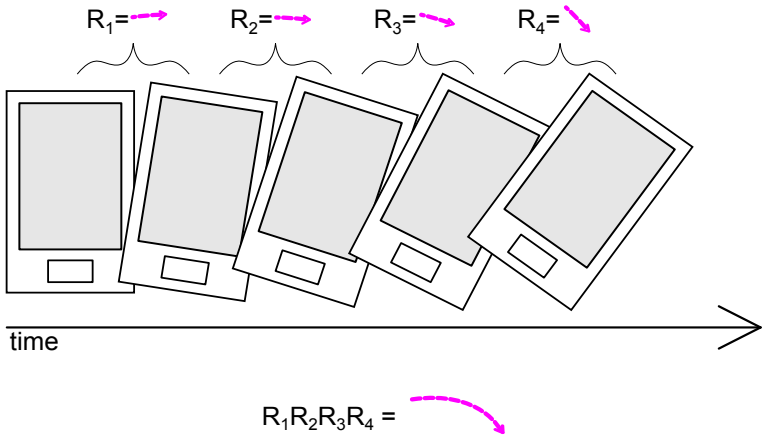


Figure 3.4: Rotation sampling

How can we create such matrices from the gyroscope data function gyro(t)? At first, we need the rotation angles representing rotation during a time interval (t_1, t_2). We can calculate these angles by integrating the gyroscope data and we will denote this vector of rotation angles by the function $\Delta\text{rot}_{\text{ang}}(t_1, t_2)$ defined in Formula 3.10.

$$\Delta\text{rot}_{\text{ang}} : \mathbb{R} \times \mathbb{R} -> \mathbb{R}^3$$
$$\Delta\text{rot}_{\text{ang}}(t_1, t_2) = \int_{t_1}^{t_2} \text{gyro}(t)\text{d}t \tag{3.10}$$

To convert this vector to a rotation matrix, we will use Formula 3.8. But what angle and what axis should we use? Rotating the device simultaneously around the coordinate axes x, y and z by the angles α, β and γ is the very same as rotating the device around the vector $v =< \alpha, \beta, \gamma >$ through the angle $||v||$. Additionally, Formula 3.8 assumes a unit vector, thus we have to normalize the vector v to a unit vector. It can be achieved by dividing it by its own size, i.e., $v_{norm} = \frac{v}{||v||}$. Using these observations and substituting the $\Delta\text{rot}_{\text{ang}}(t_1, t_2)$ to Formula 3.8, we can convert the function $\Delta\text{rot}_{\text{ang}}(t_1, t_2)$ returning a vector to a function $\Delta\text{rot}_{\text{mat}}(t_1, t_2)$ returning a matrix. This conversion is defined in Formula 3.11.

$$\Delta\text{rot}_{\text{mat}} : \mathbb{R} \times \mathbb{R} \to \mathbb{R}^{3\times3}$$
$$\Delta\text{rot}_{\text{mat}}(t_1, t_2) = R(\frac{\Delta\text{rot}_{\text{ang}}(t_1, t_2)}{||\Delta\text{rot}_{\text{ang}}(t_1, t_2)||}, ||\Delta\text{rot}_{\text{ang}}(t_1, t_2)||) \tag{3.11}$$

Now we have got everything needed to be able to define the function that returns the orientation matrix in a given time t. We need to create a rotation matrix out of the gyroscope data as often as possible and multiply these rotation matrices up to the time t. Formula 3.12 mathematically expresses this idea and defines the orientation function.

$$\text{ori} : \mathbb{R} \to \mathbb{R}^{3\times3}$$
$$\text{ori}(t) := \lim_{h\to0} \prod_{i=1}^{\lfloor t/h \rfloor} \Delta\text{rot}_{\text{mat}}((i-1)h, ih) \tag{3.12}$$

where h is the lenght of the time interval between two subsequent samples

3.1.3 The Final Formula

Now we can modify the position function from Formula 3.7 so it takes the orientation into account. We have the accelerometer data relative to the device coordinate system and we have the orientation matrix of this device coordinate system. The only thing we have to do is to pre-multiply the accelerometer data by the orientation matrix to convert them to the global coordinate system. Formula 3.13 defines this modified version of the position function.

$$\text{pos} : \mathbb{R} \to \mathbb{R}^3$$
$$\text{pos}(t) := \int_0^t \int_0^t \text{ori}(u)\text{acc}(u) - \text{acc}(0)\ \text{d}^2u \tag{3.13}$$

This formula will play a key role in the practical approach below.

3.2 Practical Approach

In this section we will discuss processing of real-world data that are neither continuous nor perfect. At first, we will assume that the data are accurate enough and in Chapter 4 we will deal with a reasonable level of inaccuracy.

In the previous section, the theoretical sensors produced data in the form of continuous functions. However, the smartphone operating system instead reads the sensors at irregular time intervals and each reading produces a sensor event. This event is delivered to an application through the OS API[1]. The practical approach will describe an algorithm, which operates on incoming events and updates the latest known state of the device.

Because the theoretical approach has utilized definite integral of continuous vector functions, we need to define a similar operation for sensor events. Let us define a delta integral of a new sensor event to be the difference between the integral before the event and the integral after the event, piecewisely for all the three coordinate axes. The definition is demonstrated in Figure 3.5 and Formula 3.14 expresses the same idea mathematically. It is obvious that a sum of delta integrals is the common integral. As the compromise between accuracy and implementation simplicity, we integrate discrete functions using linear approximation as shown in both Figure 3.5 and Formula 3.14.

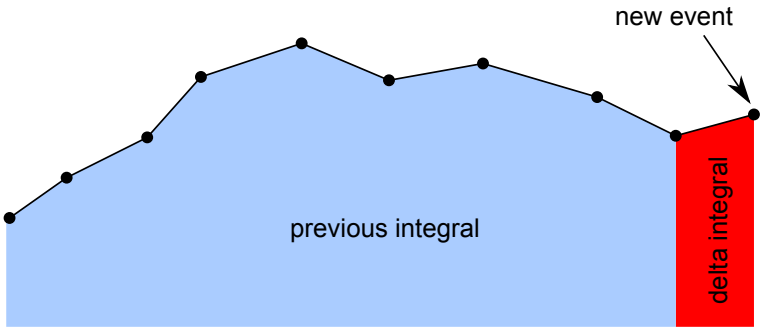


Figure 3.5: Delta integral example

$$
\begin{aligned}
&t_1, t_2 \in \mathbb{R} \\
&\quad t_1 \text{ is the timestamp of the previous event} \\
&\quad t_2 \text{ is the timestamp of the new event} \\
&e_1, e_2 \in \mathbb{R}^3 \\
&\quad e_1 \text{ is the previous event} \\
&\quad e_2 \text{ is the new event} \\
&\Delta\text{int} : \mathbb{R}^3 \to \mathbb{R}^3 \\
&\Delta\text{int}(e_2) = \frac{(t_2 - t_1)(e_2 + e_1)}{2}
\end{aligned}
\tag{3.14}
$$

Let us begin with the description of the sensor fusion algorithm. The algorithm will keep a record of these variables:

[1] Operating System Application Programming Interface

- *orientation* $\in \mathbb{R}^{3\times3}$ is the rotation matrix representing the current orientation of the device. Initially it is the identity matrix.
- *gravity* $\in \mathbb{R}^3$ is the only force affecting the accelerometer at the very beginning of the process.
- *velocity* $\in \mathbb{R}^3$ is the current velocity of the device. Initially $\langle 0, 0, 0 \rangle$.
- *position* $\in \mathbb{R}^3$ is the current position of the device. Initially $\langle 0, 0, 0 \rangle$.

The flowchart of the entire algorithm is depicted at Figure 3.6 and its individual routines will be described in Subsections 3.2.1 and 3.2.2. New sensor events are pushed onto the top of the flowchart and output of each flowchart routine (represented by rectangle) is the input of the subsequent call of the routine. It is basically a set of filters, where each filter modifies, accumulates or drops data which pass through. The implementation of this "pipeline behaviour" in the C++11 programming language will be described in Appendix B - Flowchart Implementation.

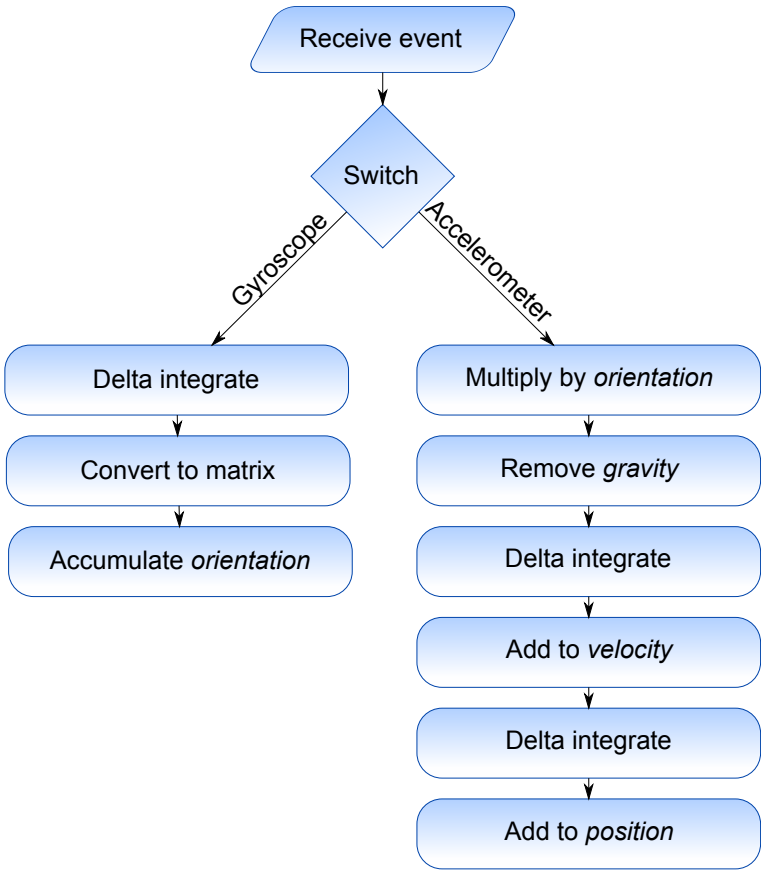


Figure 3.6: Basic flowchart of the process

3.2.1 Gyroscope

The purpose of this branch is to update the orientation matrix. There are three steps that need to be done.

Delta integrate

Input of this routine is a new gyroscope event. The output of delta integration of the event are the three rotation angles, relative to the three coordinate axes, through which the device has rotated between the last reading and the new reading.

It corresponds to Formula 3.10 from the theoretical approach.

Conversion to rotation matrix

This routine converts the vector of three rotation angles around the three coordinate axes to the rotation matrix. This matrix represents a rotation of the device between the last reading and the new reading.

It corresponds to Formula 3.11 from the theoretical approach. An implementation example is shown in Algorithm 1 where the function `rotationMatrix(vector, angle)` denotes the definition of rotation matrix around a unit vector through an arbitrary angle from Formula 3.8.

Algorithm 1 Conversion to rotation matrix

Require:
$rot \leftarrow$ three rotation angles around axes x,y and z

$vector \leftarrow (rot[0], rot[1], rot[2])$
$magnitude \leftarrow \text{sqrt}(rot[0]^2 + rot[1]^2 + rot[2]^2)$
return rotationMatrix($vector/magnitude$, $magnitude$)

Accumulate orientation

The input of this routine is the rotation matrix representing the change in the orientation of the device between the last and the new event. The current orientation matrix can be simply post-multiplied by the input matrix to compose those two rotations together. The composition will be the new orientation.

It corresponds to Formula 3.12 from the theoretical approach. An implementation example is shown in Algorithm 2.

Algorithm 2 Accumulate orientation

Require:
$R \leftarrow$ rotation matrix representing orientation change
$orientation \leftarrow$ orientation matrix

$orientation \leftarrow orientation \cdot R$
return $orientation$

3.2.2 Accelerometer

The purpose of this branch is to convert the accelerometer data to the position vector. As a by-product of the process, the velocity of the device is also calculated.

Multiply by orientation

Before we do anything else, the accelerometer data have to be converted to the global coordinate system (that is the system the device was in at the beginning = before any rotation). That is achieved through pre-multiplying the data by the orientation matrix.

This operation is shown in Formula 3.13 from the theoretical approach. An implementation example is shown in Algorithm 3.

Algorithm 3 Multiply by orientation

Require:
event ← three values of sensor reading
orientation ← orientation matrix

return $orientation \cdot (event[0], event[1], event[2])$

Remove gravity

When we have the acceleration vector in the global coordinates, we can remove the gravity force we have measured at the beginning.

It corresponds to Formula 3.6 from the theoretical approach with pseudocode in Algorithm 4.

Algorithm 4 Remove gravity

Require:
event ← three values of sensor reading in global coordinates
gravity ← gravity vector measured at the beginning

for i from 1 to 3 **do**
    $event[i] \leftarrow event[i] - gravity[i]$
end for
return *event*

Delta integrate & add to velocity

The delta integration combined with addition is the common integration. In this case, we integrate the acceleration of the device (without gravity) in the global coordinates and add it to velocity. Therefore, velocity contains integral of acceleration.

It corresponds to Formula 3.4 from the theoretical approach. A straightforward pseudocode can be seen in Algorithm 5.

Algorithm 5 Delta integrate & add to velocity

Require:
event ← three values of acceleration in global coordinates
velocity ← current velocity

velocity ← *velocity* + *deltaIntegrate*(*event*)
return *velocity*

Delta integrate & add to position

The second integration combined with addition results in the final position. That is the result of the entire algorithm.

It corresponds to Formula 3.5 from the theoretical approach. A straightforward pseudocode can be seen in Algorithm 6.

Algorithm 6 Delta integrate & add to position

Require:
event ← three values of velocity in global coordinates
position ← current position

position ← *position* + *deltaIntegrate*(*event*)
return *position*

4. Stabilization system

In this chapter, we will present typical problems of each sensor type and one possible approach for dealing with them.

4.1 Gyroscope Drift

Even though gyroscope data are usually very precise, the integration creates drift. Partially because of noise and partially because of insufficient sampling rate. With the Samsung Galaxy S III we get about two hundred gyroscope events per second, which is too little to precisely measure the orientation for more than a few seconds. Example of the drift created by ten seconds of a random motion with Samsung Galaxy S III is depicted in Figure 4.1.

