Throw Your Smartphone

Semester Thesis

Anton Beitler

abeitler@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Barbara Keller, Jara Uitto
Prof. Dr. Roger Wattenhofer

October 20, 2012

Abstract

The Android app developed in this work enables the user to take aerial photographs by throwing the smartphone in the air. This involved overcoming inherent limitations of the underlying system, such as camera lag and sensor saturation. A motion detection algorithm is proposed which reliably detects the stages of a vertical throw and copes with the system's limitation. Experiments show that the proposed method is capable of increasing peak prediction accuracy by more than 80%.

Contents

Chapter 1

Introduction

There are more than one billion of them. Smartphones - mostly running the Android operating system, mostly used by male users between 25-34 and mostly for communication and gaming[1]. First to introduce an accelerometer into the phone hardware was Nokia followed by many others. Where first they were used in cars for collision detection in order to release the air bags, now accelerometers and other motion sensing infrastructure are an integral part of the phone's interface.

The project *Throw Your Smartphone* brings this to a new level which sparks people's creativity on what can be done with a smart phone. *ThrowMeApp* is the name of the application which is designed and built in this project. Its purpose is to attract smartphone users for an interaction with the physics of nature and to engage them in creative photography. The application allows users to take aerial photographs with their smartphones by throwing it in the air. This happens in an orientation sensitive way, meaning that the user can specify a direction to which the picture is taken. Technically, ThrowMeApp comprises a number of challenges as it uses the hardware outside the normal scope of operation which brings both the hardware and the user to their limits.

Similar systems exist for example in a shape of a sphere equipped with multiple cameras which construct a spherical panorama picture [1]. Another suggested approach was to throw an actual video camera to get a continuous stream of pictures [2]. Applications where a smart phone is thrown in the air also exist in the form of games[2] which give high scores for high or long throws.

This report documents the design and development process of ThrowMeApp. It starts with an introduction to the sensing hardware used in the project and the necessary theory to use it. Then, the implementation of the application is described followed by a number of experiments that were conducted with it. The results and their consequences are discussed last.

[1]http://www.go-gulf.com/blog/smartphone, (accessed 04.10.2012)

[2]http://itunes.apple.com/us/app/hangtime!/id354893714?mt=8, https://play.google.com/store/apps/details?id=com.bytemods.throw_phone&hl=en

Chapter 2

Materials and Methods

2.1 Set-up

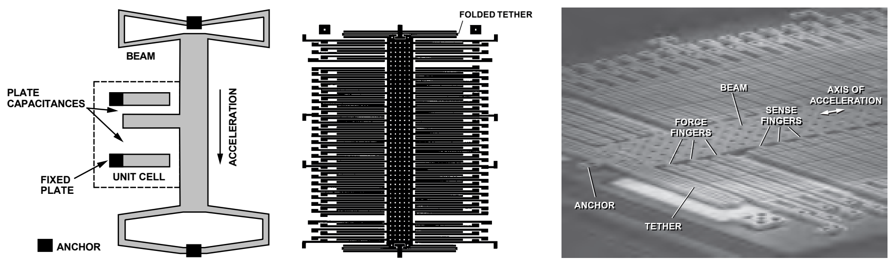
ThrowMeApp is an interactive application. Users engage with its functionalities physically and not just virtually. Smartphones provide a variety of interfaces to make this possible. How they work and what needs to be considered when using them is discussed in this chapter.

2.1.1 Motion Sensing with Smart Phones

Smartphones mostly implement motion and orientation aware applications in order to provide an additional interface to the system. This is done with the use of accelerometer and gyroscope sensors which are embedded into the phones hardware as Micro-electromechanical systems (MEMS). MEMS are micro scale systems consisting of mechanical and electrical components fabricated in processes that are common in the semiconductor manufacturing industry. As these processes became more advanced and less expensive, the MEMS technology experienced a break through in sensing applications for electronic consumer products such as smart phones.

Accelerometer

The basic structure of an accelerometer is a spring-mass system, where the elongation of the spring , i.e. the displacement of the mass, is proportional to the applied acceleration. However, a multitude of implementations differ in the method of sensing the displacement. A common approach in MEMS accelerometers employs the concept of differential capacitance [3]. An inertial suspended mass is attached to fixed anchors by thin stripes of silicon acting as a spring. Additionally there are electrodes (force fingers) attached to the mass which are positioned between a set of fixed plates (sense fingers) forming capacities between a force



(a) Conceptual structure (b) ADXL 150 structure (c) ADXL 150 under a microscope

Figure 2.1: MEMS structure of a capacitive accelerometer [4]

finger and each of the sense fingers (Figure 2.1). The difference between these capacities is proportional to the displacement of the mass and therefore a measure of the applied acceleration.

In order to obtain an acceleration vector in three dimensions, three sensors are oriented perpendicular to each other. This defines a coordinate system where each sensor measures the projection of the total acceleration on the corresponding coordinate axis.

The acceleration that is measured by such a sensor is expressed in inertial coordinates and is commonly called *proper acceleration* [5]. This is the acceleration that an object feels acting upon itself rather than the acceleration which an observer in an external reference frame might see (coordinate acceleration). For example the phone laying still on the ground has proper acceleration of $-9.81\ m/s^2$ because the spring inside the sensor is elongated by the gravitational force acting on the mass. However, its coordinate acceleration is $0\ m/s^2$ because the phone does not change its velocity with respect to the earth when laying still on the ground.

This concept can be used to detect the orientation of the phone's screen relative to the ground, which allows for automatically switching between the landscape and portrait screen modes.

2.2 Physical Model of a Vertical Throw

In this section a mathematical model for the translative motion of the vertical throw in the gravitational field of the Earth is derived. This allows to analytically describe the motion which the phone performs during the throw. Additionally, analytic expressions for characteristic quantities of the throw are derived for later use in the application.

In this model the phone is approximated by a mass point m with a position $z(t)$

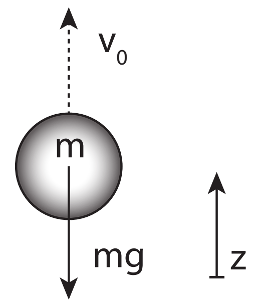


Figure 2.2: Mass point model during free-fall

at time t. As only vertical throws are modeled, $z(t)$ can also be called *height* or *altitude*. The curve which the coordinate $z(t)$ describes over time is called *trajectory*.

The throw motion performed by a human hand can be split in two phases: Firstly, the *acceleration phase* which starts at time $t = 0$ and ends at $t - t_0$ and secondly, the *free-fall phase* which starts at time $t = t_0$. The motion motion equations for each of the two phases are derived in the following paragraphs.

Acceleration Phase ($0 < t < t_0$)

At the beginning of the acceleration phase the phone is assumed to lie motionless in the hand. Hence, its initial velocity $\dot{z}(0)$ is zero. For convenience the initial position of the phone $z(0)$ is assumed to be in the origin of the coordinate system. During the acceleration phase the phone experiences an acceleration $a(t)$ caused by the motion of the hand. The Newton equation with the conditions given above state the following initial value problem for the displacement of the phone.

$$m \cdot \ddot{z}(t) = m \cdot a(t) \qquad 0 \leq t < t_0 \tag{2.1}$$

$$\dot{z}(0) = z(0) = 0 \tag{2.2}$$

Of particular interest in the acceleration phase is the velocity $v_0 = \dot{z}(t_0)$ which the phone has at the end of the phase. This value is important because it essentially determines the maximum height of the phone during the free-fall phase. It can be obtained by integrating the applied acceleration over the length of the acceleration phase:

$$(2.1), (2.2) \Longrightarrow \dot{z}(t_0) = \int_0^{t_0} a(t)\, dt = v_0 \tag{2.3}$$

Free-fall Phase ($t > t_0$)

The model during free-fall is depicted in Figure 2.2. The only force exerted on the phone during free-fall is the gravitational force resulting in a constant acceleration $-mg$ (solid arrow). Additionally, the mass has an initial velocity v_0 (dashed arrow) caused by the throw motion during the acceleration phase. The Newton equation in this case gives rise to the initial value problem

$$\begin{aligned} m \cdot \ddot{z}(t) &= -mg \qquad\qquad t \geq t_0 \\ \dot{z}(t_0) &= v_0 \\ z(t_0) &= z_0 \end{aligned} \tag{2.4}$$

with the solution

$$\dot{z}(t') = -gt' + v_0 \qquad\qquad t' = t - t_0 \geq 0 \tag{2.5}$$

$$z(t') = -\frac{g}{2}t'^2 + v_0 t' + z_0 \qquad\qquad t' = t - t_0 \geq 0 \tag{2.6}$$

where $t' = t - t_0$ denotes the time since the take-off happened. Equations (2.5) and (2.6) fully characterize the translative motion of the phone in free-fall. For example the time t_p where the phone reaches the peak point of its trajectory is given by the condition $\dot{z}(t_p) = 0$ and yields

$$\begin{aligned} \dot{z}(t_p) = 0 &= -gt_p + v_0 \\ \Leftrightarrow \quad t_p &= \frac{v_0}{g}. \end{aligned} \tag{2.7}$$

Hence, the peak height is given by

$$\begin{aligned} z(t_p) - z_0 &= -\frac{g}{2}t_p^2 + v_0 t_p \overset{2.7}{=} -\frac{g}{2}\frac{v_0^2}{g^2} + v_0\frac{v_0}{g} \\ &= \frac{v_0^2}{2g}. \end{aligned} \tag{2.8}$$

Figure 2.3 summarizes the values derived in this section on exemplary medium throw accelerations. In the graph the acceleration (blue) as well as the phone height (green) are plotted over time. The shaded area under the acceleration curve represents the throw velocity which the phone has at time t_0.

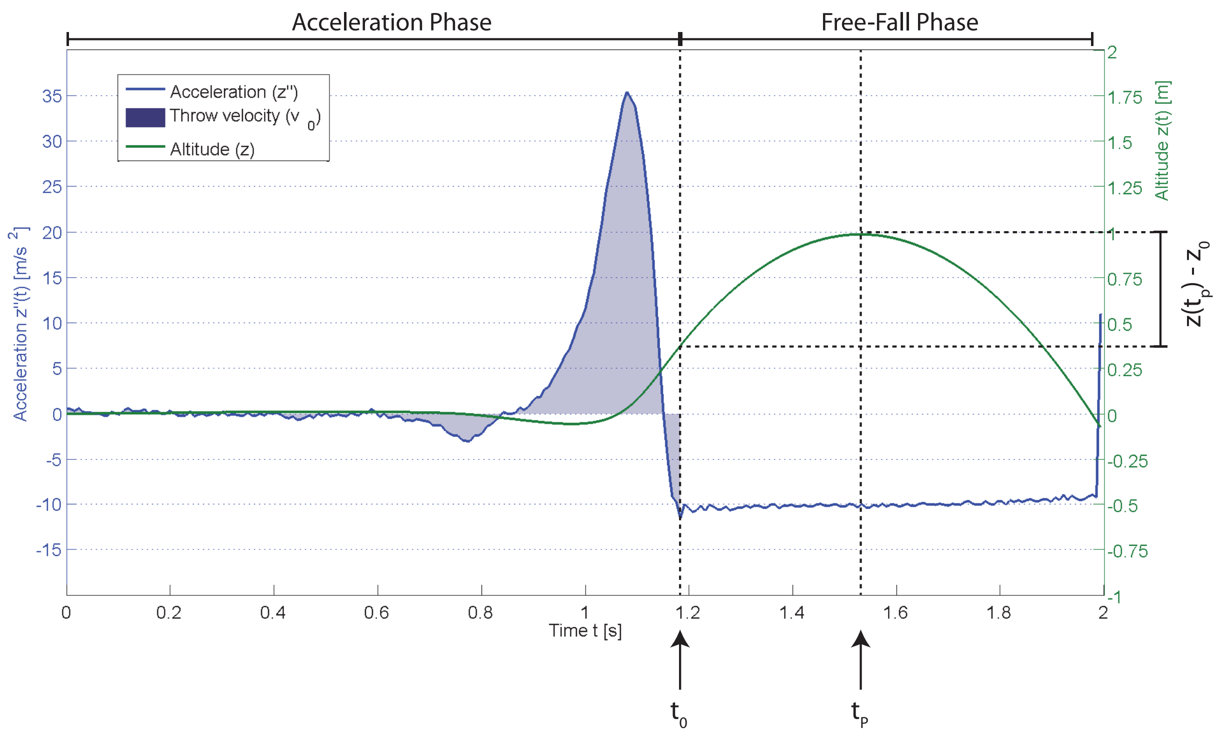


Figure 2.3: Coordinate acceleration and displacement during throw motion and free-fall

CHAPTER 3

ThrowMeApp

The goal of the Android application *ThrowMeApp* is to enable users to take aerial photographs by simply throwing the smart phone in the air. Also, the user should be able to determine the direction at which the photo is supposed to be taken.