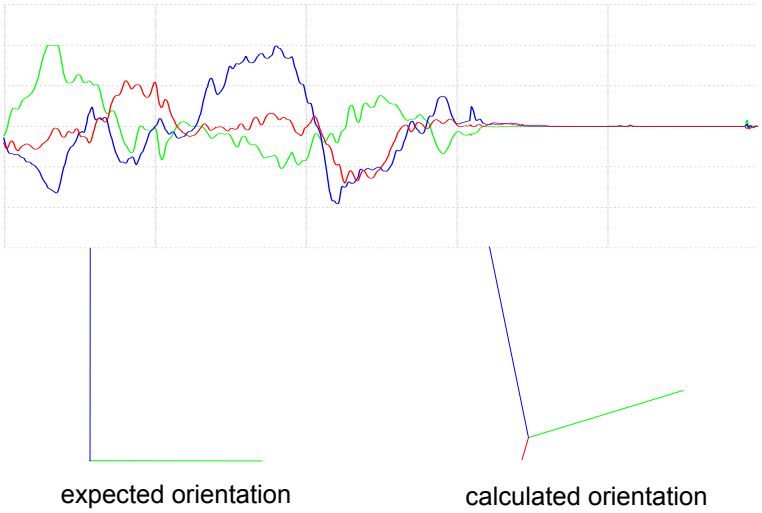


Figure 4.1: Orientation drift after ten seconds of random motion

Why is it such a problem? Because if the device orientation is just a few degrees off, Earth's gravity is removed from a wrong direction. If it creates a false acceleration of, for example, $0.1\mathrm{m\,s^{-2}}$ then after ten seconds it pulls the device 10 meters off (calculation in Formula 4.1).

$$\int_0^{10}\int_0^{10} 0.1\mathrm{d}^2t = 10 \tag{4.1}$$

Fortunately, the Earth's gravity and magnetic field point constant direction no matter the orientation and position of the device. We will use this property whenever the device is at rest, to adjust the orientation by comparing gravity and magnetic field vectors measured at the beginning with the current values of accelerometer and magnetometer.

4.1.1 Stillness Detection

The device is at rest, when it does not rotate and the only force measured by accelerometer is gravity. To detect such a moment, we will measure statistical

variance of the data received from each of the sensors (as defined in Formula 4.2) and if it stays below a specified threshold during a specified time window, we will consider the device to be still. There exists a simple algorithm for online variance calculation [3] that inspects each event only twice, when it enters and leaves the time window. Therefore, it is efficient enough to be employed in our real-time processing pipeline.

We can even improve this process for the gyroscope, because when the device is still, the rotation rate should be as close to zero as possible. Therefore, we can use a simple sum of squares instead of variance (as defined in Formula 4.3).

$$
\begin{aligned}
e_1, \ldots, e_n \in \mathbb{R} &\text{ are magnitudes of the received sensor events} \\
mean(e_1, \ldots, e_n) &= \sum_{i=1}^{n} \frac{e_i}{n} \\
var(e_1, \ldots, e_n) &= \sum_{i=1}^{n} (e_i - mean)^2 \qquad (4.2) \\
sqr(e_1, \ldots, e_n) &= \sum_{i=1}^{n} {e_i}^2 \qquad (4.3)
\end{aligned}
$$

The moment when the device is still is formally defined in Formula 4.4.

$$
\begin{aligned}
a_1, \ldots, a_n \in \mathbb{R} &\text{ are magnitudes of the accelerometer events} \\
g_1, \ldots, g_n \in \mathbb{R} &\text{ are magnitudes of the gyroscope events} \\
m_1, \ldots, m_n \in \mathbb{R} &\text{ are magnitudes of the magnetometer events} \\
W \in \mathbb{R} &\text{ is the user-specified size of the time window} \\
T_a, T_g, T_m \in \mathbb{R} &\text{ are the user-specified stillness thresholds for each sensor} \\
accStill &\iff var(a_{n-w}, \ldots, a_n) < T_a \\
gyroStill &\iff sqr(g_{n-w}, \ldots, g_n) < T_g \\
magStill &\iff var(m_{n-w}, \ldots, m_n) < T_m \\
allStill &\iff accStill \wedge gyroStill \wedge magStill \qquad (4.4)
\end{aligned}
$$

In theory, this technique of stillness detection would not recognize, for example, movement in a straight line at a constant speed. In practice, however, it is very difficult to move a device in a way the data would seem still. The sensitivity of the detection can be further customized by setting the threshold and the size of the time window. The complete list of options is described in the documentation of the software.

4.1.2 Fixing the Orientation by Gravity

When the device is still, the current acceleration vector in global coordinates should match the gravity vector measured at the beginning. If not, we will adjust the stored orientation to make it true.

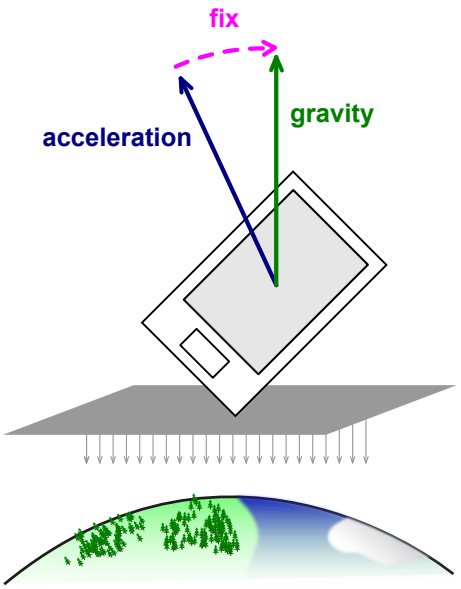


Figure 4.2: Matching the acceleration to the gravity vector

Let's assume that the device is standing still and we have the following data:

- $gra \in \mathbb{R}^3$ is the gravity measured at the the beginning.
- $acc \in \mathbb{R}^3$ is the raw vector currently measured by the accelerometer, including gravity. Since the device is still, it is nothing but the gravity in the device coordinate system.
- $ori \in \mathbb{R}^{3\times3}$ is the current orientation matrix.

Then we can fix the orientation drift in two steps:

1. Create a rotation matrix $fix \in \mathbb{R}^{3\times3}$ that rotates a vector $ori \cdot acc$ (i.e., acceleration in the global coordinates) to match the vector gra (i.e., acceleration at the beginning). Formula 4.5 expresses this idea mathematically.

$$fix \cdot ori \cdot acc = gra \tag{4.5}$$

   But how does a matrix that rotates a vector $a \in \mathbb{R}^3$ to match a vector $b \in \mathbb{R}^3$ look like? We will use the definition of the rotation matrix around an arbitrary unit vector through an arbitrary angle (Formula 3.8), but we need to find the vector to rotate around and the angle to rotate through. A very simple way of finding both is to use a known Formula 4.6 for cross product of two vectors[1].

$$\begin{aligned} &a, b \in \mathbb{R}^3 \\ &\quad \theta \text{ is the angle between } a \text{ and } b \\ &\quad n \text{ is a unit vector perpendicular to the plane generated by } a \text{ and } b \\ &a \times b = \|a\|\|b\| n \sin\theta \end{aligned} \tag{4.6}$$

The cross product is a vector perpendicular to both a and b and its magnitude depends on the angle between a and b, as visualized in Figure 4.3.

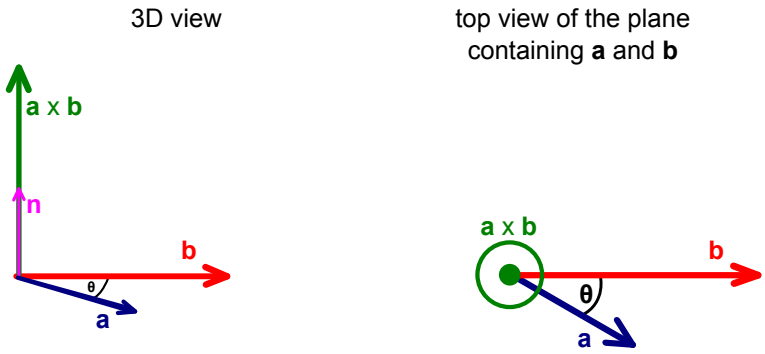


Figure 4.3: Cross product visualization

Furthermore, there is a simple way of determining the value of the cross product using only multiplication and subtraction, as shown in Formula 4.7.

$$a \times b = \begin{pmatrix} a_1 b_2 - b_1 a_2 \\ a_2 b_0 - b_2 a_0 \\ a_0 b_1 - b_0 a_1 \end{pmatrix} \tag{4.7}$$

Having the definition of the cross product, it is not difficult to calculate the vector and the angle to be substituted into the definition of the rotation matrix (3.8). The result is a matrix that rotates a vector a to match a vector b, as shown in Formula 4.8.

$$R_{match}(a, b) = R\left(\frac{a \times b}{\|a \times b\|}, \sin^{-1}\left(\frac{\|a \times b\|}{\|a\|\|b\|}\right)\right) \tag{4.8}$$

In the pseudocodes below, Formula 4.8 will be represented by the function `rotationMatrixMatch(a, b)`.

Having Formula 4.8, we can simply substitute appropriate vectors $ori \cdot acc$ and gra for the a and b to define the matrix fix mentioned in Formula 4.5. The result is shown in Formula 4.9.

$$fix = R_{match}(ori \cdot acc, gra) \tag{4.9}$$

2. The second step to fix the orientation of the device is to pre-multiply the orientation matrix by the fix.

One possible implementation of fixing the orientation by gravity has pseudocode in Algorithm 7.

Algorithm 7 Fixing orientation by gravity

Require:

$gravity \leftarrow$ gravity vector
$acceleration \leftarrow$ current acceleration
$orientation \leftarrow$ current orientation

$globalAcc \leftarrow orientation \cdot acceleration$
$orientation \leftarrow rotationMatrixMatch(globalAcc, gravity) \cdot orientation$
return $orientation$

4.1.3 Fixing the Orientation by Magnetic Field

Even when the orientation of the device is fixed using the gravity vector, it still remains ambiguous as demonstrated in Figure 4.4.

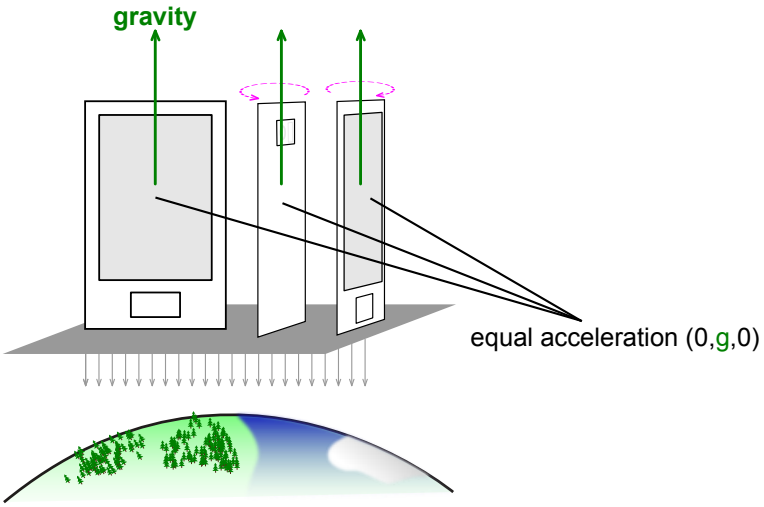


Figure 4.4: The device measures the same acceleration in multiple orientations

To eliminate this ambiguity, we will use the magnetometer sensor. It might be tempting to repeat the same actions as we have done with the gravity fix, but this time with the magnetic field. Such thing, however, could break the previous gravity fix and alter the orientation matrix in a way that the accelerometer data in the global coordinates would not perfectly match the gravity measured at the beginning. In such a case, the gravity would again be removed from the wrong direction and that, as we have previously calculated, is a serious flaw. Therefore, after this process of fixing the orientation by magnetic field, we still require the acceleration vector in global coordinates to point the very same direction as the gravity.

The only way of adjusting the orientation matrix with the persistency of the accelerometer vector in global coordinates, is to perform the rotation around the accelerometer vector itself. To have the fix as accurate as possible at the same time, we will choose the rotation angle minimizing the distance between the current magnetic field vector in global coordinates and the magnetic field vector measured at the beginning. The idea is depicted in Figure 4.5, where the accelerometer vector in global coordinates is denoted as gravity, which is equal, thanks to the previously described fix of the orientation by gravity.

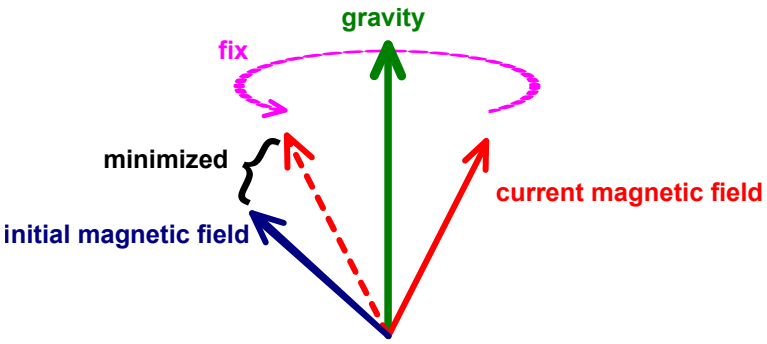


Figure 4.5: Rotating around the gravity vector so that the initial and current magnetic vectors are as close as possible

Such fixing matrix can by constructed using just the tools we already have. Let's assume that the device is standing still, we have already fixed the orientation using gravity and we have the following data:

- $mag_b \in \mathbb{R}^3$ is the magnetic field measured at the the beginning.
- $mag_c \in \mathbb{R}^3$ is the vector currently measured by the magnetometer.
- $gra \in \mathbb{R}^3$ is the gravity measured at the the beginning and the accelerometer vector in global coordinates at the same time.
- $ori \in \mathbb{R}^{3\times3}$ is the current orientation matrix.

The steps to fix the orientation are as follows:

1. Project the mag_b and mag_c vectors in global coordinates to the plane perpendicular to the gravity vector[1]. Denote the results as $pmag_b$ and $pmag_c$. It is equal to viewing the situation from such position that the gravity vector points right towards us, as demonstrated in Figure 4.6.

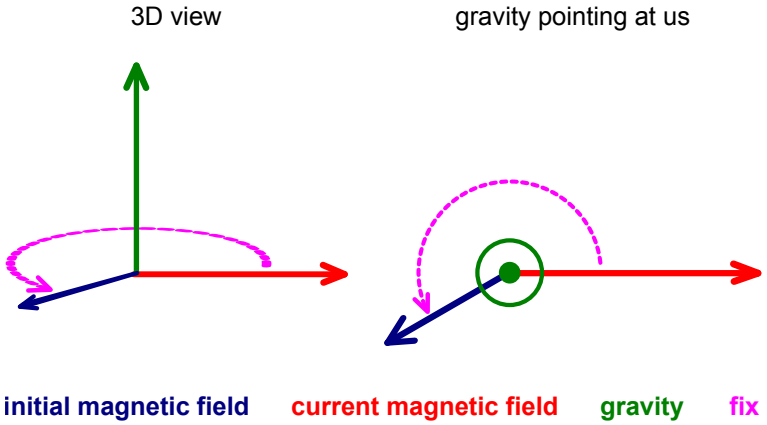


Figure 4.6: The vectors after projection to the plane perpendicular to the gravity

[1]We can achieve the projetion by creating a rotation matrix that rotates the gravity vector to the z axis, then rotate the mag_b and $ori \cdot mag_c$ vectors using this matrix, set their z coordinate to zero and rotate them back. This approach is used in Algorithm 8

2. Create a rotation matrix fix that rotates the vector $pmag_c$ to the vector $pmag_b$. We can observe that the rotation matrix has to represent a rotation around the gravity vector.

3. Pre-multiply the orientation by the fix.

Pseudocode for these steps is shown in Algorithm 8.

Algorithm 8 Fixing orientation by magnetic field

Require:
$gravity \leftarrow$ gravity vector
$initMag \leftarrow$ initial magnetic field vector
$curMag \leftarrow$ current magnetic field vector
$orientation \leftarrow$ current orientation

$gravityToZ \leftarrow rotationMatrixMatch(gravity, (0, 0, 1))$
pre-multiplication is forward rotation
$projInitMag \leftarrow gravityToZ \cdot initMag$
$projInitMag[2] \leftarrow 0$
post-multiplication is reverse rotation
$projInitMag \leftarrow projInitMag \cdot gravityToZ$
$projCurMag \leftarrow gravityToZ \cdot orientation \cdot curMag$
$projCurMag[2] \leftarrow 0$
$projCurMag \leftarrow projCurMag \cdot gravityToZ$
$fix \leftarrow rotationMatrixMatch(projCurMag, projInitMag)$
$orientation \leftarrow fix \cdot orientation$
return *orientation*

After both corrections the orientation of the device is unambigious and the drift should be compensated.

4.2 Accelerometer Accuracy Varies with Orientation

Surprisingly, the gravity we measure at the beginning is about $1.5\,\mathrm{m\,s^{-2}}$ weaker than the gravity measured when the device is turned over (as depicted in Figure 4.7). This stands for both the Samsung Galaxy S III and Huawei U8660 Honor.

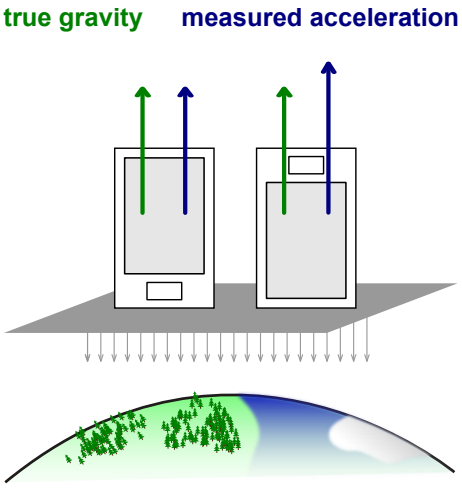


Figure 4.7: Gravity has different magnitude when the device is turned over

Therefore, the accelerometer sensitivity depends on the orientation of the device, which causes a serious deviation of the calculated motion from reality. Figure 4.8 plots a false one metre long movement during a simple turnover of the device.

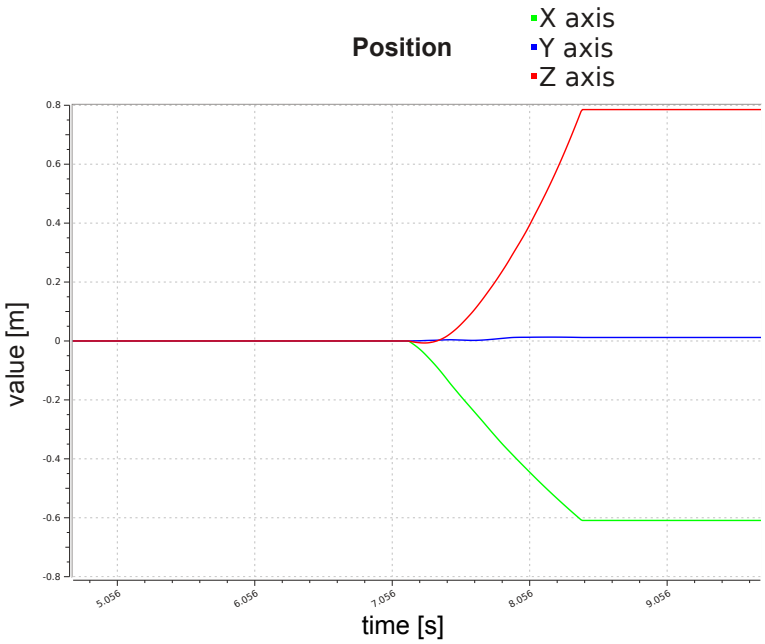


Figure 4.8: False position change when the device is turned over

The most influenced are the motions containing long rotations. To reduce the deviation, it is convenient to not only fix the orientation when the device is still, but to also adjust the magnitude of the stored gravity vector. If done right after the orientation fix, the gravity can be simply replaced by the current acceleration in global coordinate system, as implemented in Algorithm 9.

Algorithm 9 Replace gravity by current acceleration

Require:

gravity ← gravity vector
acceleration ← current acceleration
orientation ← current orientation

gravity ← *orientation* · *acceleration*
return *gravity*

4.3 Magnetic Field Noise and Latency

When the device is moving close to, for instance, a Wi-Fi receiver, mobile phone or a braking tram-car, the magnetic field is significantly unstable. Furthermore, the magnetometer data are a little bit delayed compared to the gyroscope and accelerometer as depicted in Figure 4.9.

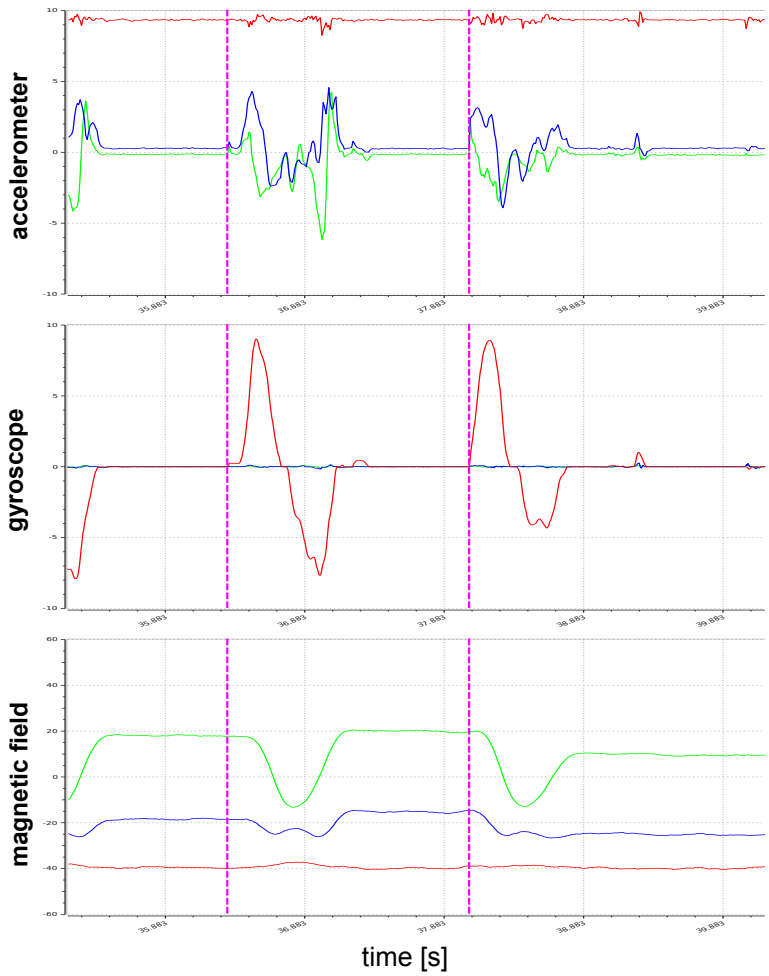


Figure 4.9: Magnetic field delay when device rotated in pulses

To bypass both problems, it is convenient to check the stillness of the magnetometer separately from other sensors to make the stabilization less dependent on the magnetic field. Our solution, for example, checks for the gyroscope and

accelerometer stillness, then it fixes the orientation using gravity and if the magnetometer is also still at the same time, it fixes the orientation using magnetic field. This way even if the magnetic field is fluctuating, the stillness detector can fix the orientation using gravity.

Another problem with magnetometer is, that after some time it loses its precision and has to be recalibrated. We recommend the calibration to be the first thing to do before using the software. There are many ways to perform the calibration on an Android device, one of them is installing GPS Status & Toolbox application from the Android's Play Store [22] and follow its calibration procedure. The sensor should be calibrated in a magnetically stable environment.

4.4 Sensor Initialization Produces Noisy Data

With some devices (e.g. Huawei U8660 Honor) the first few tens of events are very noisy and should be ignored as demonstrated in Figure 4.10.

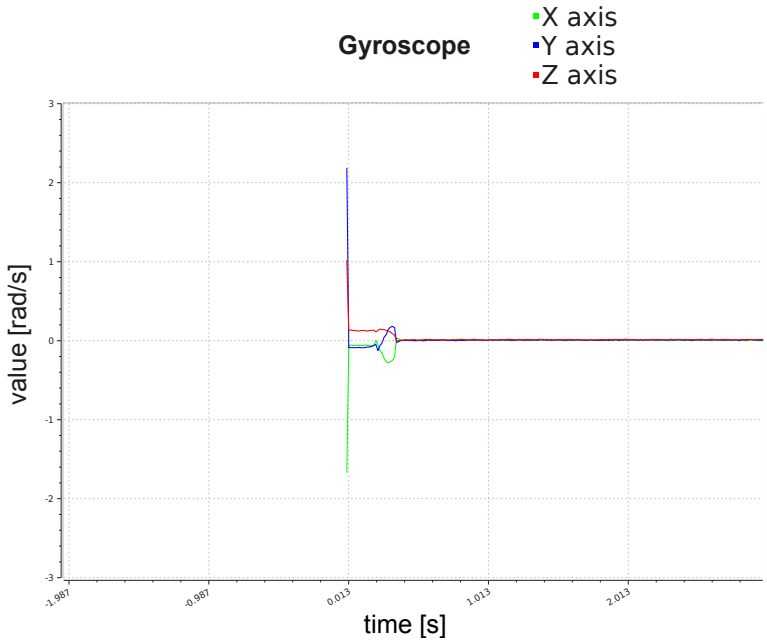


Figure 4.10: Gyroscope initialization noise with Huawei U8660

Because of the noise and unpredictable sensor initialization, the first event received should not be stored as the gravity vector and the same stands for the magnetic field. The solution is to reuse the stillness detection we have already implemented. At the very beginning, we wait until the device is still and afterwards we store the mean of the accelerometer and magnetometer events received during the stillness time interval into the gravity and initial magnetic field respectively.

Another problem with the sensors is, that their behavior changes as they warm up. We suggest to idly read (i.e., read and drop) the sensor data for a few minutes before experimenting with the software.

4.5 Device Is Still But Velocity Is Not Zero

Even when we stop the motion and the device is at rest, the drift has caused that acceleration is not integrated to zero. It can be a serious problem because the integrated acceleration represents velocity and the algorithm behaves as if the device is still moving, as demonstrated in Figure 4.11.

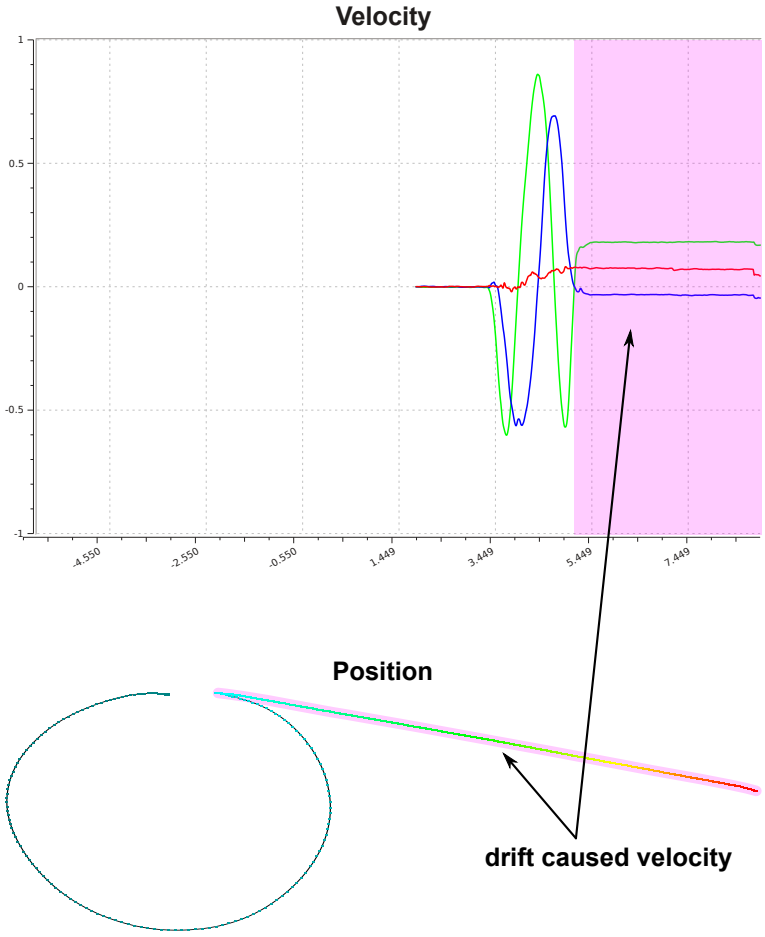


Figure 4.11: Device is at rest, but velocity is not zero

We have solved the problem by setting the velocity to zero, whenever the algorithm detects stillness. It should be noted, however, that in theory this operation eliminates movement in a straight line at a constant speed. As we have mentioned in Section 4, it is not much of a problem in practice.

In addition, for all the sensors, we set event values to zero when the magnitude is below a specified threshold. This operation eliminates in-place oscillation caused by noise of the sensors when the device is still. However, since the noise is very weak and symetrically distributed, the influence on the resulting position is negligible. Settings of the threshold is described in the software documentation.

4.6 Final Flowchart

Considering all the pitfalls mentioned above, we have created an extended version of the flowchart. The first part (depicted in Figure 4.12) works the same way the

basic flowchart does, i.e., each rectangle represents a filter. The `calibrate / fix_drift` routine passes its input to the next routine without modification. It is just a call of a function, whose flowchart (depicted in Figure 4.13) complies to general flowchart rules.