The realization of this task involves the implementation of a motion aware system employing the motion sensors provided on a typical smart phone. In particular the accelerometer senses the translative dynamic component whereas the gyroscope provides information about the rotation of the phone. The app has to process these sensor measurements in real-time and invoke a photo capture event whenever suitable. This is when the desired orientation is reached and the phone is as close to the highest point of its motion as possible. Therefore, the core tasks of the application are data acquisition, data processing and feedback on a graphical interface.

These three core tasks manifest themselves in the structure of the program which is based on the Model-View-Controller (MVC) design pattern. In the MVC approach the model component represents the state of the application. The view component displays the information to the user. The controller component takes the users input from the view component and updates the model.

In the case of the ThrowMeApp application, the model is responsible for data acquisition and storage. The controller implements the motion detection algorithm and the view handles the graphical user interface. The communication between the components happens via a cascade of listeners which allows for an event driven application flow. Hence, a reevaluation of the model by the controller is timed by the sampling rate of the sensor. The structure of the application is depicted in Figure 3.1 where the MVC pattern is visualized in different colors. The arrows represent function calls which are responsible for the information flow from one block to another. There, the cascade of listeners is clearly visible.

The following sections further elaborate on the three software components mentioned above as well as their interaction with each other and the Android API. Furthermore, a number of limitations that have been encountered during the

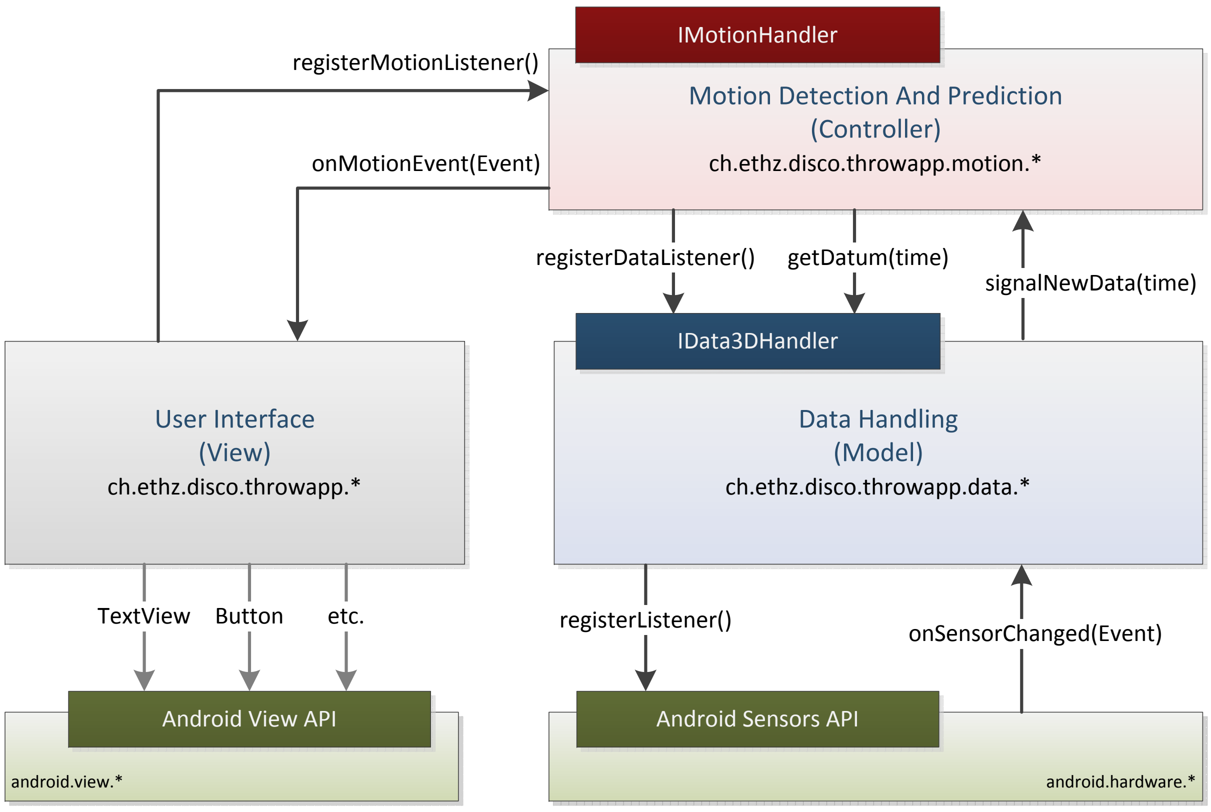


Figure 3.1: Structure of ThrowMeApp

design process as well as their implications on the resulting application are described.

3.1 The Android Hardware API

Before going into specifics of the implementation of the ThrowMeApp a short preliminary introduction to the Android Hardware API is given in this section. It serves to understand the underlying concepts of the hardware abstraction layer implemented in the Android operating system. These concepts together with their inherent limitations propagate throughout the whole application and have an impact on design choices being made. This applies specifically for the *Camera API* and *Sensors API* which are particularly relevant for ThrowMeApp.

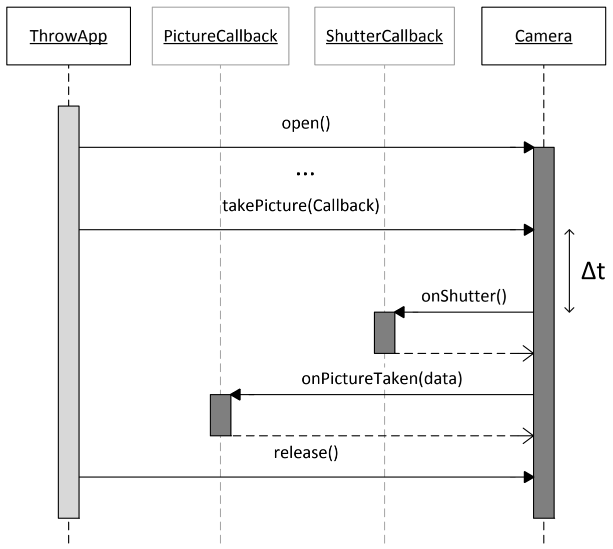


Figure 3.2: A typical API call sequence for taking a photograph. $\triangle t$ denotes the camera delay.

3.1.1 Camera API

The camera is a shared resource. Therefore, it is necessary to acquire the camera semaphore when the application is running in the foreground and release it when the application is put into the background. How this is done is depicted in Figure Figure 3.2. The acquisition and release of the semaphore is done by calling the `open` and `release` functions which are provided in each `Camera` class. Once the ownership of the camera is established photo capture events can be invoked by calling `takePicture`. This function takes a number of callback objects as parameters. The callback objects have to implement interfaces which are defined in the Android API.

The API provides two callback interfaces: `ShutterCallback` and `PictureCallback`. The `PictureCallback` interface includes the method `onPictureTaken` which is invoked when the picture data is available. The method `onShutter` is provided by the `ShutterCallback` interface and is called when the photo is being taken.

In order to configure the behavior of the camera, a wide rage of parameters can be set, for example the picture format and size as well as focus and flash modes. Despite the seemingly flexible design the Camera API provided in the latest version, there are two limitations with great influence on this work.

Camera Limitation 1: Camera Auto Settings

Firstly, the Camera API fails to provide control over the essential and typical photography settings such as ISO and exposure. The only way to influence these values is provided indirectly through a variety of *scene modes*. However, the use case of the ThrowMeApp is not covered sufficiently by any of the modes. Yet, simply choosing a short enough exposure time and an appropriate ISO value would already greatly diminish the effect of motion blur. Unfortunately, the API does not provide the means to set these parameters.

Camera Limitation 2: Camera Delay

Secondly, the API only vaguely specifies the point of time when the `onShutter` function call happens which is crucial for a time critical application such as ThrowMeApp. Literally it states: "[The method is] called as near as possible to the moment when a photo is captured from the sensor. [...] This may be some time after the photo was triggered, but some time before the actual data is available"[6]. The time between the `takePicture` method invocation and the `onShutter` callback represents the delay between a photograph being issued and actually being captured. This delay has to be taken into account to ensure that the picture is taken at the right moment during the flight.

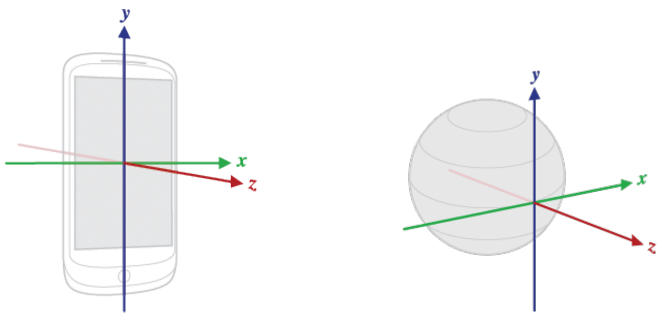
3.1.2 Sensors API

Additionally to the camera ThrowMeApp makes use of sensors. Android provides access to the typical sensors found in a smart phone, such as the magnetic field sensor, the gyroscope and the accelerometer. Additionally a number of software sensors are available, for example the linear acceleration and the rotation vector. Such sensors are implemented in software and calculate each sample using a combination of hardware sensor measurements. To read the measurements the API provides a listener interface defining the `onSensorChanged` method which is called every time a new sample from the specified sensors is available.

The motion detection algorithm of ThrowMeApp uses the accelerometer to track the translative motion of the phone. The orientation is read from the rotation vector which is a software sensor calculating the orientation of the phone from accelerometer and gyroscope data.

Accelerometer

The accelerometer measures proper acceleration (as described in Section 2.1.1) along the three axes of an inertial Cartesian coordinate system depicted in Figure 3.3(a).



(a) Sensor coordinates (b) World coordinates

Figure 3.3: Sensors API coordinate systems

The speed at which the values are polled from the sensor can be specified by the user.