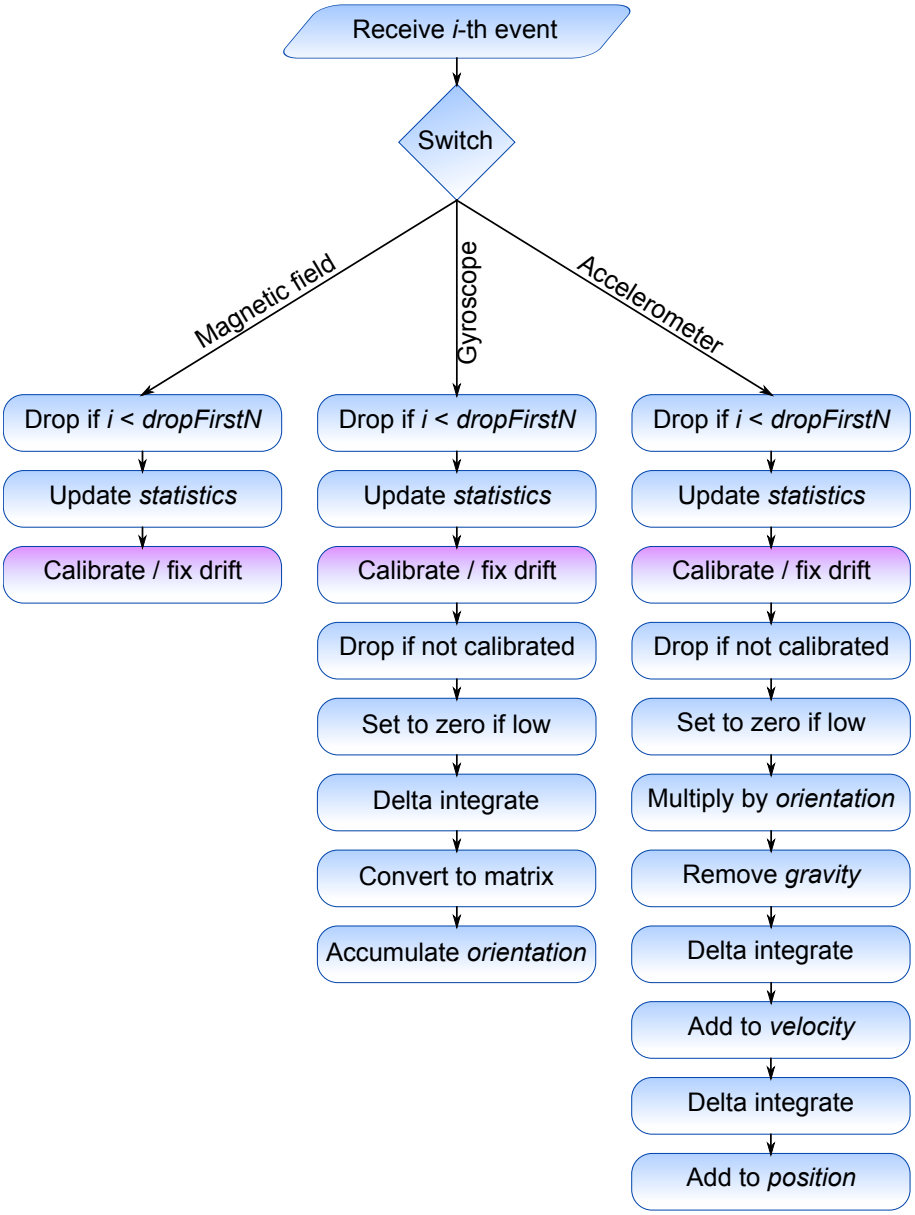


Figure 4.12: Extended flowchart of the process

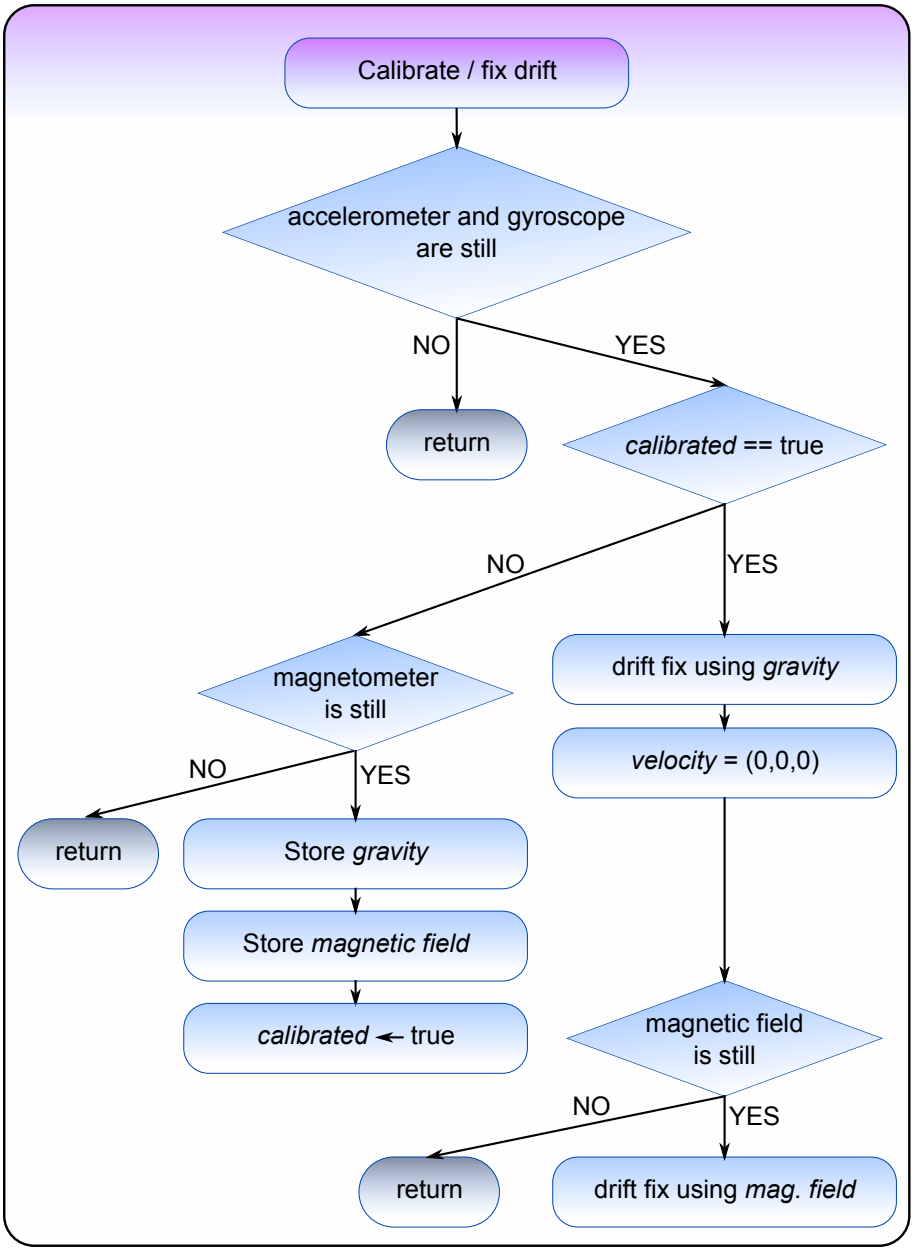


Figure 4.13: Calibrate / fix drift function

5. Experimental Results

In this chapter we will experiment with our implementation of motion tracking and compare the results with the implementation distributed with the Android operating system. In Section 5.1, we will describe the two smartphones we have used for the experiments, including their performance. In Sections 5.2 and 5.3 various simple motions will be demonstrated and we will discuss the accuracy of our implementation. Afterwards, in Section 5.4, we will perform a complicated walk tracking to see its limits. At the very end of the chapter, in Section 5.5, we compare the results to the Android implementation of sensor fusion.

The performed experiments will use all the corrections mentioned in the Stabilization System chapter. The settings of thresholds and window sizes was set for each device and each experiment individually to produce more accurate results. It is convenient to have different constants for an experiment performed with the device laying on a table and for an experiment performed while walking and holding the device in hand.

The situation will always be visualized from two angles, as demonstrated in Figure 5.1. The top view watches the device from the direction of the z axis and the front view watches the device from the direction of the y axis.

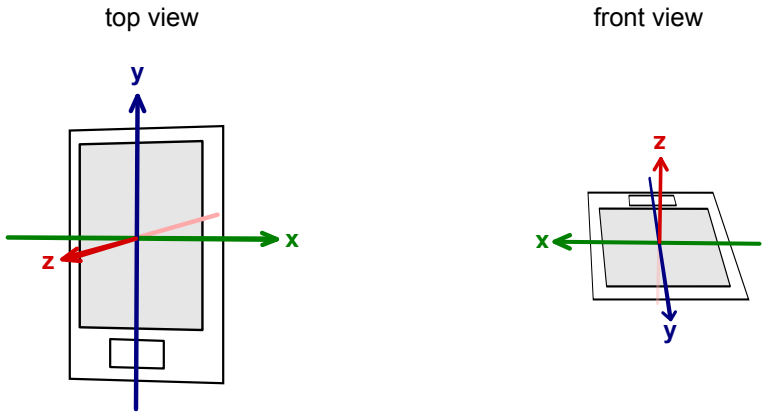


Figure 5.1: Two views of a smartphone laying on a table

The color of each tracepoint will denote the distance from the viewer, with red being the closest and purple being the furthest. The entire spectrum is depicted in Figure 5.2.

Figure 5.2: Distance spectrum

5.1 Devices

The experiments will be performed on two different smartphones using the TrackMe Sensors Android application, which is attached to the thesis. We have used the following devices for their significant difference in price and performance:

- **Samsung Galaxy S III I9300** was a Samsung flagship until 2013, when it was superseded. At the time of launch, the price of the phone was one of the highest on the market.
- **Huawei Honor U8660** released in 2011 is one of the cheaper devices, manufactured by a fast growing Chinese company.

Since both devices have different technical specifications, they produced a different number of sensor events per second. The comparison is shown in Table 5.1.

	Samsung Galaxy S III	**Huawei Honor U8660**
accelerometer	105	52
gyroscope	206	61
magnetometer	102	95
sum	**413**	**208**

Table 5.1: Comparison of number of events per second for the two smartphones

Not surprisingly, the cheaper phone has a lower reading rate compared to the more expensive one.

5.2 Circle

In this experiment, the device will lay on a table and we will move it in the shape of a circle. The result is depicted in Figure 5.3.

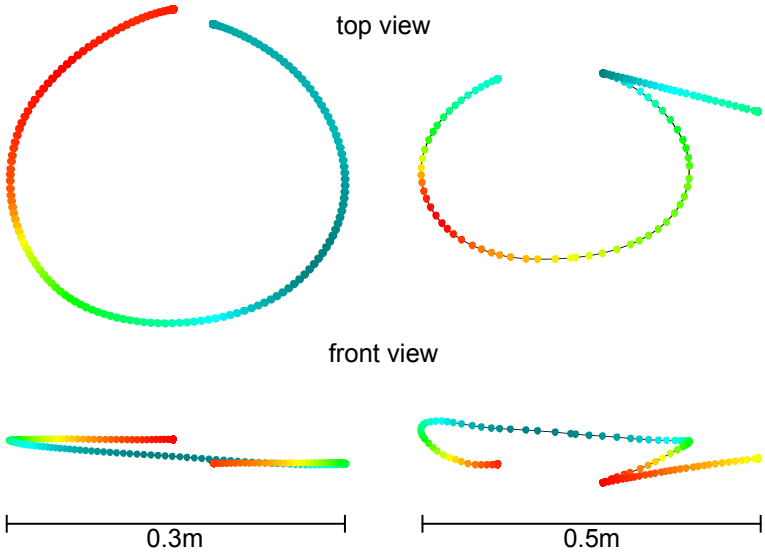


Figure 5.3: Circle shape - 30cm in diameter - Galaxy S III (left) and Huawei Honor (right)

Let us explain the colors by examining the Figure 5.3. Looking at the front view of the Galaxy motion, the beginning and the ending of the circle are painted red. It means that watching the device from the front, it was closest to the viewer at the beginning and ending of the motion. Looking at the top view of the same experiment, the beginning of the circle is red and the ending is blue.

Therefore, the device was moving further off from the top viewer because the software inaccurately calculated that it fell a few centimeters down through the table.

The movement took less than three seconds and the result is surprisingly accurate. Especially for the Galaxy phone, the shape is smooth from both views. Let us prolong the motion duration and draw the same circle three times in a row. The result is drawn in Figure 5.4.

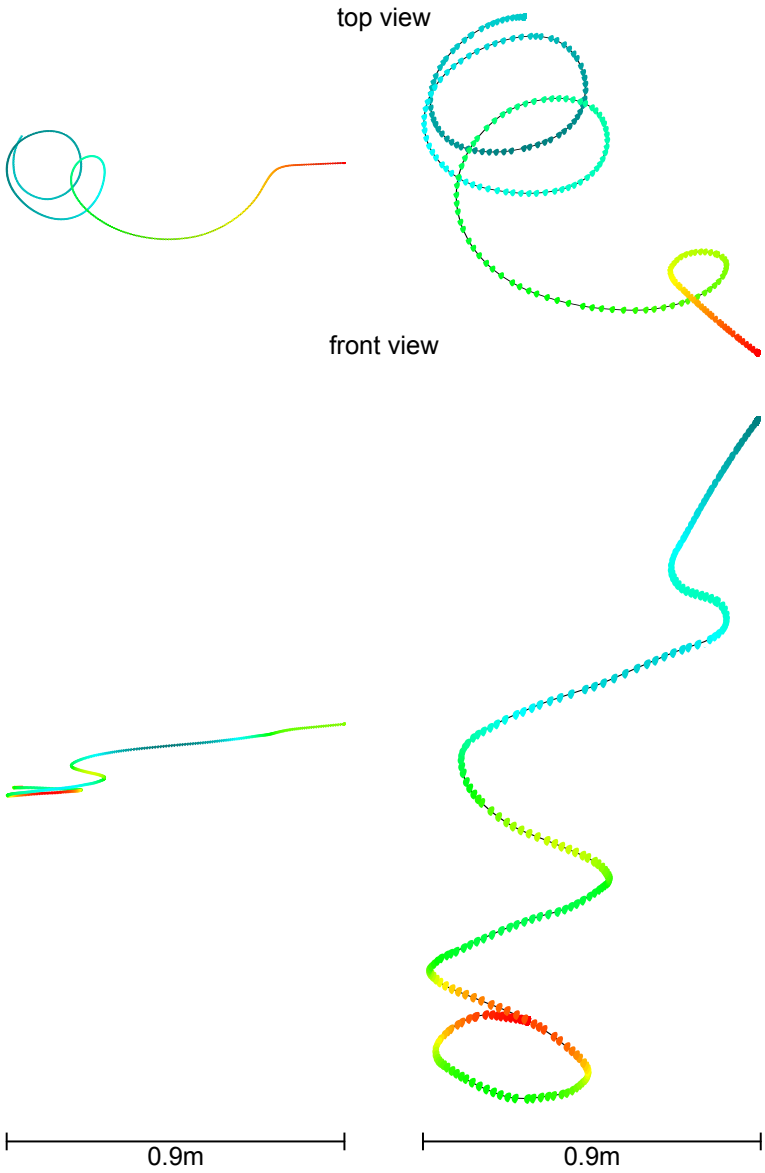


Figure 5.4: Three circle shapes - Galaxy S III (left) and Huawei Honor (right)

When the motion duration prolongs, the shape can no longer be recognized because of the gyroscope drift and consequent false acceleration. The first circle is clearly visible but even the next one is off by a diameter. The reason is, we have not given the stabilization system a chance to fix the drift and the result would actually be the same even if the stabilization would not be implemented at all. To see its benefits, we will insert a small pause (cca. one second) after each circle. During this pause, the stabilization system detects stillness and updates

the orientation matrix. Figure 5.5 shows the same three circles with two pauses in between. The motion took twelve seconds.

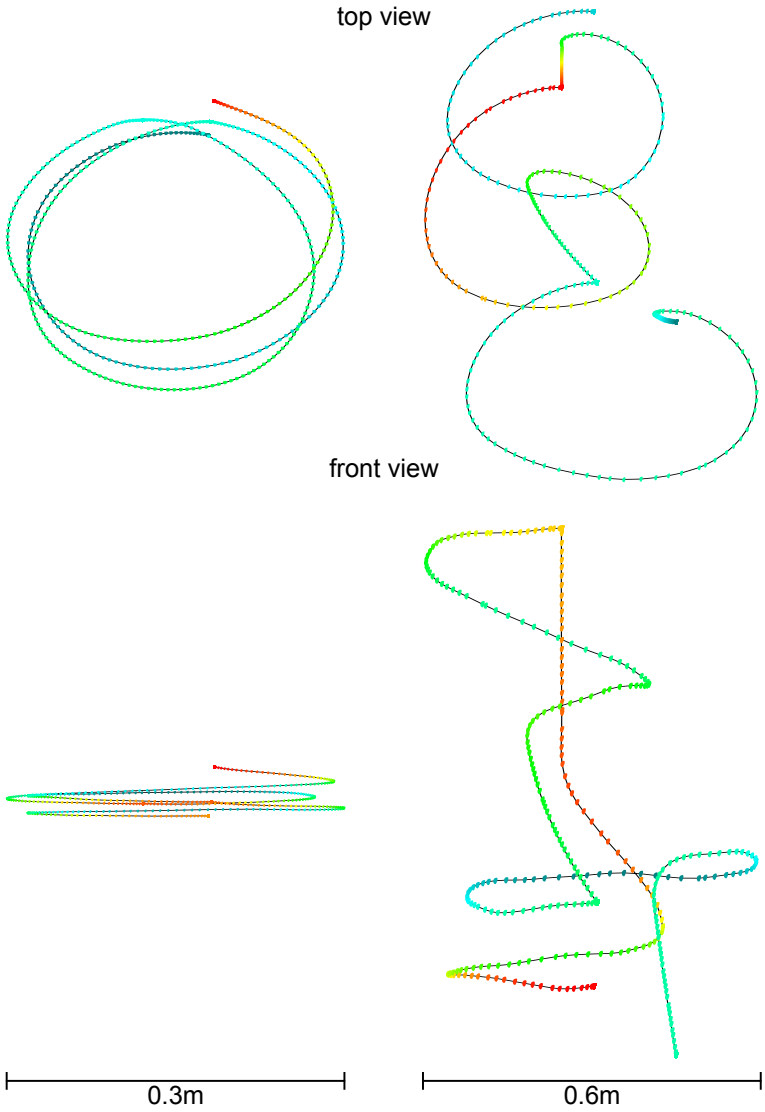


Figure 5.5: Three circles with pauses - Galaxy S III (left) and Huawei Honor (right)

The Galaxy phone, again, draws a smooth and accurate picture. It has all the circles clearly visible and the starting position is almost the same as the ending position. The Huawei device, however, still suffers from the gyroscope drift. Surprisingly, the drift is in the direction of the z axis (i.e., up and down) even though the device was laying on a table.

To compare this result to the case where the stabilization system is turned off, we have performed the same experiment once again, but we have disabled all the corrections introduced in Chapter 4. The drawing is demonstrated in Figure 5.6.

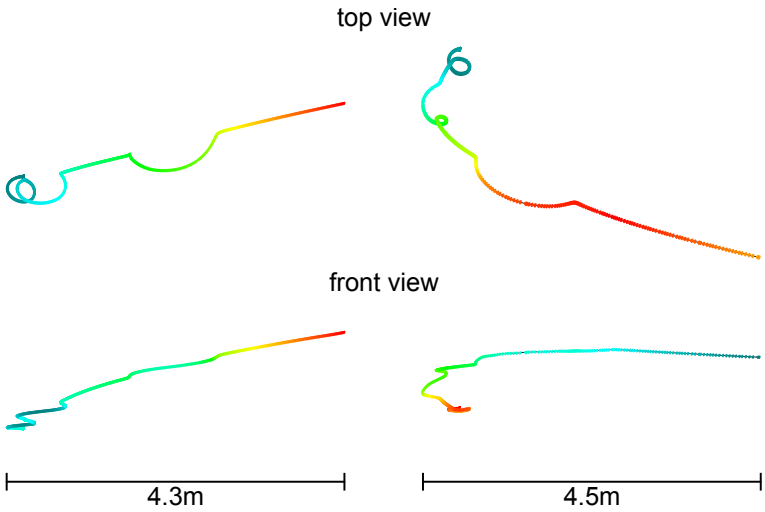


Figure 5.6: Three circles with pauses and disabled stabilization system - Galaxy S III (left) and Huawei Honor (right)

Not surprisingly, the results are even worse than in the experiment without pauses (as shown in Figure 5.4). The double integration of false acceleration introduces a drift whose strength rises quadratically with time. The last circle was drawn after twelve seconds of motion and ended up as nothing but a little curve on a fast nonexistent movement.

All the previous motions had very little rotation included. Therefore, in the next experiment, we will draw the same circle as before but the device will be turning around during the entire motion, as depicted in Figure 5.7.

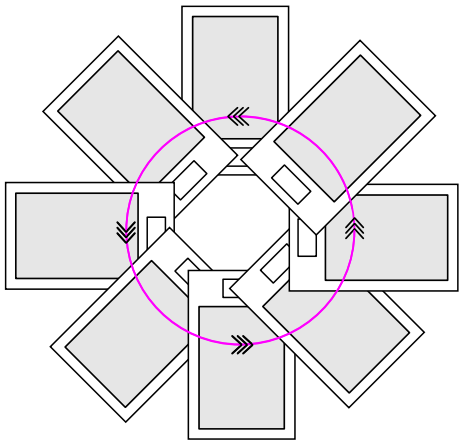


Figure 5.7: Circle with rotation - sample

The result is demonstrated in Figure 5.8.

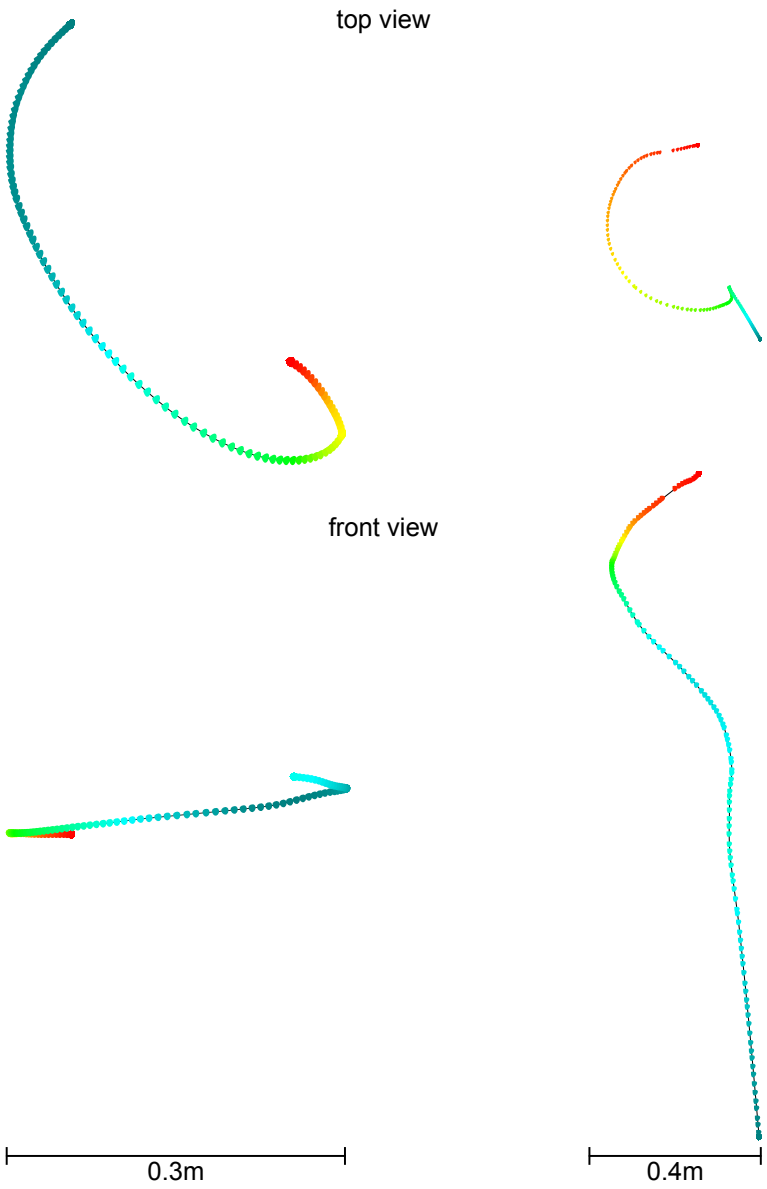


Figure 5.8: Circle with rotation - Galaxy S III (left) and Huawei Honor (right)

It is clear that with added rotation, the results rapidly decline. In the author's opinion, the blame is on accelerometer, whose sensitivity depends on the direction of the acceleration vector.

5.3 Square

In this section, we will draw a square shape to experiment with sharp edges. The result is demonstrated in Figure 5.9.

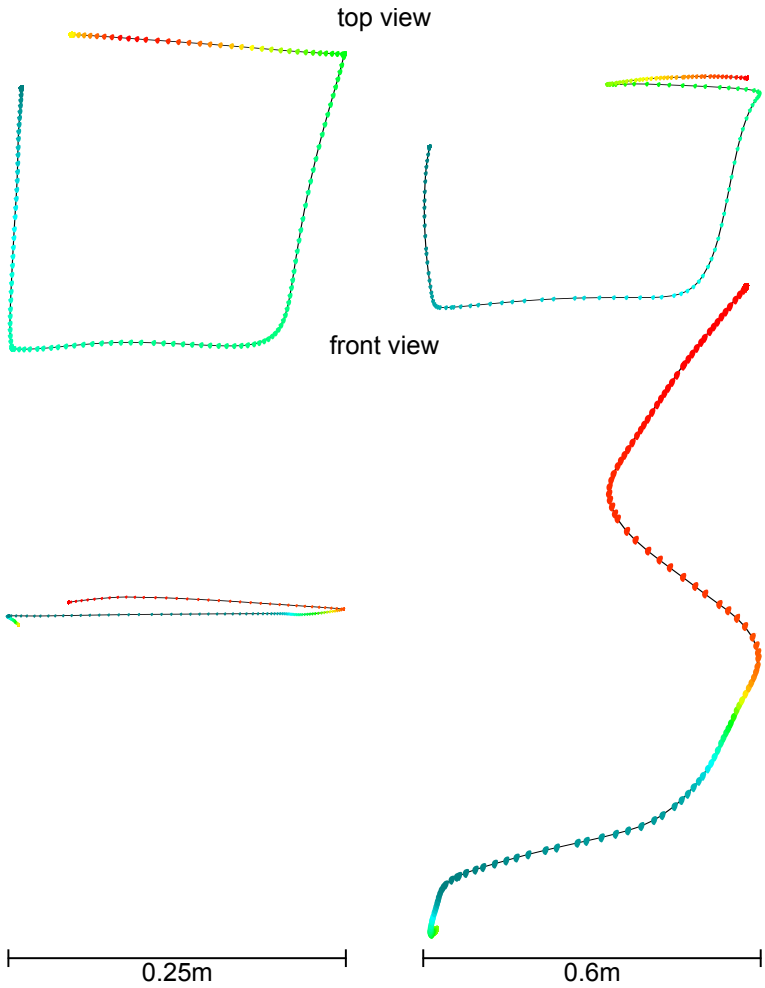


Figure 5.9: Square shape - 30cm on side - Galaxy S III (left) and Huawei Honor (right)

The Galaxy phone creates a smooth shape. The Honor phone data, however, drift again in the direction of the z axis, in which the device has not moved at all.

5.4 Walk

The last motion will be a walk. The author will hold the phone in his hand, will watch its display and will walk forward (i.e., in the direction of the y axis) through a seven metres long horizontal path. The result is depicted in Figure 5.10.

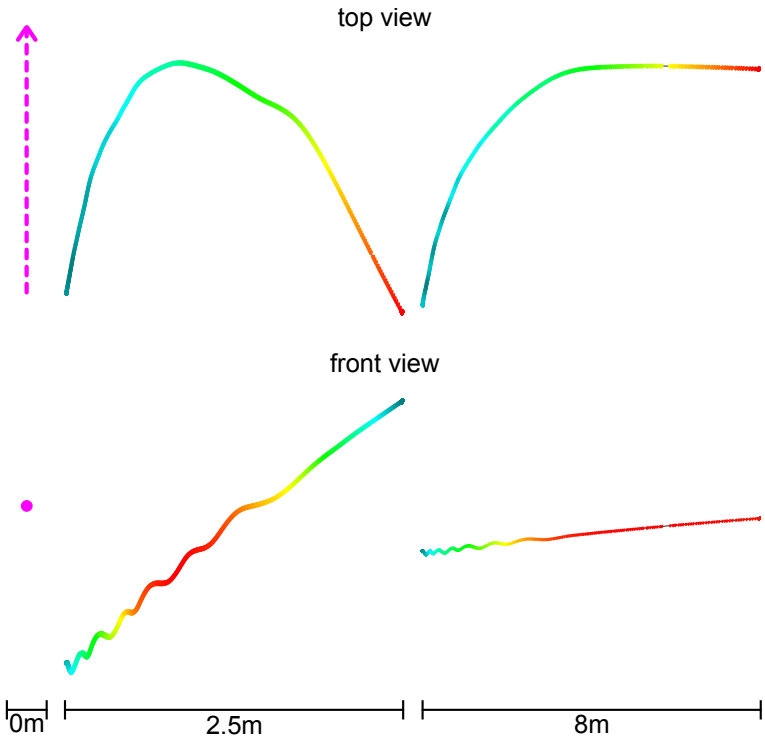


Figure 5.10: Seven metres long walk - expectation (left), Galaxy S III (center) and Huawei Honor (right)

Both the phones suffer from a strong gyroscope drift and the path is far from reality. We will perform the same trick as with the circles and insert a small pause after each two steps. The result is depicted in Figure 5.11.