Rotation Vector

The values of a rotation vector represent the absolute orientation of the phone in world coordinates. The function `getRotationMatrixFromVector` which is provided in the Android API converts these values into a three dimensional rotation matrix Q. This matrix maps a vector from the sensors coordinate system into the world's coordinate system. Figure 3.3(b) shows how the world's coordinate system is defined in the API.

Sensor Limitation: Saturation

The configuration possibilities for the sensors are very restricted, introducing a major limitation to the application. For example most accelerometers used in smart phones offer configurable dynamic ranges from $\pm 2g$ (meaning twice the gravitational acceleration into each direction of each axis) up to $\pm 8g$ or higher. However, in most cases the vendor's firmware limits the dynamic range to $\pm 2g$ to achieve the highest sensitivity possible. Consequently, the sensor quickly saturates during typical throw motions where acceleration magnitudes above $2g$ are applied. In a similar manner the gyroscope is limited by a smaller dynamic range than possible.

3.2 Data Handling (Model)

The package `*.throwapp.data.*` contains the data handling block of the application. A detailed schematic representation is given in Figure 3.4. The three

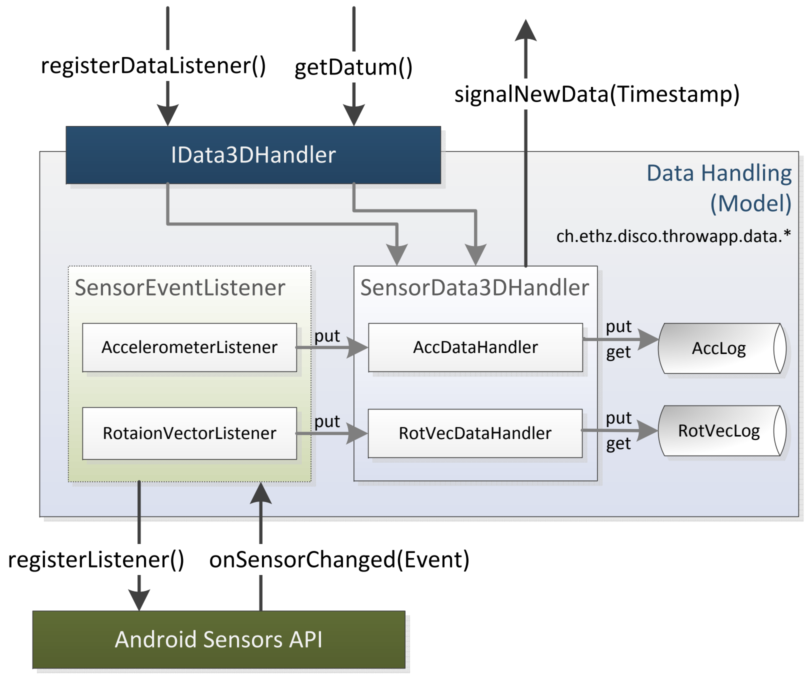


Figure 3.4: Detailed structure of the model block of ThrowMeApp

main tasks performed in this block are data acquisition, storage and provision. In reference to this figure these tasks are discussed in the following.

Data Acquisition: For each of the sensors a corresponding `SensorEventListener` is implemented. For every new sensor measurement the Android OS calls the `onSensorChanged` method which is implemented in the listeners. The datum is then relayed to the appropriate `SensorData3DHandler` for further handling.

Data Storage: The data store is handled by the `SensorData3DHandler` which accepts data from the listeners and stores them in data base. The data structure used for storage is sorted list indexed with the time stamp of the associated measurement.

Data Provision: The `IData3DHandler` provides two ways to access the stored data. One possibility is to register an instance of a `DataListener` which implements the function `signalNewData`. This function is called whenever a new datum is put into the data store. The other possibility is to access a single datum or a whole range by specifying the time or time range, respectively.

The data store is the core of the data handling block as it contains all the information about the motion of the phone. Therefore, the data handling block corresponds to the *model* in the MVC pattern. Generally, the model does not contain functions for data handling. However, in this application simple data relay functions are integrated. The advantage of this realization of the data handling block is its generic public interface. For instance it is easily possible to implement the interface using an ASCCII file of test measurements as a data source. For test driven software development this is a key feature.

3.3 Motion Detection and Prediction (Controller)

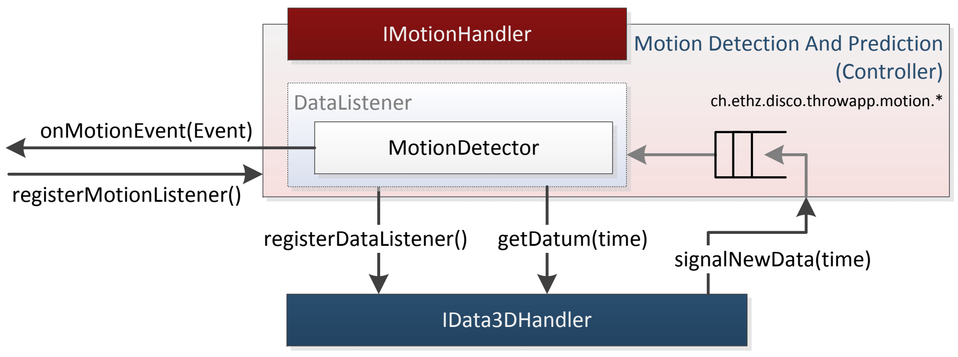


Figure 3.5: Detailed structure of the controller block of ThrowApp

The package `*.throwapp.motion.*` contains the controller block of the program. It implements the detection and decision logic which triggers motion events in real time based on the input. The data is provided by the `IData3DHandler` callback interface. Every time a new datum is signaled a reevaluation of the whole data history is performed based on a throw detection algorithm. To allow for backlog due to time consuming calculations the incoming data is buffered in a First-In-First-Out (FIFO) channel.

The `MotionDetector` defines four characteristic motion events that occur once for each throw:

Launch The launch marks the beginning of a throw motion. It is the point in time when the phone significantly increases its speed while still in the hand of the user.

Takeoff The takeoff is the moment when the phone leaves the hand and transitions into a free-fall movement.

Peak The peak is the highest point of the phone's trajectory. At that point the velocity of the device in z-direction is zero.

Touchdown At the point of touchdown the phone ends its free-fall. It is the first time since the takeoff that the phone experiences accelerations beyond the Earth's gravitation.

The fifth event does not necessarily occur on each throw but may occur multiple times per throw:

Oriented The phone is considered oriented when its orientation is aligned with the specified direction at which the picture is supposed to be taken.

The algorithms used here have been developed with the help of numerous empirical experiments and extensive sensor data analysis. The following sections give an insight into how the throw detection, peak prediction and orientation detection algorithms work.

3.3.1 The Throw Detection Algorithm

The throw detection algorithm (TDA) is responsible for detecting the launch, takeoff and touchdown events of a throw. The algorithm operates under the following premises.

1. Each throw motion has to be purely vertical such that the z-axis of the phone's and Earth's coordinate systems are aligned. Any other movement would cause a significant increase in the complexity of the algorithm because the orientation of the phone would need to be considered for the conversion between proper acceleration and coordinate acceleration. By restricting the throw motion to be purely vertical the algorithm is less complex and more accurate.
2. When the algorithm is started, the phone has to have no velocity and its display needs to be pointing towards the sky. This is the initial state that the algorithm uses as a reference.
3. The magnitude of the gravitational acceleration is $g = 9.81\frac{m}{s^2}$. This is the value used for the conversion between proper acceleration and coordinate acceleration.

In order to detect the motion events the TDA checks the accelerometer samples for certain conditions. The most recent sample as well as a history of samples are used in different stages of the algorithm. Figure 3.6 summarizes its functionalities in a flow chart.

The following methods are used to detect the corresponding events:

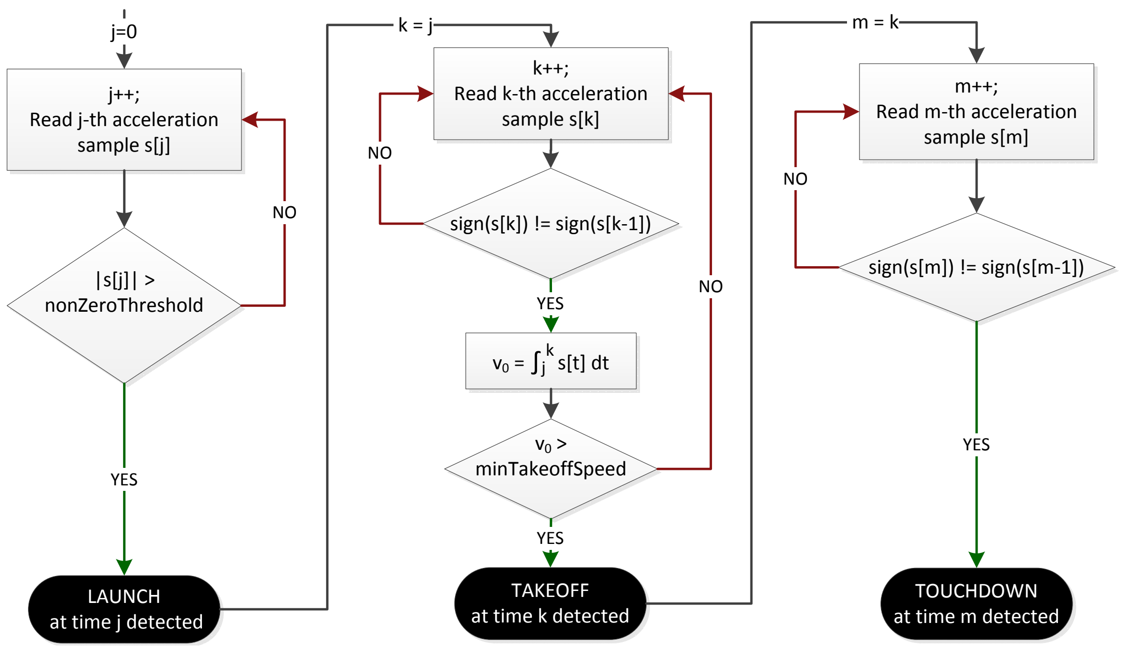


Figure 3.6: Throw Detection Algorithm Flowchart

Launch The launch is characterized by a *significant* increase in the velocity. This corresponds to non-zero accelerations that needs to be detected. However, due to the presence of sensor noise it is not sufficient to look for non-zero acceleration values. Therefore, a `nonZeroThreshold` is introduced. This threshold defines the upper bound for which an acceleration measurement is considered to be zero. Hence, the launch is detected when the magnitude of an incoming acceleration sample exceeds the `nonZeroThreshold`.

Takeoff At takeoff the force applied by the hand rapidly vanished and the acceleration therefore drops from high positive values to $-1g$. This can be detected by checking for a change of the sign of two consecutive samples. However, a changed sign may also be caused by short sudden movements during the throw motion. To disregard such movements a second parameter is introduced: the `minTakeoffSpeed`. It defines the minimum speed which is necessary for proper throw. If the integral over all acceleration samples between the launch and the sign change position is greater than the threshold the takeoff is considered valid.

Touchdown The touchdown is the inverse of a takeoff. Here, the acceleration changes from $-1g$ to some high value. In this case it is sufficient to check for the sign change of two consecutive samples as significant disturbances during free-fall are highly unlikely.

3.3.2 Peak Prediction

Due to the inherent camera delay discussed in Section 3.1.1 it is not practical to detect the time when the peak is reached in real-time. Instead, a prediction should be calculated and the delay subtracted from the prediction. Equation 2.7 provides a way of calculating the time of reaching the peak as a function of the initial throw velocity v_0 and the gravitational acceleration g. In ThrowMeApp this calculation is done as soon as v_0 is known. This is the case when the takeoff event is detected by the TDA. However, the value v_0 which is calculated in the takeoff detection loop of the TDA underestimates the actual initial velocity of the phone $\tilde{v}_0$ due to sensor saturation as discussed earlier. Surely, it is not possible to recover $\tilde{v}_0$ completely because the accelerometer fails to sense the actual acceleration applied to the device. The countermeasure to the estimation error of the initial velocity due to sensor saturation used in ThrowMeApp is described in the following.

Compensation of Sensor Saturation

The implemented approach to diminish the estimation error is to calculate a correction factor $k = \tilde{v}_0/v_0$ which is the ratio between the actual and the estimated initial velocity. $\tilde{v}_0$ can be derived from the hang time t_h which is the time the phone spend in free-fall. If we assume that the phone is caught at the same position as where it has been released into free-fall, it is possible to derive $\tilde{v}_0$ from Equation 2.6, yielding

$$\begin{aligned} & z(t_h) = z_0 = -\frac{g}{2}t_h^2 + \tilde{v}_0 t_h + z_0 \\ \Leftrightarrow \quad & \tilde{v}_0 = \frac{1}{2} g t_h. \end{aligned}$$

However, in most cases the phone takes off at a higher altitude than where it is caught due to the nature of the throw motion. This difference $\triangle z$ is considered in

$$\begin{aligned} & z(t_h) = z_0 - \triangle z = -\frac{g}{2}t_h^2 + \tilde{v}_0 t_h + z_0 \\ \Leftrightarrow \quad & \tilde{v}_0 = \frac{1}{2} g t_h - \frac{\triangle z}{t_h}. \end{aligned} \tag{3.1}$$

Using Equation 3.1 the correction factor k can be expressed in terms of the hang time t_h and the estimated initial velocity v_0 as follows

$$k = \frac{g \cdot t_h}{2v_0} - \frac{\triangle z}{t_h \cdot v_0}. \tag{3.2}$$

The calculation of k can only be done after the throw is completed and the touchdown is detected, i.e. t_h is available. Hence, it can be considered for the following throws.

In ThrowMeApp the correction factor value is stored in the shared preferences database which persists between executions of applications. For each throw the value is read from the preferences and multiplied to $\tilde{v}_0$ to obtain a better initial velocity estimate needed to calculate an accurate peak time prediction. After a throw is performed, a new correction factor is calculated, low-pass filtered and stored back into the preferences database. Figure 3.7 shows a block diagram of the method by which the correction factor `corrFactor` is computed using a newly calculated `newCorrFactor`. Basically it is an IIR low-pass filter implemented in *Direct Form 1* with coefficients $b_0 = 0.6$ and $-a_1 = 0.4$. The preceding saturation block stabilizes the output by limiting the input to values between 1 and 5.

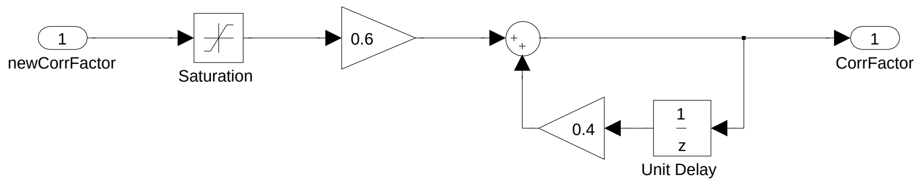


Figure 3.7: Correction Factor update function as a block diagram

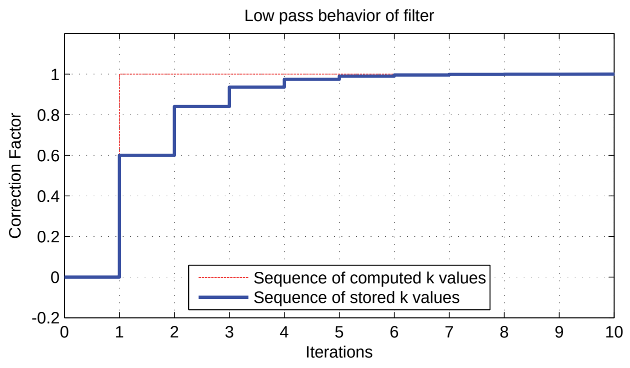


Figure 3.8: Step response of the IIR filter showing its low-pass characteristic

The purpose of the low-pass characteristic is to avoid large changes of k between two successive throw iterations. For example an erroneous value could arise when the phone is not caught but let drop on the ground after free-fall. This would spoil the peak calculation for the next throw if the value of the correction factor would be adopted. By averaging between several successive values the effect of one erroneous instance is reduced. This is illustrated in Figure 3.8 showing the step function of the filter. The step simulates a rapid transition of calculated k values between the first two throws followed by a constant value of 1 (thin line). The bold line shows the progression of the actual value used in each throw, which adopts the step only after four iterations. Hence, the correction factor k represents the characteristics of the five preceding throws.

Once k is determined by the described method, the time at which the peak is reached, is best approximated by

$$t_p = \frac{k \cdot v_0}{g}.$$

3.3.3 Orientation Detection

Next, the orientation detection mechanism is explained. It is the method responsible for detecting the desired direction at which the picture is to be taken.

The direction to which the camera points is specified by the vector $(0, 0, -1)$ in the phone's inertial coordinates (Figure 3.3(a)). To identify how the camera is oriented in the world's coordinate system, this vector has to be multiplied with the rotation matrix Q as discussed in Section 3.1. Therefore, the negated third column of the rotation matrix $-(q_3, q_6, q_9)$ has to be compared to a specified direction vector in world coordinates. In ThrowMeApp the vector $-(q_3, q_6, q_9)$ is compared to $(0, 0, -1)$ which is a vector pointing towards the ground.

However, it is not feasible to check the two vectors for equality because a prefect match is rarely achieved. Therefore, a tolerance `orientationTol` is introduced. This value specifies the maximum Euclidean distance between the two vectors for which they are considered equal.

3.4 User Interface (View)

The user enters the ThrowMeApp application through the `MainActivity` which displays basic usage instructions. From there the user can adjust settings in the `SettingsActivity` or initiate the `ThrowActivity`. Depending on whether a photograph has been taken by the `ThrowActivity`, the user is displayed the picture in the `ResultView` or given an error message in the `FailureActivity`, respectively. Figure 3.9 shows the typical GUI path in solid arrows and the exceptional GUI path in dashed arrows. Each stage on a path extends the `android.app.Activity` class and implements view components provided by the Android View API, such as TextViews, Buttons or Surfaces. The following paragraphs give a more detailed description of each of the five activities.

SettingsActivity The `SettingsActivity` provides means to adjust application specific preferences. Two GUI control parameters can be set here:

1. **Prompt Throw Details**: Activate to show diagnostic statistics about the throw that has been performed. This includes throw velocity, predicted peak time, predicted peak height, hang time, etc.

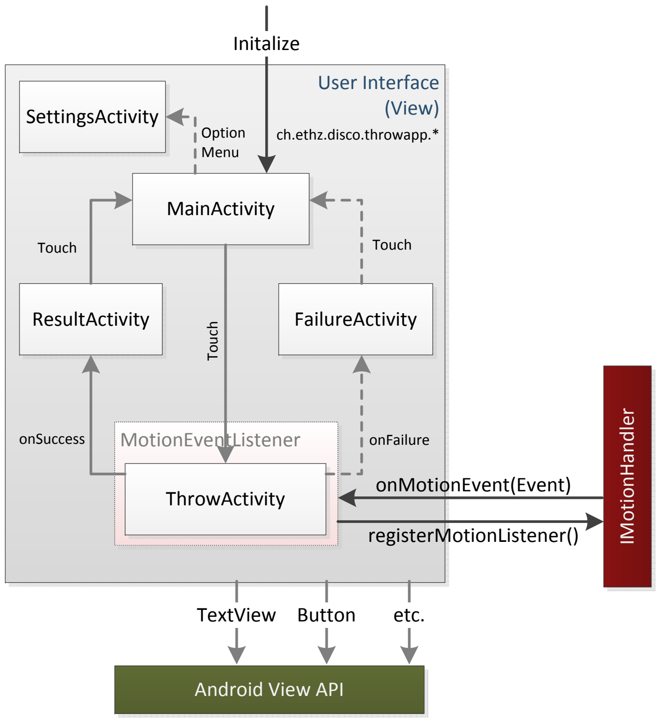


Figure 3.9: Detailed structure of the view block of ThrowApp

2. **Detect Peak**: Activate to enable an audible feedback at the predicted peak point.

MainActivity This is the landing activity which is shown to the user when ThrowMeApp is started. Short usage information is displayed here. By touching the screen the application proceeds to the `ThrowActivity`.

ThrowActivity This activity shows the preview of the camera in full screen. Additionally, it implements the `MotionEventListener` interface from the `*.throwapp.motion.*` package and registers itself as a listener to the `MotionHandler`. Thus, detected motion events are signaled to this activity which performs the appropriate actions. For example a beep sound is played at the predicted peak time and the camera is triggered when correct orientation is detected. An incoming touchdown event causes the application to proceed to the `ResultActivty` in case pictures have been taken. If no pictures were taken the `FailureActivity` is executed instead.

ResultActivity The `ResultActivity` displays the resulting photograph.

FailureActivity Here an error message is displayed and the user can proceed to the `MainActivity` to try again.

Chapter 4

Experiments and Results

ThrowMeApp has been tested on two different devices:

1. Samsung Galaxy S III (GT-I9300)
   - Android Version: 4.0.4
   - Accelerometer/Gyroscope: STMicroelectronics LSM330DLC
2. Samsung (Google) Nexus S
   - Android Version: 4.0.4
   - Accelerometer: STMicroelectronics KR3DM
   - Gyroscope: STMicroelectronics K3G

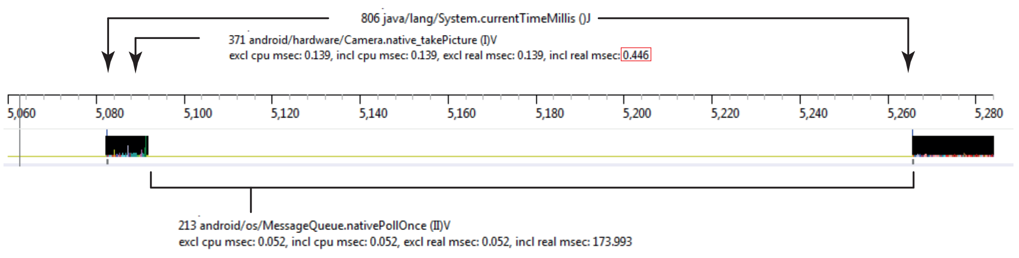
Mostly, ThrowMeApp behaves equally well on both phones since they run the same version of Android. However, different behavior may occur due to differences in hardware (especially Sensors, Camera, CPU) and their driver implementations. Most noticeable are the performance gaps between the two device in graphical interface transitions and camera delays. The experiments described in this chapter have been performed on both phones. The results are generally platform independent. However, some relevant differences in the outcomes are pointed out where applicable.