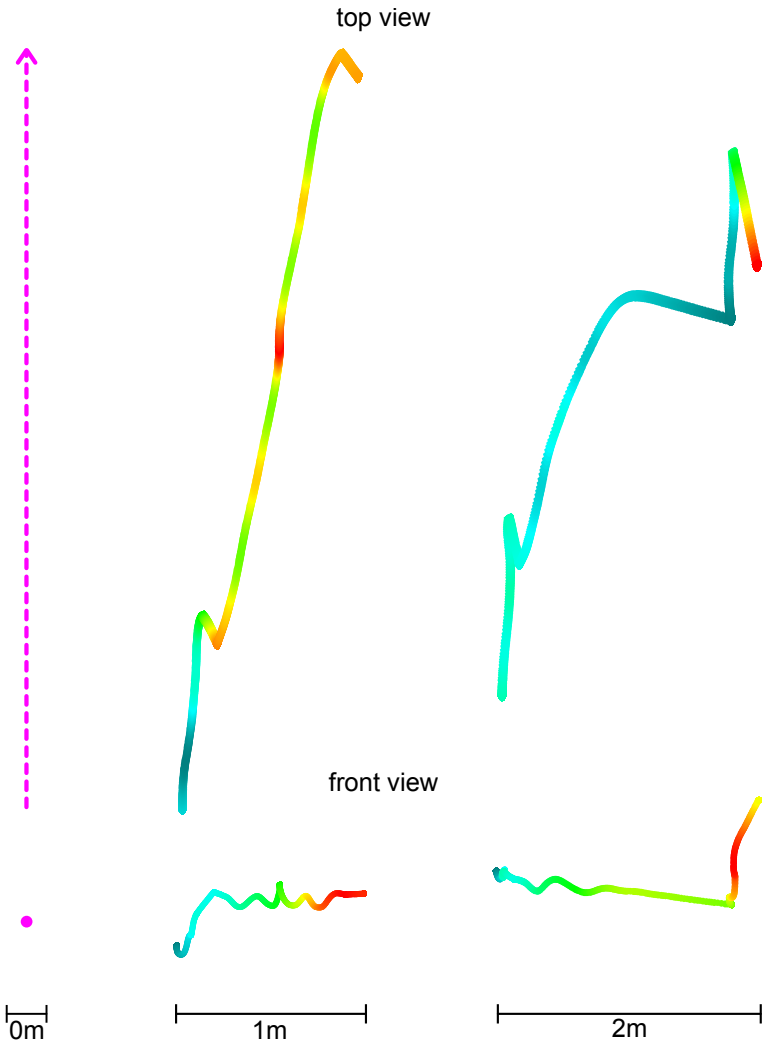


Figure 5.11: Seven metres long walk with pauses after each two steps - expectation (left), Galaxy S III (center) and Huawei Honor (right)

The result has significantly improved, thanks to the stabilization system. On the drawing from the Galaxy phone, even individual steps leave a trace.

We will repeat the very same experiment with disabled stabilization system to demonstrate its benefits. The outcome is depicted in Figure 5.12.

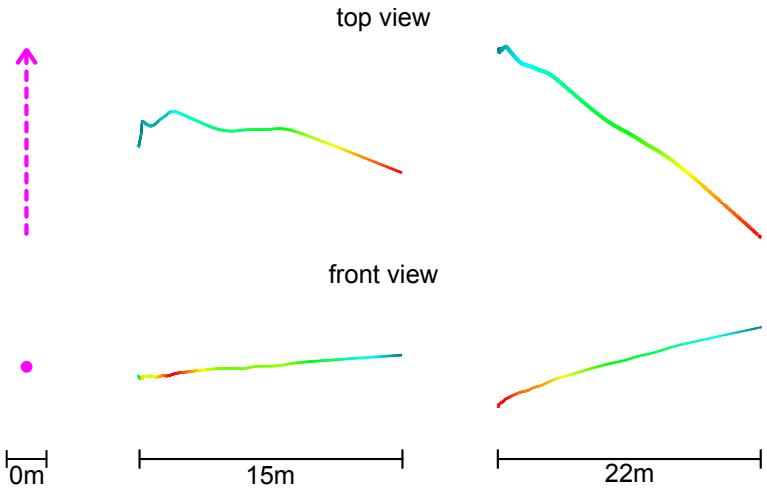


Figure 5.12: Seven metres long walk with pauses after each two steps and disabled stabilization system - expectation (left), Galaxy S III (center) and Huawei Honor (right)

The result is terribly inaccurate and no view reminds the real path.

5.5 Android Implementation

Android OS has its own implementation of sensor fusion hidden in the linear acceleration sensor. Simple double integration of its data should provide the position of the device. We have implemented the processing of this sensor, but the results are nowhere near reality. We suppose that Google did not mean to use it that way and did not implement any kind of stabilization. The Figure 5.13 demonstrates this sensor compared side by side to our implementation. The experiment was performed with the Galaxy S III.

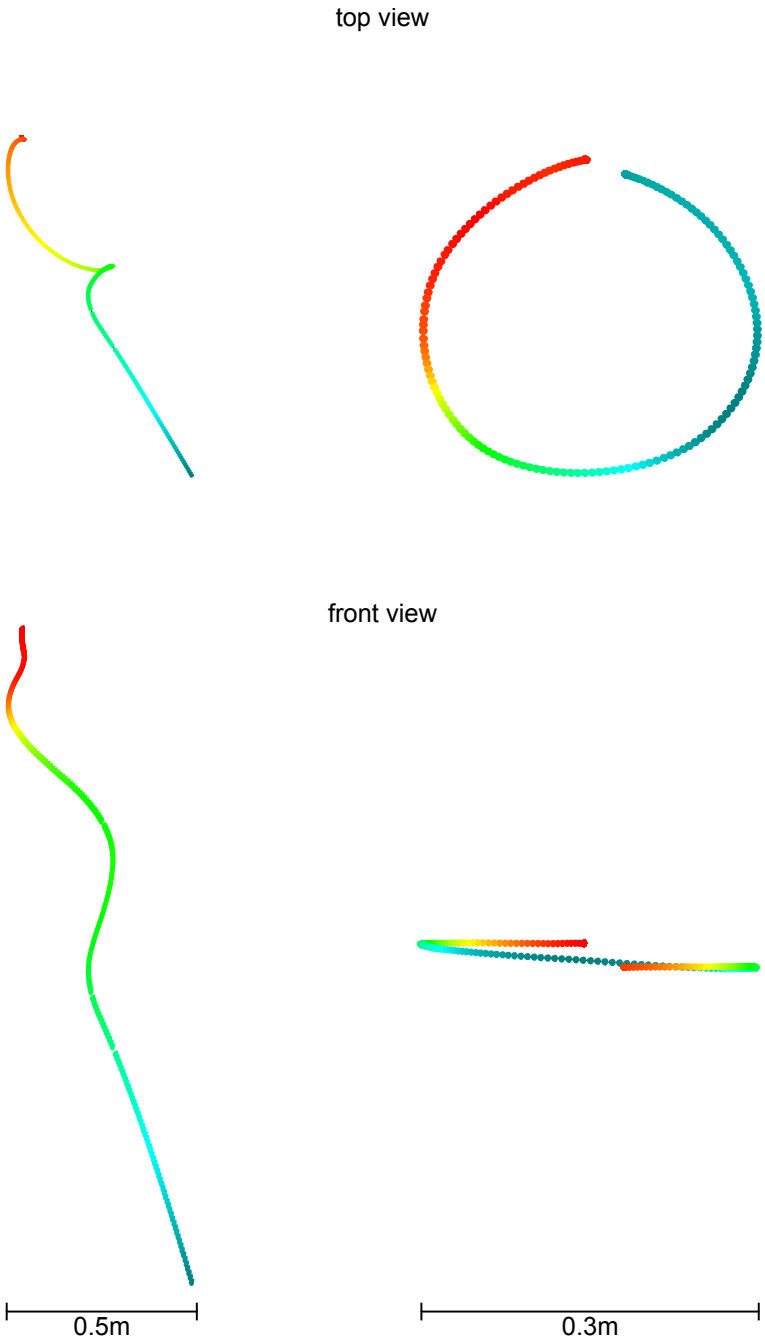


Figure 5.13: Circle from the Galaxy S III - double integration of linear acceleration (left) and our implementation (right)

The drift of the linear acceleration is too strong for the shape to be recongizible.

6. Conclusions

According to the results of our experiment, it is possible to track a short term motion of a device equipped with a low-end accelerometer, gyroscope and magnetometer. The motion path is accurate enough to e.g., recognize a gesture or give us a height from which the phone has fallen on the ground. Thanks to our stabilization system it is effectively possible to track any movement separable to short intervals. For example, a flying quadcopter that uses grid coordinates can fix its position using the inertial navigation, the only limitation is that during a long move, it has to stop every few seconds so that the stabilization system refreshes the orientation. The cheap sensors cannot be used as a jogging tracker or a GPS replacement, but are capable of tracking the motion of a device in a matter of seconds.

On the other side, when a motion contains significant change in orientation, the accelerometer has to be thoroughly tested if it is capable of keeping the same accuracy in all directions. In our case, both accelerometers drifted significantly, thus no movement containing strong rotation could be tracked.

When developing an application for Android operating system, we highly discourage from double integration of the linear acceleration sensor. The drift is too strong no matter if the device was laying still before the motion or not.

We have implemented a non-model method of tracking. However, for a specific device (like a quadcopter) we recommend to utilize the Kalman filter mentioned earlier, or similar model method, to improve the results.

A suggestion for a future work might be to design a method which does not require the stabilization of a device every few seconds. It may be achievable by, for instance, estimation of the direction of the drift and its real time compensation. It is possible that deeper knowledge of the sensor technology or more sophisticated experiments would uncover the pattern of the drift behaviour. Such improvement would bring the full inertial navigation with cheap sensors one big step closer to reality.

Bibliography

[1] Bartsch, Hans-Jochen. *Matematické vzorce*. 4th revised edition. Prague: Academia, 2006. ISBN 80-200-1448-9

[2] Sachs, David. *Google Tech Talks - Sensor Fusion [online]*. Available from: `http://www.youtube.com/watch?v=C7JQ7Rpwn2k`. August 2, 2010.

[3] Knuth, Donald E. *The Art of Computer Programming, volume 2: Seminumerical Algorithms*. 3rd edition. Boston: Addison-Wesley, 1998. ISBN 0201896842.

[4] Murray, Glen. *Rotation About an Arbitrary Axis in 3 Dimensions [online]*. Available from: `http://inside.mines.edu/fs_home/gmurray/ArbitraryAxisRotation/`. June 6, 2013.

[5] Prokop, Mirek. *Víc než pět smyslů, Computer (magazine)*. March, 2014. ISSN 1210-8790

[6] Qt project. *Qt project official website [online]*. Available from: `http://qt-project.org/`. May 11, 2014.

[7] Russel, Stuart, Norvig, Peter. *Artifical Intelligence: A Modern Approach*. 3rd edition. Prentice Hall, 2010. ISBN 0-13-604259-7.

[8] Android Operating System. *Android project official website [online]*. Available from: `http://www.android.com/`. May 11, 2014

[9] Wikipedia. *Kalman filter on Wikipedia [online]*. Available From: `http://en.wikipedia.org/wiki/Kalman_filter`. May 11, 2014.

[10] Oculus VR. *Oculus Rift official website [online]*. Available from: `http://www.oculusvr.com/`. May 11, 2014.

[11] Google. *Project Tango [online]*. Available from: `https://www.google.com/atap/projecttango/`. May 11, 2014.

[12] Oculus VR Team. *Oculus VR blog - Building a Sensor for Low Latency VR [online]*. Available from: `http://www.oculusvr.com/blog/building-a-sensor-for-low-latency-vr/`. January 4, 2013.

[13] Oculus VR Team. *Oculus VF blog - Oculus Joins Facebook [online]*. Available from: `http://www.oculusvr.com/blog/oculus-joins-facebook/`. March 25, 2014.

[14] Victor Luckerson for time.com. *Time.com - Facebook Buying Oculus [online]*. Available from: `http://time.com/37842/facebook-oculus-rift/`. March 25, 2014.

[15] DARPA Media. *Darpa official website - One chip to navigate without GPS [online]*. Available from: `http://www.darpa.mil/NewsEvents/Releases/2013/04/10.aspx`. April 10, 2013.

[16] Razer Inc. *Razer Hydra official website [online].* Available from: http://www.razerzone.com/gaming-controllers/razer-hydra-portal-2-bundle/. July 29, 2014.

[17] Microsoft *MSDN Kinect Specification [online].* Available from: http://msdn.microsoft.com/en-us/library/jj131033.aspx. July 29, 2014.

[18] Nintendo *Nintendo Wii official website [online].* Available from: http://wii.com/. July 29, 2014.

[19] InvenSense Inc. *InvenSense official website [online].* Available from: http://invensense.com/mems/gyro/documents/articles/071508.html. July 29, 2014.

[20] Caron, Frank. *Of gyroscopes and gaming for Arstechnica.com [online].* Available from: http://arstechnica.com/gaming/2008/08/wii-motion-sensor/. July 29, 2014.

[21] Seifert, Kurt Camacho, Oscar. *Freescale Application Note AN3397 - Implementing Positioning Algorithms Using Accelerometers [available online].* Available from: http://www.freescale.com/files/sensors/doc/app_note/AN3397.pdf. February, 2007.

[22] Android Play Store. *GPS Status & Toolbox [online].* Available from: https://play.google.com/store/apps/details?id=com.eclipsim.gpsstatus2. June 10, 2014.

Appendix A - TrackMe User Documentation

6.1 Overview

On the CD attached to this bachelor thesis, there are two executable programs. The first one, TrackMe Sensors, is intended to run on a smartphone and stream its sensor data to a computer network. The second one, TrackMe GUI, is intended to run on a computer and receive and process the data from the smartphone. The GUI application is based on a library which implements the extended flowchart from Figure 4.12. The overall picture of the software solution is shown in Figure 6.1.

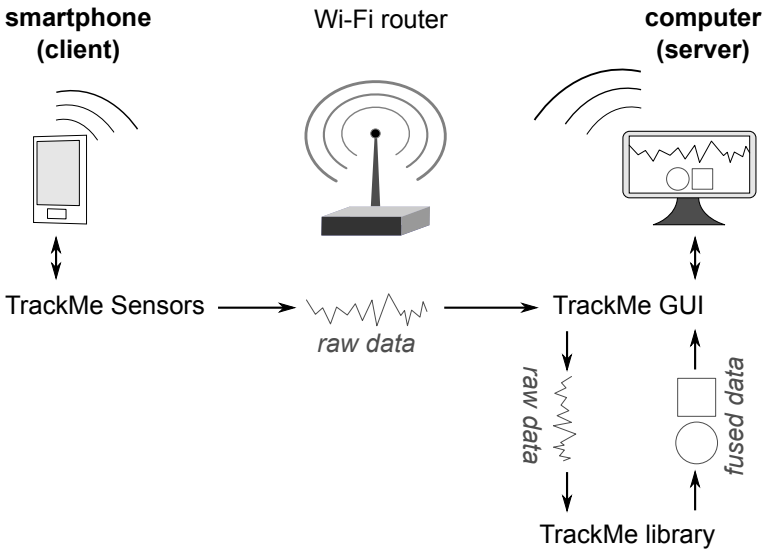


Figure 6.1: Software overview

6.2 TrackMe GUI

TrackMe GUI is a computer software, which provides a user-friendly interface for processing and visualization of data produced by accelerometer, gyroscope and magnetometer sensors. The entire application supports all commonly used operating systems, i.e., Linux, Mac and Windows.

The application provides a tabbed interface, where each tab visualizes different data. The list of individual tabs and their description will be discussed in Subsections 6.2.2-6.2.7. All the tabs have synchronized clock and show the same time interval.

6.2.1 Startup

When the application is executed, it searches for a configuration file `trackme_config.txt` in the current working directory. All the options which are not explicitly specified in the configuration file receive their default value. The

same stands if the file does not exist. All the options and their default values are listed in Section 6.2.9.

After the configuration, a TCP server is started on port 50000 (by default) on all available IP addresses. The list of addresses is visible in the status bar of the application (screenshot in Figure 6.2). When a client connects to the server, the server prepares for data processing and blocks all other connection requests until the client disconnects or the connection is interrupted.

After a successful connection, the underlying library will wait for the stillness of the device. On the first successful stillness detection, the library stores initial accelerometer and magnetic field vectors. This process is called calibration and its state is shown in the Statistics Tab (individual tabs will be described later). Until the software calibrates, the only valid plot is the one with the raw sensor data, all the other plots remain empty. After the calibration, the empty plots will automatically activate.