A number of experiments have been conducted in order to assess the functionality and feasibility of the implementation of ThrowMeApp. First, the camera delay, as discussed qualitatively in Section 3.1.1, is put into numbers. Then, sensor data of a typical throw is shown and the accuracy of the peak detection mechanism is examined. This is followed by a presentation of the rotation dynamics during a typical throw and two special cases of stable and unstable rotations. Finally, a set of resulting pictures are presented.

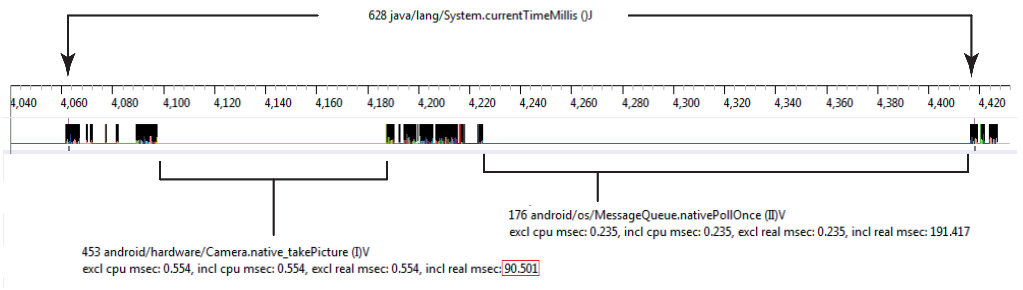
4.1 Quantifying Camera Delay

The camera delay is an inherent limitation of the system. As ThrowMeApp relies on a fast response to a `takePicture` command, it is worth estimating the duration and behavior of the delay under different conditions. For that purpose the diagnostic Android application `CameraDelayTester` was developed. This App triggers the camera every three seconds in an infinite loop. It measures the delay between the event being triggered and the `onShutter` callback being executed and displays the result. When the App is exited a log is stored to the file system containing two values for each measurement. One represents the delay as described above in milliseconds and the other is the delay minus the exposure of the corresponding photo. The latter is called *normalized delay*.

As the first step of the camera delay identification experiment, the execution profile of the `takePicture` method called in the `CameraDelayTester` App is examined. Figures 4.1(a) and 4.1(b) show common execution profiles of the Galaxy S 3 and the Nexus S, respectively.



(a) Galaxy S 3



(b) Nexus S

Figure 4.1: The execution profile during a `takePicture` method call

The Galaxy execution trace indicates an approximately 174 ms long execution period of the method `MessageQueue.nativePollOnce` between the time measurements performed by `System.currentTimeMillis`. The `Camera.native_takePicture` call, however, is neglectable with an execution time of approx. 446 μs. For the Nexus S the execution of the `MessageQueue.nativePollOnce` method takes approximately 191 ms. The `Camera.native_takePicture` method executes in approximately 90.5 ms.

The second step is to monitor the camera delay behavior over a number of successive executions under different conditions. Experiments with high sample sizes have been performed on both phones and under good and poor light conditions. Good light conditions simulate the normal mode of operation with typically short shutter speeds whereas poor light conditions entail long shutter speeds. The camera delay and the normalized delay have been recorded. A complete set of histogram plots of the data is provided in Appendix B. In Table 4.1 the statistics of the gathered measurements is summarized.

		Galaxy S 3		Nexus S	
		good light	poor light	good light	poor light
Mean exposure	μ_e	1/165 s	1/17 s	1/136 s	1/13 s
Sample size	n	565	769	525	308
Mean delay	μ_d	102.1 ms	158.8 ms	233.4 ms	356.6ms
Std. deviation	σ_d	9.8 ms	17.9 ms	28.9 ms	33.4 ms

Table 4.1: Summary of camera delay statistics

4.2 A Typical Throw

Now, that the camera delay is measured it is practicable to examine the speeds present in a typical throw. This is where the Android application `Sensations` comes into play. The App displays live measurements from a number of relevant sensors and stores their values to the file system in the form a comma separated values (csv) file.

In this section the acceleration during a typical medium height throw is considered. Appendix C includes a full motion study for this particular throw. Figure 4.2 shows the resulting acceleration measurements from two sensors. The measurements in the graph are already transformed into coordinate acceleration by a subtraction of $g = 9.81$ m/s^2.

As the range of the internal sensor is limited to $\pm 2g$ the saturation starting at second 1 can be clearly observed. The external sensor with a greater dynamic range measures accelerations up to $3.5g$. The shaded area under the curve represents the initial velocity v_0. In this throw the actual initial velocity derived from unsaturated measurements is $\tilde{v}_0 = 3.7$ m/s. However, the area under the saturated internal accelerometer measurements curve is $v_0 = 1.45$ m/s. This yields a correction factor

$$k = \frac{3.7 \text{ m/s}}{1.45 \text{ m/s}} \approx 2.55.$$

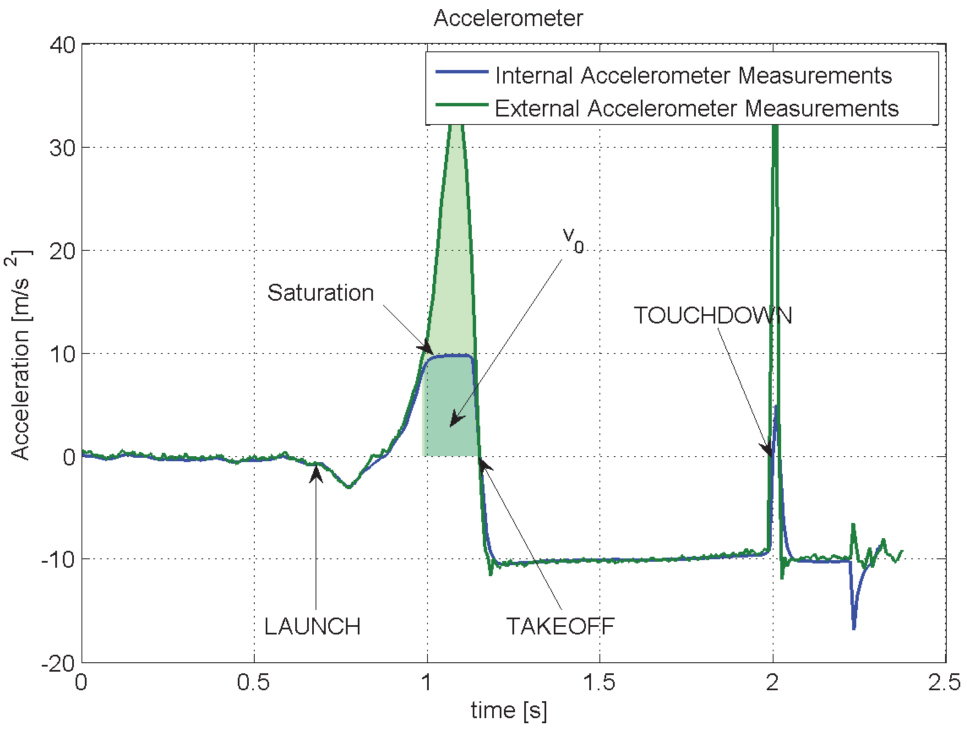


Figure 4.2: Acceleration during medium height throw. Internal sensor saturates.

The predicted peak time t_p and height $z(t_p)$ for this example, without the use of the correction factor, are calculated by Equation 2.7 and yield

$$t_p = \frac{v_0}{g} = \frac{1.45\frac{\mathrm{m}}{\mathrm{s}}}{9.81\frac{\mathrm{m}}{\mathrm{s}^2}} \approx 147.8\ \mathrm{ms}$$

$$z(t_p) = \frac{v_0^2}{2g} = \frac{1.45^2\frac{\mathrm{m}^2}{\mathrm{s}^2}}{2\cdot 9.81\frac{\mathrm{m}}{\mathrm{s}^2}} \approx 10.7\ \mathrm{cm}.$$

The Accuracy of peak prediction is further analyzed in the following section.

4.3 Peak Prediction Accuracy

The accuracy prediction of the peak time and height depends heavily on the correct value of the initial takeoff speed v_0. Especially, for the height any error influences the result quadratically. Therefore, a correction factor has been introduced as described in Section 3.3.2.

To evaluate the effect of the correction factor on the estimation error of the peak height prediction, a sequence of experiments have been conducted. Each experiment consisted of a medium height vertical throw which was recorded on video. After each throw the predicted peak height provided by ThrowMeApp was compared to the actual peak height which could be measured with the help of the video sequence. However, due to motion blur in the video recording the takeoff position can only be measured with low precision. The effect of motion blur is shown in Figure 4.3.

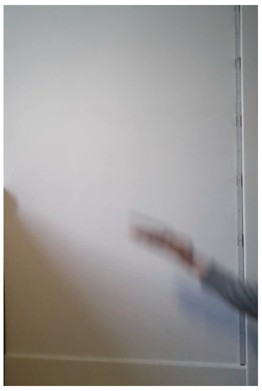


Figure 4.3: Motion blur in the video recording

Figure 4.4 shows the results from an experiment with ten consecutive throws of roughly the same height. The graph on the left indicates the predicted height, video-measured height with 5 cm precision tolerance and the error where no correction factor was used. The graph on the right shows the same with the correction factor being used.

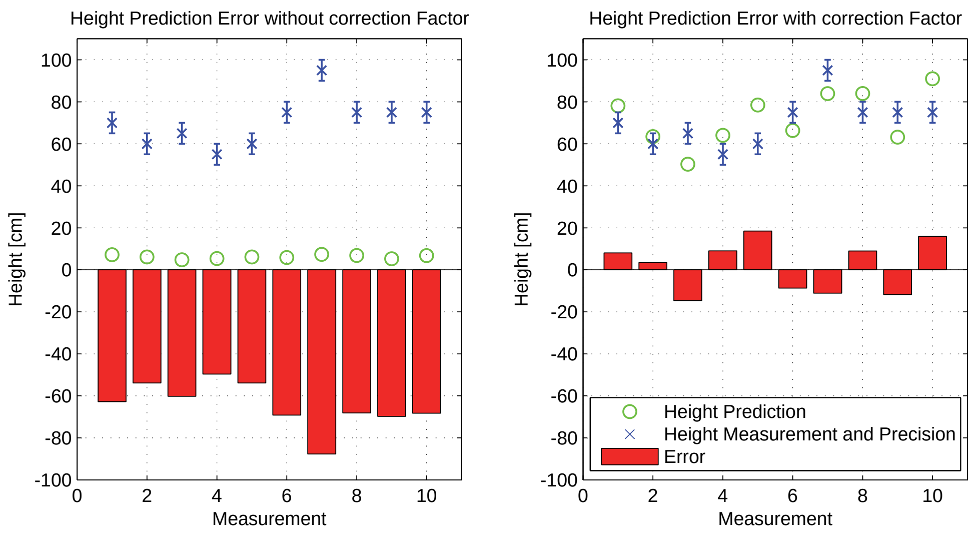


Figure 4.4: Effect of correction Factor on height prediction Error

The average error without correction in place is $\overline{E} = 64$ cm. The use of the proposed correction method reduces the average error by about 83% to $\overline{E}_c = 11$ cm.

4.4 Rotation Stability

In addition to the position of phone its orientation needed to be considered as well when taking the photo. This includes the use of the gyroscope to track the phone's rotation. However, during the course of this work it turned out that it is not possible to trigger a photo by analyzing gyroscope data in real-time. Rather the times of reached desired orientation has to be known prior to triggering the photo event to account for the camera delay. A simple experiment serves the purpose to analyze whether this is a feasible approach.

In order to predict the rotation of the device one has to assume that during free-fall no net momentum is applied to the mass, thus the angular momentum L is constant. This implies that the magnitude and direction of the total angular velocity ω remains constant throughout the free-fall because $\mathbf{L} = I\omega$ holds and the moment of inertia does not change.

In order to investigate the stability of the rotation axis raw gyroscopic data as well as the rotation matrix for two exemplary throws have been acquired. Figures 4.5 and 4.6 show the results of attempted pure y-axis and x-axis throw rotations, respectively. The top-left graphs show the raw angular velocities measured by the g. The top-right graphs show the direction of the third column of the rotation matrix which is the direction of the camera evolving of the course of the throw. The bottom two graphs show the direction of the rotation axis in phone and world coordinates over time.

For this experiment it can be observed that the rotation axis in world coordinates of the y-axis rotation depicted in Figure 4.5 remains fairly constant. However, the rotation axis in world coordinates of the x-axis rotation clearly changes its direction.

4.5 Camera Pictures

The pictures taken by ThrowMeApp suffer from motion blur. A sample photograph shown in Figure 4.7 exhibits this distortion even at bright light conditions.

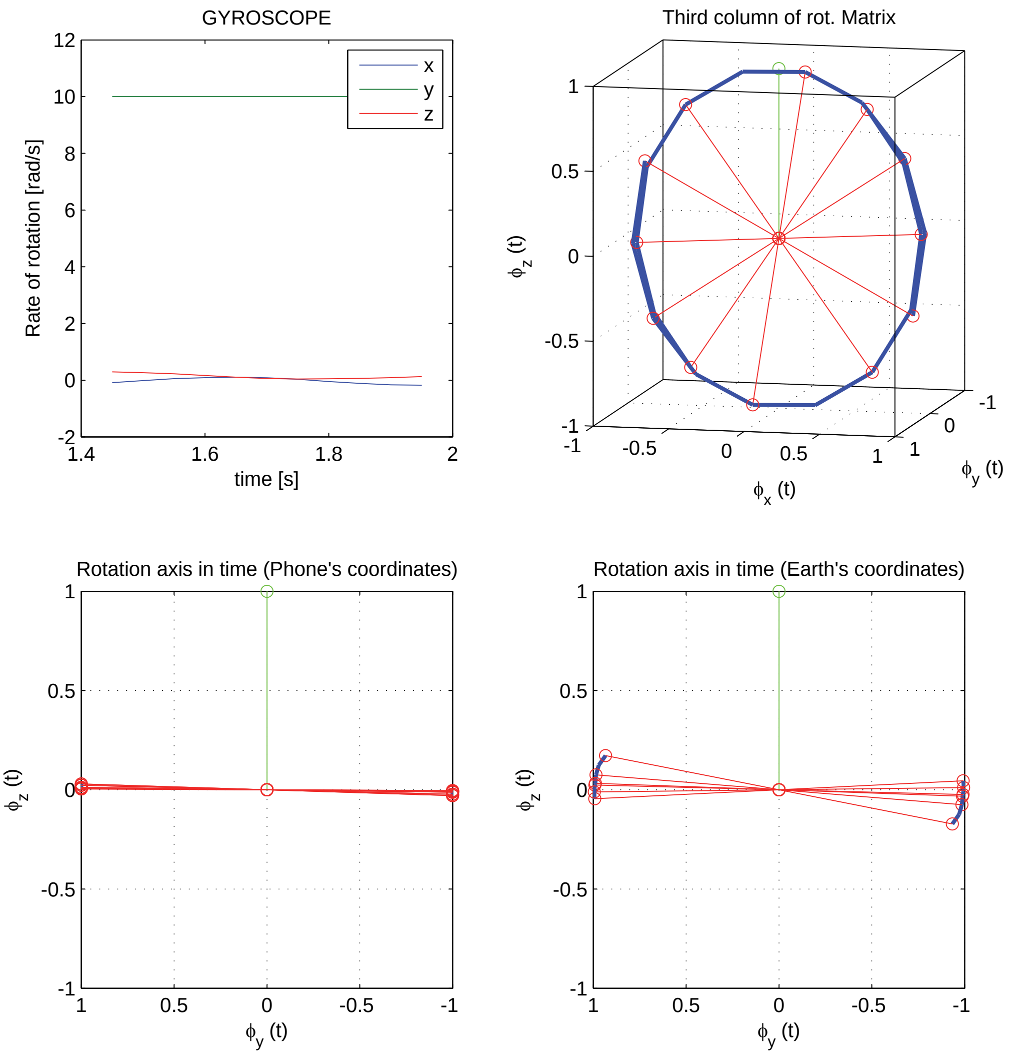


Figure 4.5: Constant rotation axis in y-direction

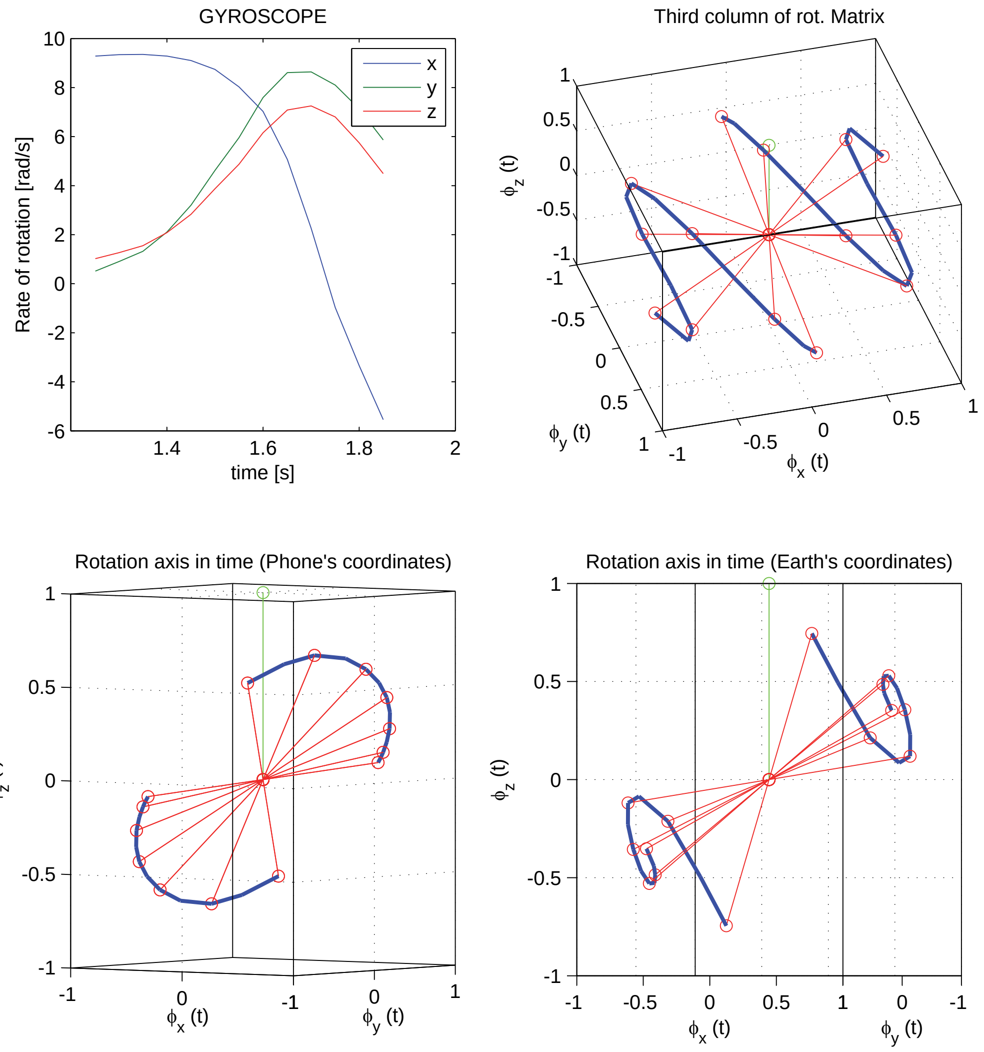


Figure 4.6: Variable rotation axis in x-direction

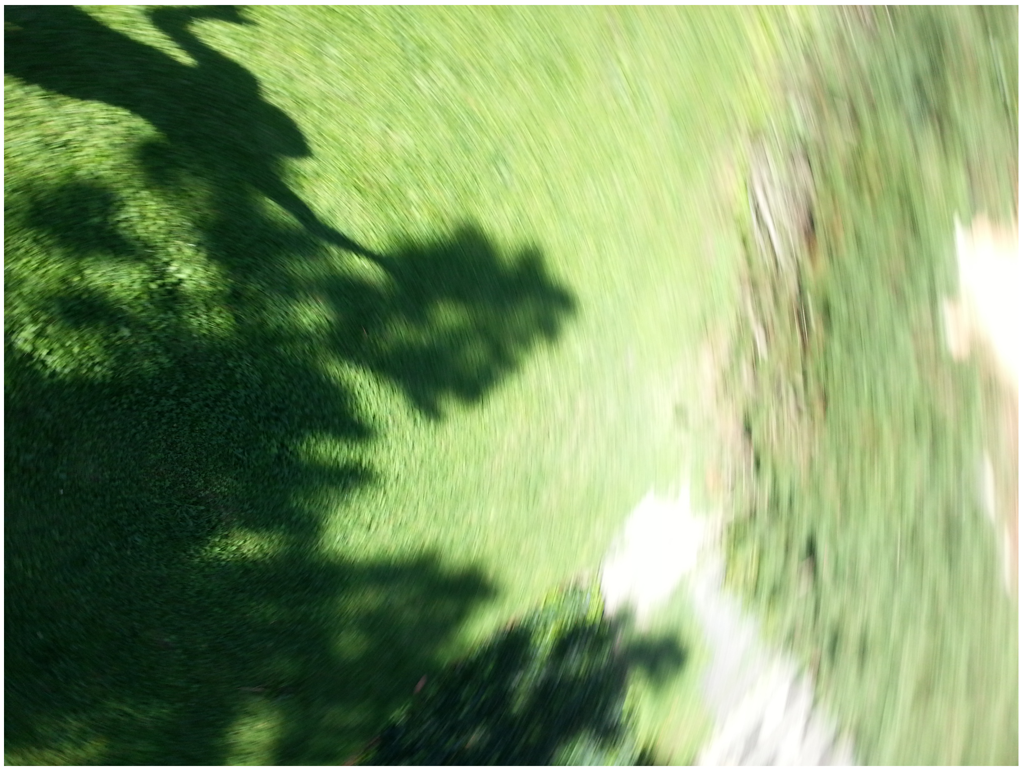


Figure 4.7: Motion blur even at good light conditions

Chapter 5

Discussion and Conclusion

So far the theoretical background has been derived and the functionality of ThrowMeApp has been described and tested. The following discussion summarizes the findings of this work and puts them into perspective for future applications.

5.1 Discussion of Results

5.1.1 Accuracy of Motion Detection

A method for real-time motion detection of a throw motion in the presence of saturated sensor measurements has been proposed in this work. The accuracy of the motion detection and the prediction of the peak height have been evaluated. An error reduction of 83 % was achieved by using the proposed sensor saturation countermeasure for the peak height prediction. The peak height was estimated with an accuracy of 11 cm for a sequence of similar throws. This value is sufficient to produce acceptable results for the standard use case of the ThrowMeApp application. However, the user will have to perform several throws in a row in order to experience better peak detection precision.