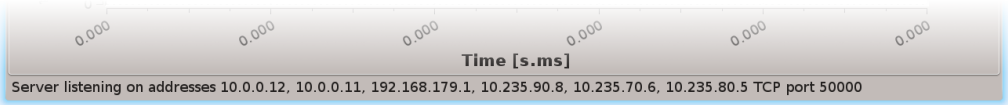


Figure 6.2: IP address list at the bottom of the GUI window

The following sections list and describe all available tabs.

6.2.2 Raw Sensors Tab

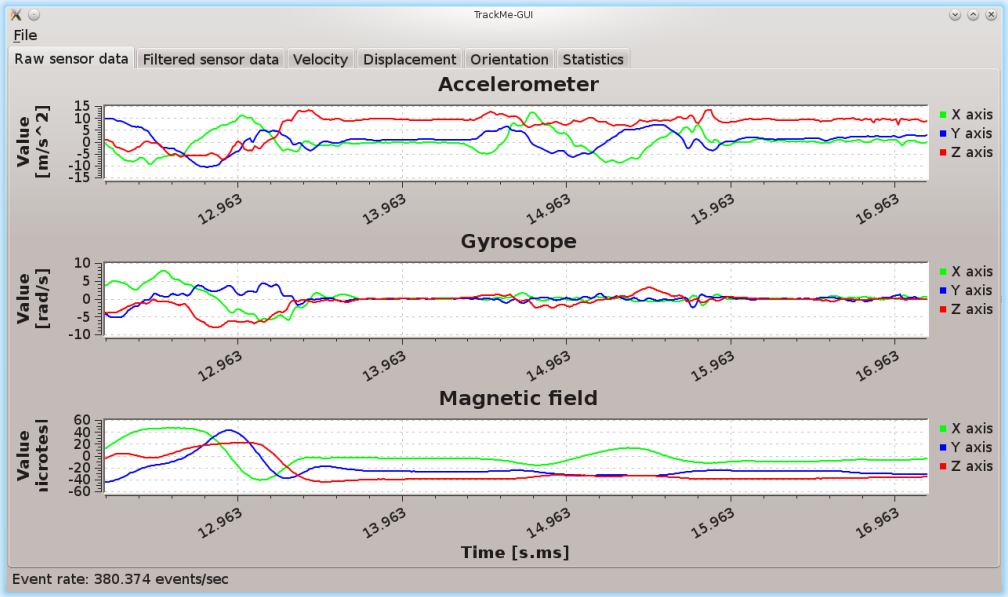


Figure 6.3: Raw sensors tab

The Raw Sensors tab (screenshot in Figure 6.3) visualizes the raw sensor data. No filtering is applied and no data are dropped.

6.2.3 Filtered Sensors Tab

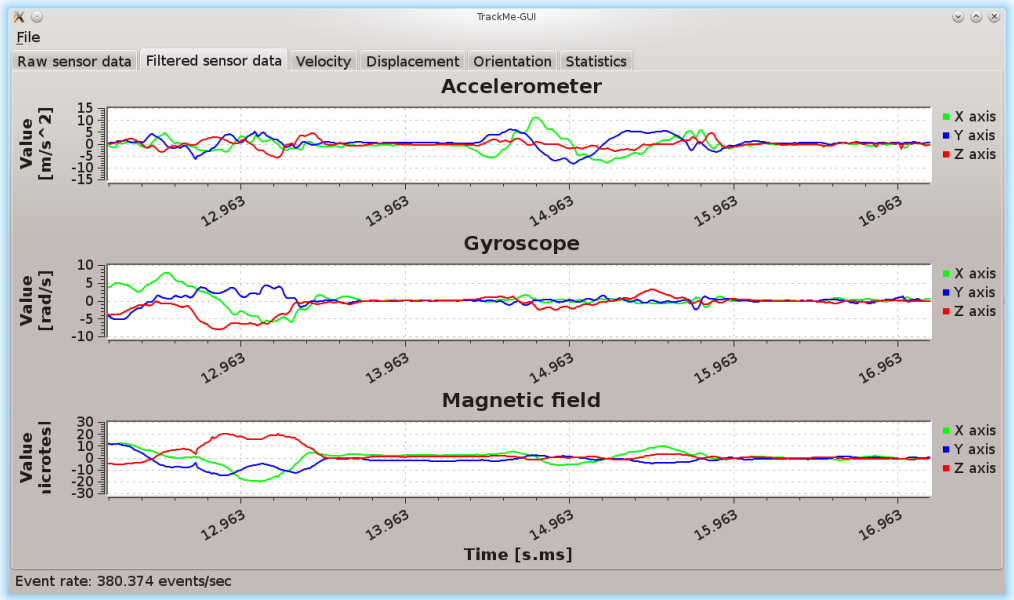


Figure 6.4: Filtered sensors tab

The Filtered Sensors tab (screenshot in Figure 6.4) visualizes data after application of various filters from the extended flowchart from Figure 4.12. The list of applied filters is following:
For accelerometer:

- drop first n events
- drop until calibrated
- set to zero if low
- multiply by orientation (transform to global coordinates)
- remove gravity

For gyroscope:

- drop first n events
- drop until calibrated
- set to zero if low

For magnetic field:

- drop first n events
- set to zero if low
- multiply by orientation (transform to global coordinates)

- remove initial force

The individual filters have been described in Chapter 4.

The magnetic field and the acceleration data are already pre-multiplied by the orientation matrix, thus they are relative to the global coordinate system. During the stillness of the device, filtered values of all the sensors should be close to zero.

6.2.4 Velocity Tab

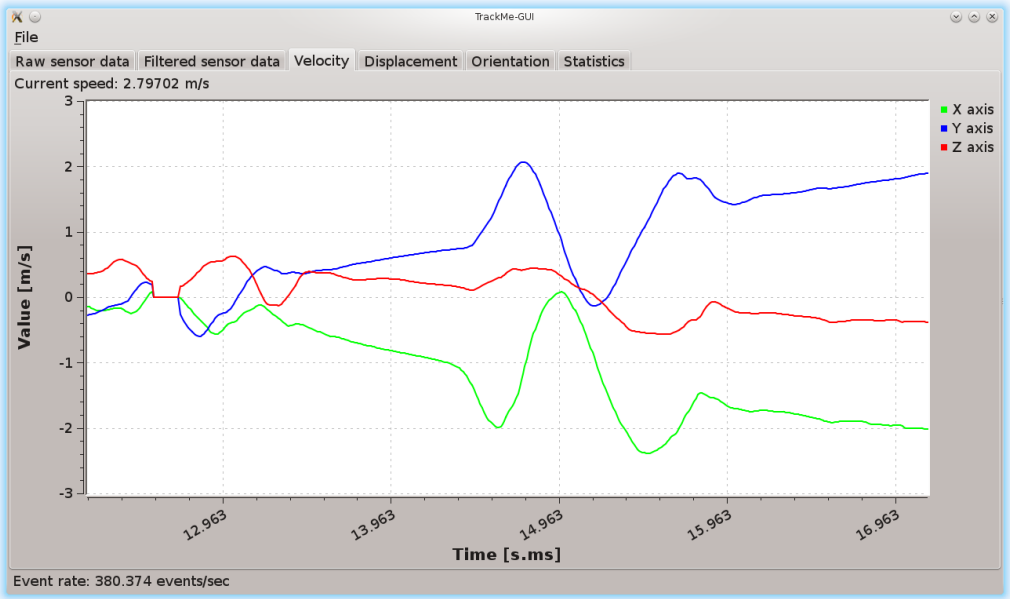


Figure 6.5: Velocity tab

The Velocity tab (screenshot in Figure 6.5) plots velocity and prints speed (magnitude of velocity) of the device.

6.2.5 Displacement Tab

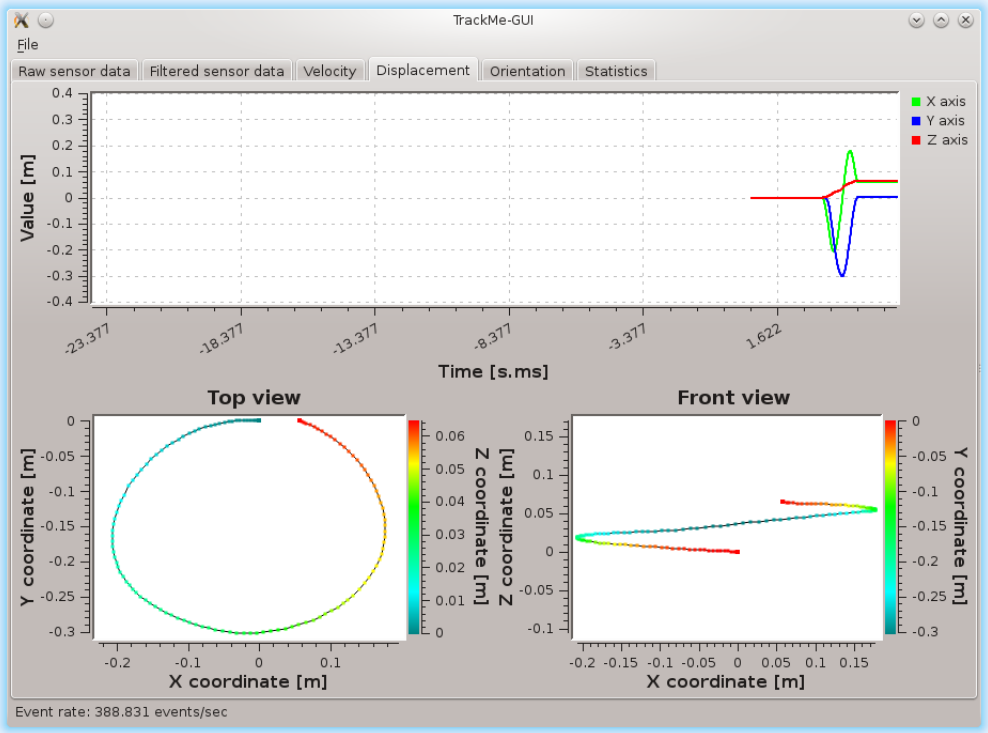


Figure 6.6: Displacement tab

The Displacement tab (screenshot in Figure 6.6) visualizes the position of the device relative to its initial placement. The plot at the top of the window simply plots the position as a vector and the remaining two plots are projections of the position to the xy and xz planes respectively. The two projections correspond to the top and front views demonstrated in Figure 5.1.

6.2.6 Orientation Tab

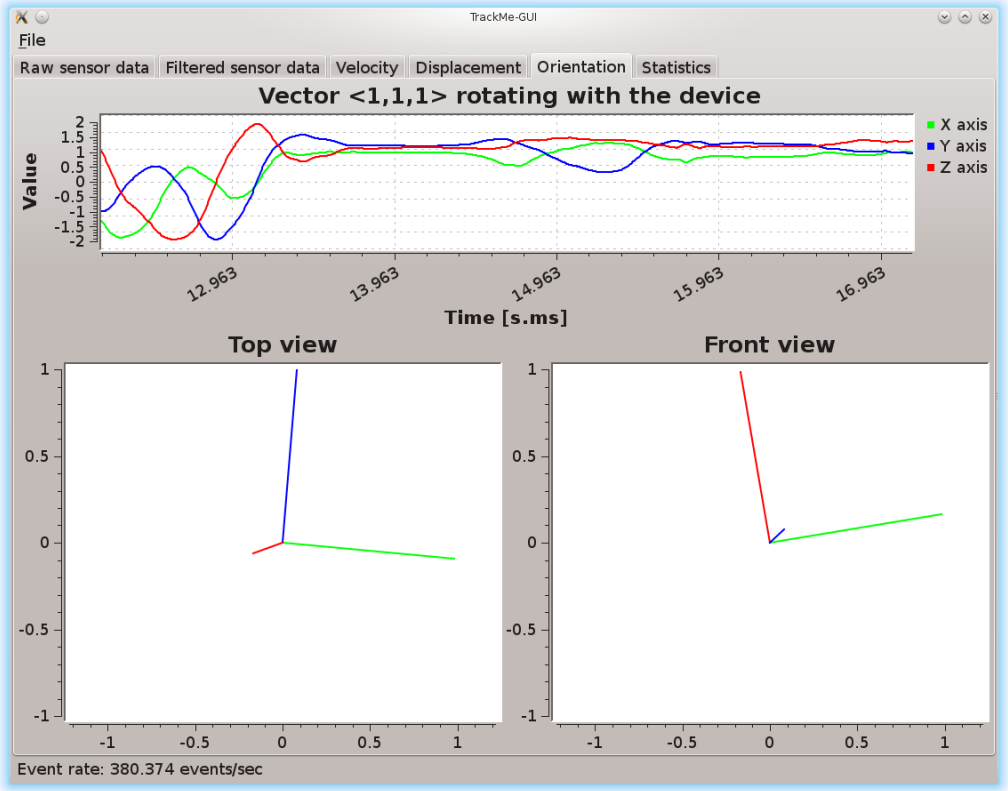


Figure 6.7: Orientation tab

This Orientation (screenshot in Figure 6.7) visualizes the orientation of the device. The plot at the top of the window shows the vector $\langle 1, 1, 1\rangle$ pre-multiplied by the device orientation matrix. That corresponds to the vector $\langle 1, 1, 1\rangle$ rotating with the device converted to the global coordinate system. The remaining two plots are created by pre-multiplication of the canonical base (i.e., vectors $\langle 1, 0, 0\rangle$, $\langle 0, 1, 0\rangle$ and $\langle 0, 0, 1\rangle$) by the orientation matrix and projecting the result to the xy and xy planes respectively. The two projections correspond to the top and front views demonstrated in Figure 5.1.

6.2.7 Statistics Tab

Figure 6.8: Statistics tab

The Statistics tab (screenshot in Figure 6.8) shows various statistics about the received data. The first line tells us whether the software is calibrated, that is, whether gravity and initial magnetic field vectors have already been stored.

The other values shown for each of the sensors are:

- **Number of samples**: Number of sensor events from which the statistics is calculated.
- **Event rate**: Number of incoming sensor events per second.
- **Mean**: Mean of the event values.
- **Mean magnitude**: Magnitude of the mean (e.g., acceleration mean magnitude in still state should be eqal to 9.81, which is the Earth's gravity).
- **Magnitude variance**: Variance of the event magnitudes.
- **Sum of squares**: Sum of squared magnitudes.

6.2.8 File Menu

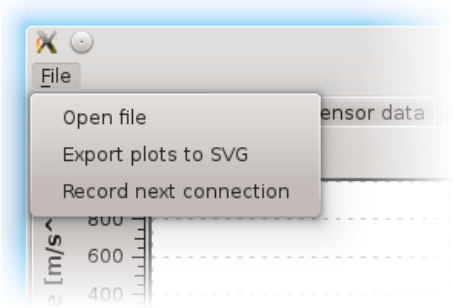


Figure 6.9: File menu

The File Menu (screenshot in Figure 6.9) is the only clickable part of the entire application. It contains the following options:

- Open File: Instead of reading data from a network stream, the application can open a file in the same binary form as the streamed data should be. The TrackMe Sensors and the TrackMe GUI applications are both able to create such a file. It is even possible to record the network stream on a lower level by, for instance, UNIX Netcat. When the binary file is opened, the software simulates real time reception by putting appropriate time pauses between the read events.