A rotation prediction mechanism could not be reliably implemented using the orientation sensing provided in the Android API. The results in Section 4.4 show that the measured rotation axis of a rotation in free-fall does not guarantee to remain in constant direction during free-fall. This inhibits the extrapolation of the rotation dynamics. The poor orientation sensing can have numerous reasons one of which is certainly the insufficient orientation sensing facilities provided by the Android API.

5.1.2 Timing and Delays

The camera delay was already identified as a major source of trouble in an early stage of the project. An analysis of its scale revealed dependencies with the phone's hardware and the driver's implementations making it impossible to compensate the lag with a constant time value.

In addition, the camera lag prohibits reasonable orientation awareness in the ThrowMeApp. Even with a high-end phone and best light conditions a delay of 100 ms has to be taken into account. For a phone rotating at moderate 10 rad/s this already means a nearly 60° angular displacement between an orientation is detected and the picture is taken.

5.1.3 Resulting Picture Quality

In ThrowMeApp great effort was made to increase the picture quality and reduce motion blur. This includes and is unfortunately limited to setting a certain scene mode and tweaking the exposure compensation. Besides these two functions the API does not provide any means to set parameters like the ISO value or the shutter speed which are essential for keeping motion blur at minimum even with the risk of underexposed pictures.

With some practice the photo quality can be increased, for example by trying to throw the phone with as little rotation as possible. Also, the app should be used outside during bright sun light. This allows the auto settings of the phone to adjust for hight shutter speeds which reduces motion blur.

5.2 Conclusion and Outlook

The goal of this work was to create an application which enables users to take aerial photos with their smart phone by throwing it in the air. In essence, ThrowMeApp delivers this functionality. With ThrowMeApp even multiple aerial pictures can be taken with one touch and one throw. With a little practice and sunshine the quality can be enhanced satisfying results can be produced.

The major limitations of this work were the Android API and the lack of control over low level hardware operations. If one chooses to go beyond what is offered in the Android SDK it is possible to solve at least two remaining problems. Firstly, the dynamic range of the sensors can be set by hardware register manipulation on the firmware level. Secondly, additional camera settings such as ISO settings are provided in the JNI (Java Native Interface) portion of the Android operating system. Getting access to these functions via the SDK or outside of it will provide means to reduce motion blur in the pictures.

Bibliography

[1] Pfeil, J., Hildebrand, K., Gremzow, C., Bickel, B., Alexa, M.: Throwable panoramic ball camera. In: SIGGRAPH Asia 2011 Emerging Technologies. SA '11, New York, NY, USA, ACM (2011) 4:1–4:1

[2] Kuwa, T., Watanabe, Y., Komuro, T., Ishikawa, M.: Wide range image sensing using a thrown-up camera. In: Multimedia and Expo (ICME), 2010 IEEE International Conference on. (july 2010) 878 –883

[3] Maluf, N., Williams, K.: Introduction to Microelectromechanical Systems Engineering. Microelectromechanical Systems Series. Artech House (2004)

[4] Samuels, H.: Single- and dual-axis micromachined accelerometers (1996)

[5] Sethuramalingam, T., Vimalajuliet, A.: Design of mems based capacitive accelerometer. In: Mechanical and Electrical Technology (ICMET), 2010 2nd International Conference on. (sept. 2010) 565 –568

[6] Google: Android API reference - Camera.ShutterCallback. https://developer.android.com/reference/android/hardware/Camera.ShutterCallback.html (09 2012)