- Export plots to SVG: For presentational purposes it might be useful to save a plot as a picture. This menu action lets the user to choose a folder to which all the plots are exported in the SVG[1] file format.

- Record next connection: This action allows the user to record the communication with a client to a file. When selected, the software asks the user for the output file path and waits until the next connection. All the raw sensor events received during this connection will be stored into the file in the same format as the streamed data.

6.2.9 Configuration Options

During the startup the application searches for a configuration file in the current working directory. The file name has to be

`trackme_config.txt`

The default self-documented configuration file should be distributed with the executable. When no such file is found, the software uses default values for all the available options.

Each line of the file should have one of the following forms:

- `# lines beginning with '#' are ignored`

[1] Scalable Vector Graphics

- `# empty lines`

  `# are ignored too`

- `VARIABLE=VALUE`

The list of available options and their default values is following:

- `STILLNESS_ACC_EVENT_NUMBER = 16`
  `STILLNESS_GYR_EVENT_NUMBER = 32`
  `STILLNESS_MAG_EVENT_NUMBER = 32`

  Number of samples from which the statistics are calculated.

- `STILLNESS_ACC_VARIANCE_THRESHOLD = 0.01`
  `STILLNESS_GYR_SQUARES_THRESHOLD = 0.1`
  `STILLNESS_MAG_VARIANCE_THRESHOLD = 0.15`

  Threshold under which the data are considered to be still.

- `STILLNESS_GRAVITY_EPSILON = 1.5`
  `STILLNESS_MAG_FIELD_EPSILON = 30.0`

  Another condition to stillness detector is that the current value and initial value do not differ by more than the epsilon. To turn off the entire stabilization system, set these epsilons to negative values.

- `STILLNESS_SPEED_QUOCIENT = 0.0`

  When the device is considered to be still, the velocity is multiplied by this value.

- `boolean STILLNESS_UPDATE_GRAVITY = 1`

  When the device is considered to be still, should the gravity magnitude be updated?

- `ACC_DROP_INIT = 32`
  `GYR_DROP_INIT = 32`
  `MAG_DROP_INIT = 32`

  Number of events that are dropped at the very beginning.

- `ACC_EPSILON_ZERO = 0.07`
  `VEL_EPSILON_ZERO = 0.3`
  `GYR_EPSILON_ZERO = 0.02`

  If the magnitude of an event is less than the epsilon, the software sets all the values to zero. It can be useful for noise removal. Set to 0.0 to disable this feature.

- `TCP_PORT = 50000`

  TCP port number the server binds to.

- `PLOT_REFRESH_INTERVAL = 100`

  Time interval between replots in milliseconds.

- `PLOT_INTERVAL = 5000`

  Time interval of visualized data in milliseconds.

- `SVG_EXPORT_WIDTH = 300`
  `SVG_EXPORT_HEIGHT = 200`

  SVG canvas size of exported plots in millimeters.

6.3 TrackMe Sensors

TrackMe Sensors is an application for the Android OS intended to be used on smartphones and similar mobile devices. The application has the ability to read and stream sensor data in real time through the TCP protocol which allows the use of an existing Wi-Fi or mobile network. The application provides a straightforward one-window interface, whose screenshot is depicted in Figure 6.10. The application is written mainly in the Java programming language.

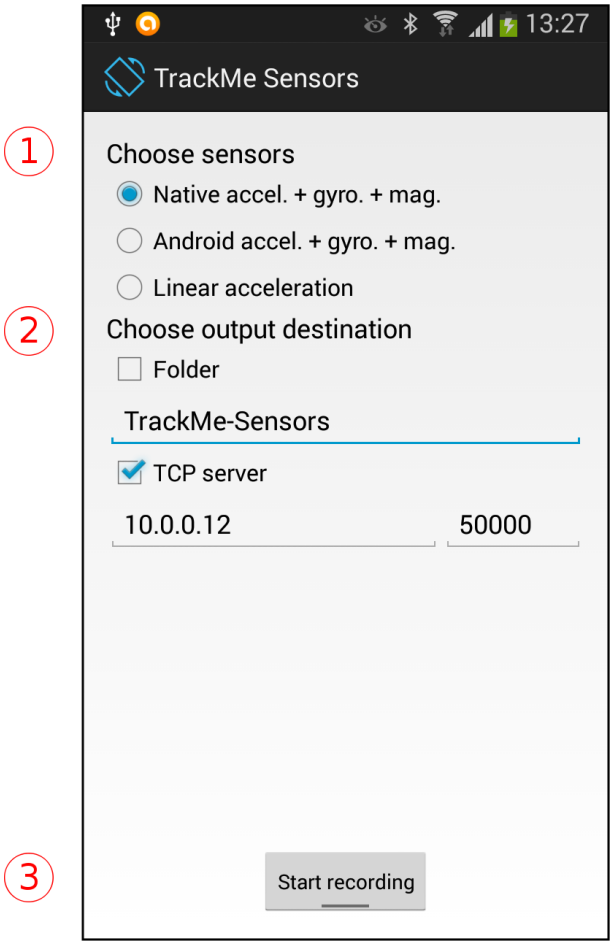


Figure 6.10: TrackMe Sensors application

For the situation with no Wi-Fi network in the area or weak signal, the most Android phones themselves have the ability to create a hotspot network. It is called a tethering or a portable hotspot and the capability of this service should be consulted with the device handbook.

In the following subsections, the individual parts of the window are described.

6.3.1 Choose sensors

The first step is to choose the source of the sensor data. Available options are:

- **Native accel. + gyro. + mag.**: This option uses Android Native Development Kit to bypass the Java and read the sensors using the C programming language. It is the recommended option.
- **Android accel. + gyro. + mag.**: This option uses the Android API and relies entirely on the Java event mechanism.
- **Linear acceleration**: This option reads the Android's linear acceleration sensor. It is a software based sensor whose data are synthesized by fusing the data from other sensors.

6.3.2 Choose output

The second step is to choose the output destination. The application is capable of two options - streaming the data to a TCP server and writing the data to a file. The file contains the same binary data as the network stream, thus it can be streamed later or directly opened by the TrackMe GUI application.

6.3.3 Start recording

The last step is to click the button at the bottom of the window. It starts the reading and the streaming. There is a 300ms delay between clicking the button and the beginning of the reading to eliminate the influence of the click on the sensor data.

6.4 Communication Protocol

The communication between the server and the client is a one-way TCP stream of sensor data from the client to the server. It consists of consecutive 25 bytes long chunks of data, where each chunk represents a sensor event. The format of each chunk is depicted in Figure 6.11.

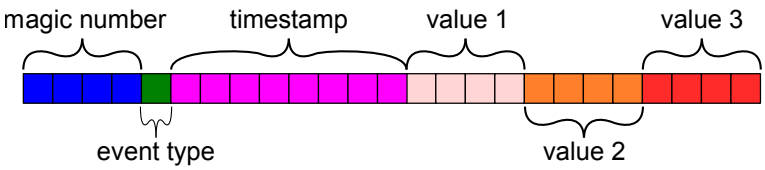


Figure 6.11: Chunk of a sensor event. Each square represents one byte.

All the values have the Big Endian byte order. The individual parts are following:

- **Magic number** is a signed 32-bit integer constant, always set to 0x56289740.
- **Event type** is a signed 8-bit integer, whose value determines the purpose of the chunk. Available values are:

  - UNKNOWN = 0x0
  - ACCELEROMETER = 0x01
  - GYROSCOPE = 0x02
  - MAGNETIC_FIELD = 0x03
  - ORIENTATION = 0x04
  - VELOCITY = 0x05
  - DISPLACEMENT = 0x06
  - LINEAR_ACCELERATION = 0x07
  - START = 0x10
  - STOP = 0x20

- **Timestamp** is a signed 64-bit integer representing the time of the sensor reading in nanoseconds. The absolute value is not important, the difference between subsequent events is what matters.
- **Value 1, value 2 and value 3** are single-precision IEC-559 floating point numbers representing the values of the sensor reading.

The very first chunk has to be the `START` type and the very last chunk has to be the `STOP` type. All the other values in these chunks are meaningless.

6.5 Usage Example

The order of actions to use the entire software set is very straightforward. It consists of the following steps:

1. Launch of the TrackMe GUI application on a computer.
2. Launch of the TrackMe Sensors application on a smartphone.
3. Setting the server IP address and port number in the TrackMe Sensors application. These values are visible in the status bar of the TrackMe GUI, as shown in Figure 6.2.
4. Click on the "Start recording" button.

If the stabilization system never detects stillness or, on the contrary, the displacement tab does not react to slow motions, then the stillness variance thresholds in the configuration file are not properly set. In the case of never detecting stillness, they should be heightened and in the case of low sensitivity, they should be lowered. The specific values can be deduced from the Statistics Tab of the TrackMe GUI application. For instance, the device is laying on a table and the system is not calibrating. Therefore, there exists one or more sensors whose stillness thresholds are lower than the current variance of the sensor data. By opening the configuration file and raising the thresholds above the values of the current variance, which is shown in the Statistics tab, we assure that in the next launch, the software will detect stillness whenever the device lays on a table.

Appendix B - Flowchart Implementation

In this appendix, we will describe our implementation of the extended flowchart from Figure 4.12. The code will be written in the C++11 programming language and understanding of this chapter requires its deep knowledge.

The flowchart resembles a pipeline into which the individual sensor events are pushed and whenever it has enough data available, it produces the position and orientation of the device. Each routine (flowchart rectangle) represents a filter which modifies, accumulates or drops the data which pass through and output of each routine is the input of the subsequent call of the routine. We will propose a programming pattern which allows the creation of a source code in a form closely reminding the original flowchart.

In Section 6.6, we will describe how the individual routines will be represented and in Section 6.7 we will connect the routines together.

6.6 Routine Representation

We need a representation satisfying all the potential requirements of each routine:

- **receive event**: Capability to receive an input.
- **pass event**: Capability to produce an output.
- **drop event**: Capability not to produce any output. For example, the "drop if not calibrated" routine consumes sensor events if the software is not calibrated and passes them through otherwise.
- **storage**: Capability to store a persistent data. For example, the delta integration has to remember the previous event.

The only reasonable representation of such object in the C++ programming language is the `class` or the `struct`. We will provide this class with a `receive()` function used as the entry point for the input and a `send()` function used to emit the output. When the object receives an input through the `receive()` function, it might or might not call the `send()` function to pass an output to the subsequent object. The Program 1 demonstrates an example of a filter that will be receiving numbers and with each tenth value it will produce the average of the last ten values.

Program 1 AverageFilter source code

```
struct AverageFilter {
  int sum = 0;
  int cnt = 0;

  void receive(int value) {
    sum += value;
    cnt += 1;
    if (cnt % 10 == 0) {
      send(sum / 10);
      sum = 0;
      cnt = 0;
    }
  }

  ...
};
```

But how should the `send()` function look like? Its destination can vary from instance to instance, thus it has to be remembered by the object itself. We can use the `std::function<>` template to store the destination of the `send()` function, as demonstrated in Program 2.

Program 2 Mutable send function source code

```
struct AverageFilter {
  ...


  std::function<void(int)> send;
};
```

Using this approach, the implementation of all the filters from the flowchart is fairly straightforward and we will not provide further description. All the filters are well documented in the API documentation included on the attached CD.

In the next section, we will connect the filters together.

6.7 Routine Connection

For the sake of simpler description, we will create a second filter. This time, it will consume a number and print its value on the standard output. Its source code is demonstrated in Program 3.

Program 3 PrintFilter source code

```
struct PrintFilter {
  void receive(int value) {
    std::cout << value << std::endl;
  }

  std::function<void()> send;
};
```

If there are just two or three filters, we might perform the connection manually, as shown in Program 4.

Program 4 Manual connection

```
AverageFilter A,B;
PrintFilter C;

A.send = [&](int value) { B.receive(value); };
B.send = [&](int value) { C.receive(value); };
```

However, when the pipeline prolongs, it becomes quite cumbersome and unmanagable. Therefore, we will implement a function for an automatic pipeline connection.

The filters are of different types and thus cannot be stored in a container. Therefore, we cannot easily iterate through and connect them together. Fortunately, the new C++11 standard provides a powerful tool for compile time programming called variadic templates. A variadic template allows a function to receive an arbitrary number of arguments of arbitrary types and process them recursively. The process might be very intuitive for those familiar with functional and logical programming languages.

Let us implement a function, which receives a list of filters and links their `send()` functions to the corresponding `receive()` functions. It always connects the first two filters together and recursively calls itself on the remaining filters. When there is just one argument left, the recursion stops. The implementation of this function is demonstrated in Program 5.

Program 5 Automatic pipeline connection function

```
// Recursion bottom
template <typename Last>
void connectPipeline(const Last &)
{
}

// Recursive connector
template <typename First,
          typename Second,
          typename... Rest>
void connectPipeline(First &first,
                     Second &second,
                     Rest&... rest)
{
  // Get the output type of the first filter
  using Type = typename decltype(first.send)::argument_type;

  // Connection
  first.send = [&](Type arg) { second.receive(arg); };

  // Recursion
  connectPipeline(second, rest...);
}
```

The application of the `connectPipeline()` function on the filters from Program 4 is demonstrated in Program 6.

Program 6 Automatic pipeline connection

```
AverageFilter A,B;
PrintFilter C;

connectPipeline(A,B,C);
```

Using this technique, the flowchart can be literally replicated to a source code. Program 7 demonstrates a slightly modified piece of code from the TrackMe library processing the accelerometer data.

Program 7 Accelerometer processing from the TrackMe library

```
auto accelerometer_filters = std::make_tuple(
  Drop<T>{ACC_DROP_INIT},
  Statistics<T>{&stats_, STILLNESS_ACC_EVENT_NUMBER},
  calibrate,
  fix_drift,
  PreMultiplyBy<T, OT>{&orientation_},
  RemoveValueOf<T>{&gravity_},
  EpsilonZero<T>{ACC_EPSILON_ZERO},
  DeltaIntegrate<T>{},
  AddValueTo<T>{&velocity_},
  DeltaIntegrate<T>{},
  AddValueTo<T>{&displacement_},
);
connect_pipeline(accelerometer_filters);
```

The CD attached to this thesis contains a `RecursivePipeline.hpp` file with the implementation of an extended version of the described process.

Attachments

Contents of the CD attached to this bachelor thesis:

- `TrackingOf3DMovement.pdf` - Electronic form of this bachelor thesis.
- `src/TrackMe/` - Source code and API documentation of the library.
- `src/TrackMe-GUI/` - Source code of the GUI front-end.
- `src/TrackMe-Sensors/` - Source code of the Android application.
- `TrackMe-GUI-linux64/` - GUI executable for the Linux operating system.
- `TrackMe-GUI-win32/` - GUI executable for the Windows operating system.
- `TrackMe-Sensors.apk` - Executable of the Android application.
- `experiments/` - Recorded sensor streams of various experiments. The files can be opened in the TrackMe-GUI.