Appendix A

Complete UML Diagram of ThrowApp

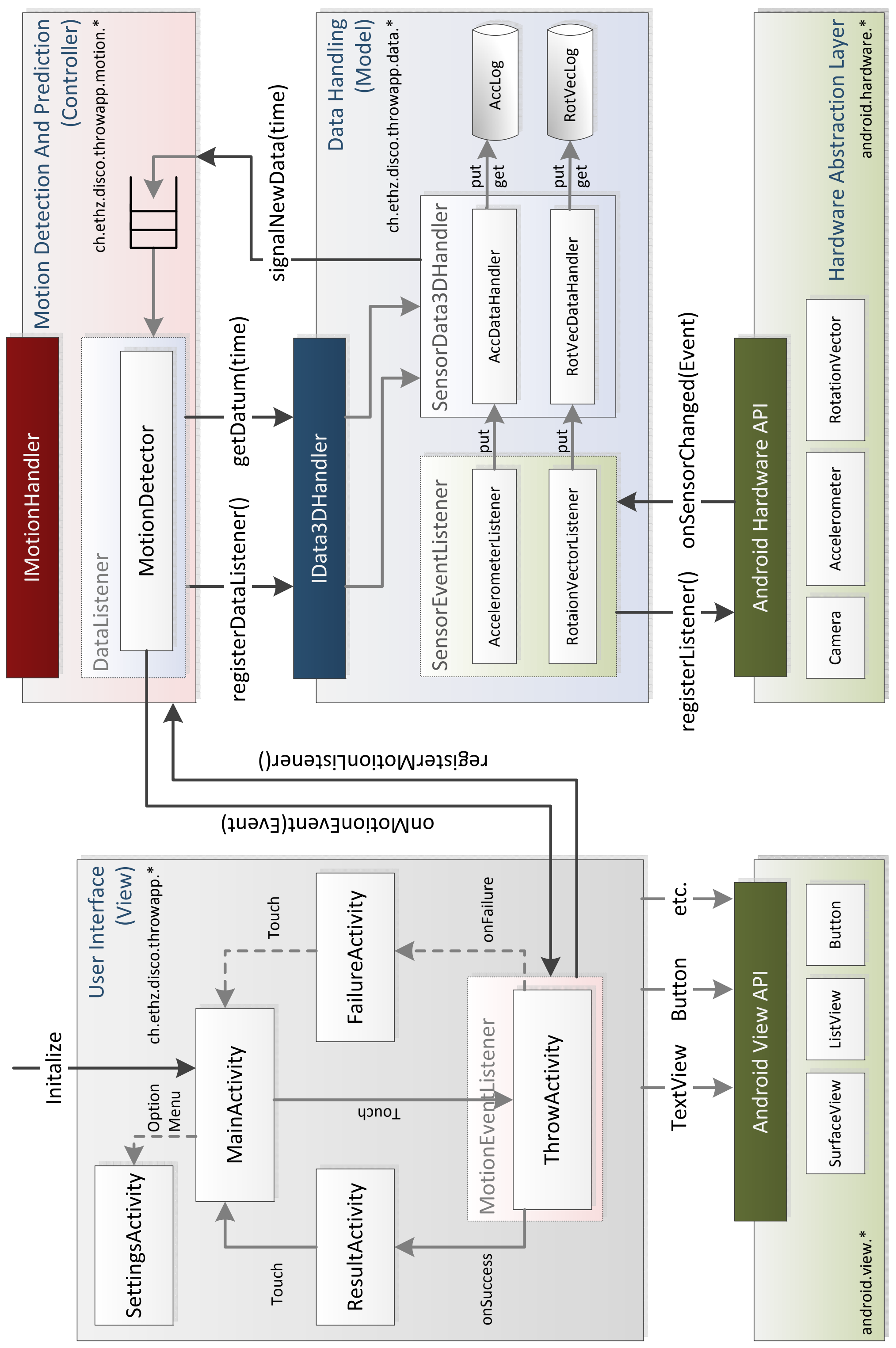


Figure A.1: CompleteUML diagram of ThrowApp

Appendix B

Full Statistics of Camera Delay

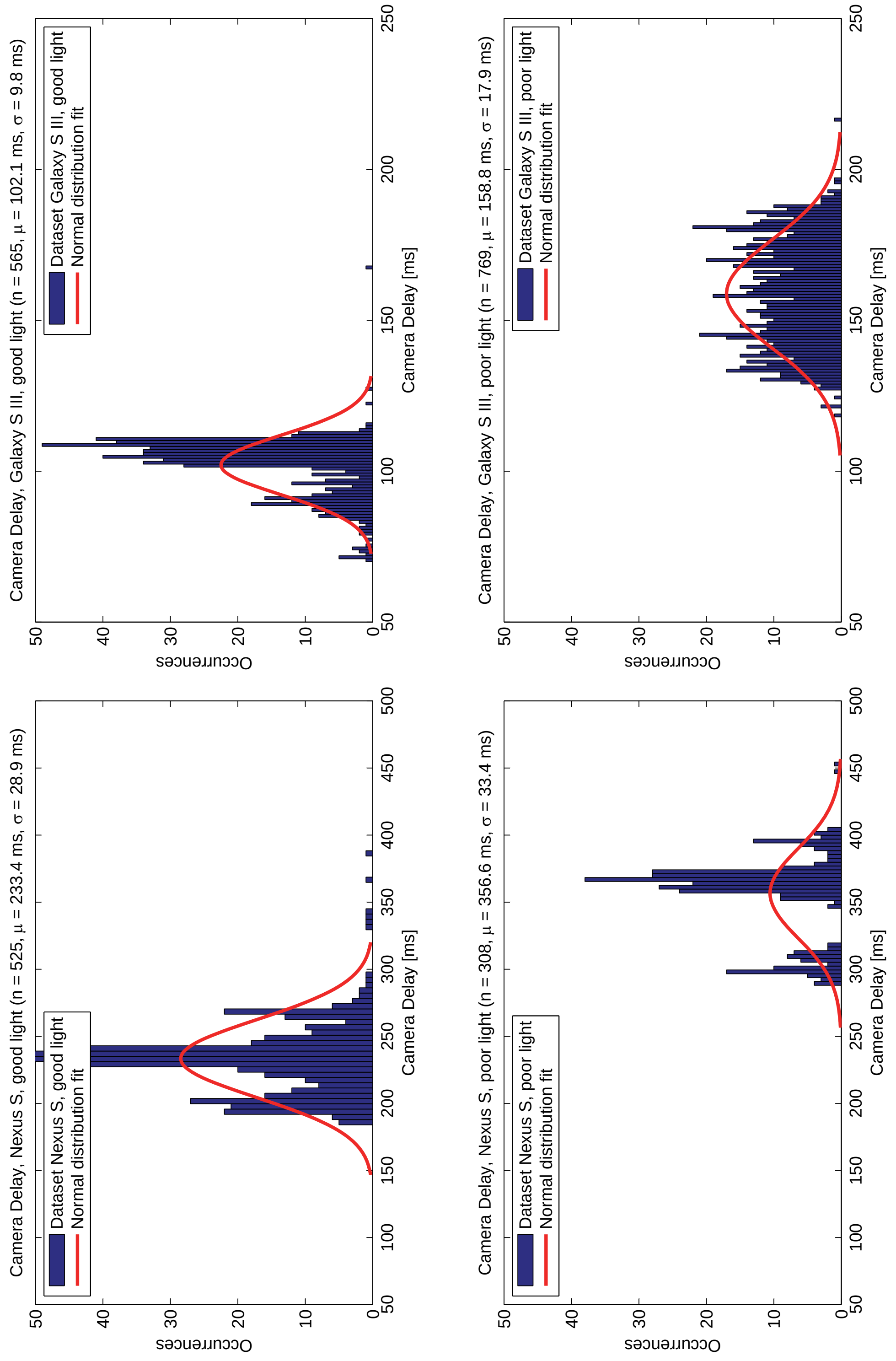


Figure B.1: Camera Delay

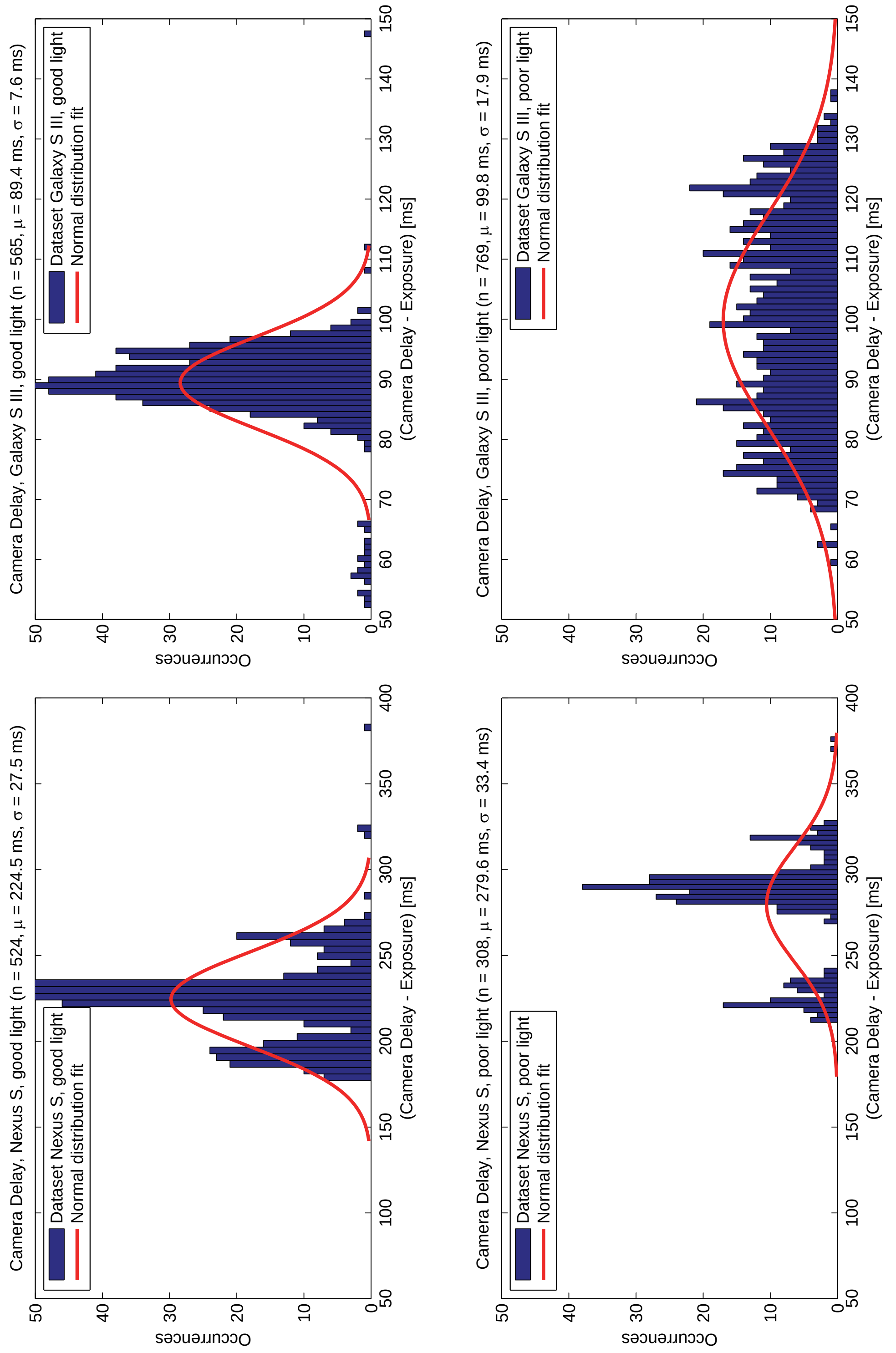
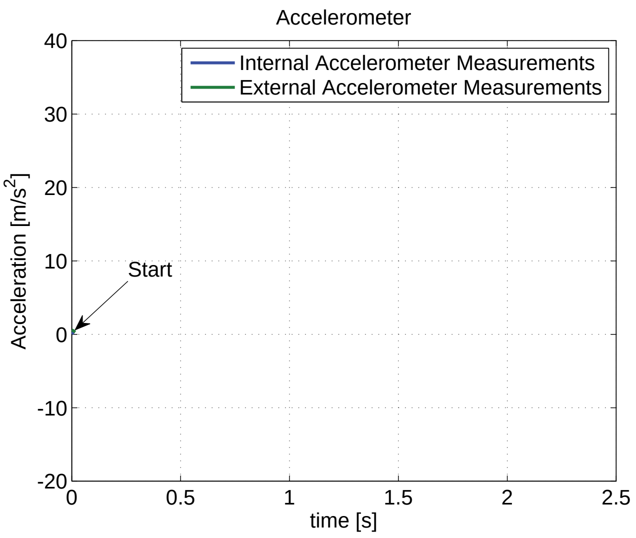


Figure B.2: Camera Delay without Exposure

Appendix C

Motion Study of a Medium Height Throw



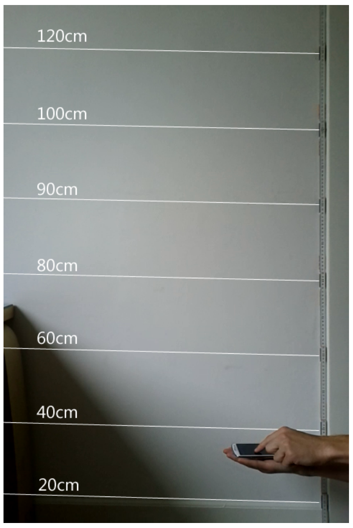
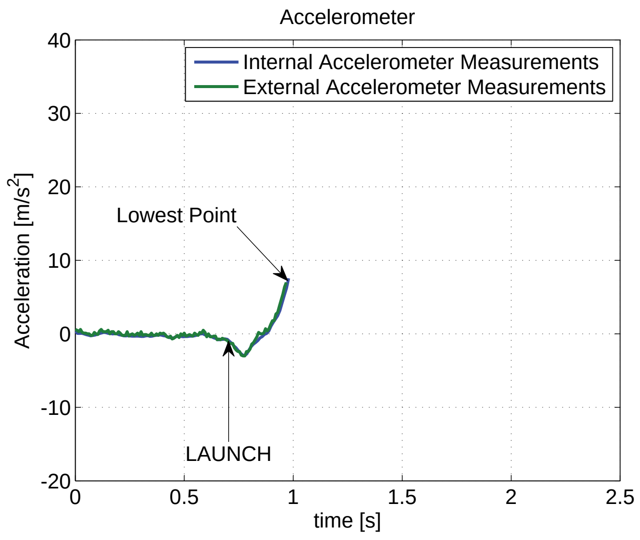


Figure C.1: 1. Start



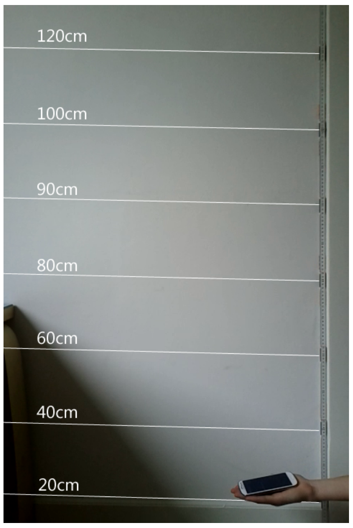
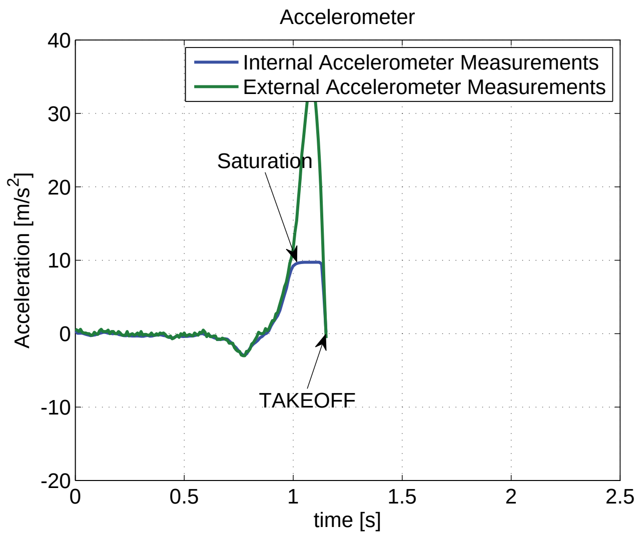


Figure C.2: 2. Lowest Point. The lowest point is reached when the area under the acceleration between LAUNCH and the time of lowest point is zero. This corresponds to velocity being zero.



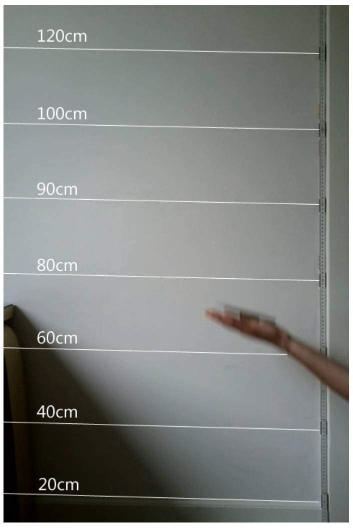
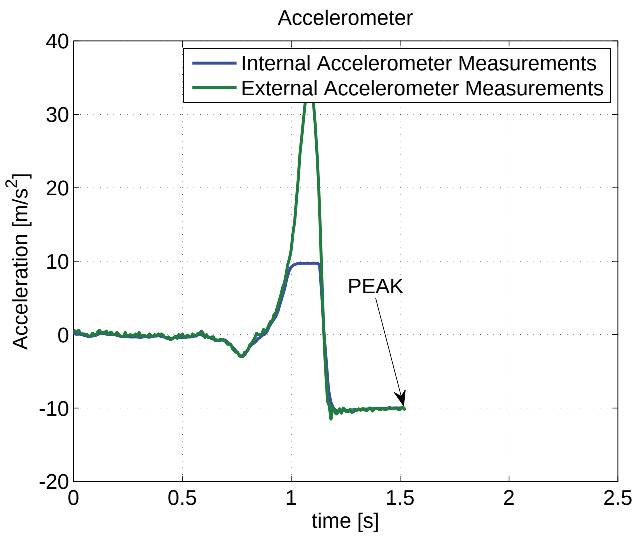


Figure C.3: 3. Takeoff Event. The takeoff event is triggered by a change in the sign of the acceleration and a high enough initial velocity. The saturation in the internal accelerometer causes the plateau at $10\frac{m}{s^2}$.



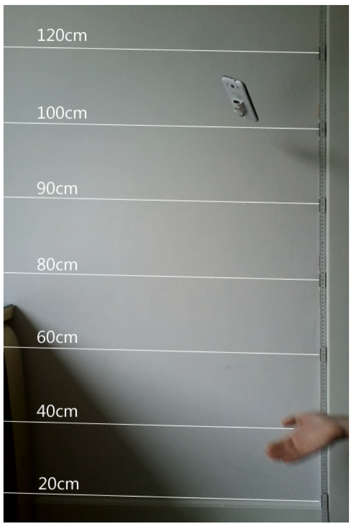


Figure C.4: 4. Peak event. The peak event occurs when the integral over the acceleration from the time of launch until the time where the peak occurs is zero.

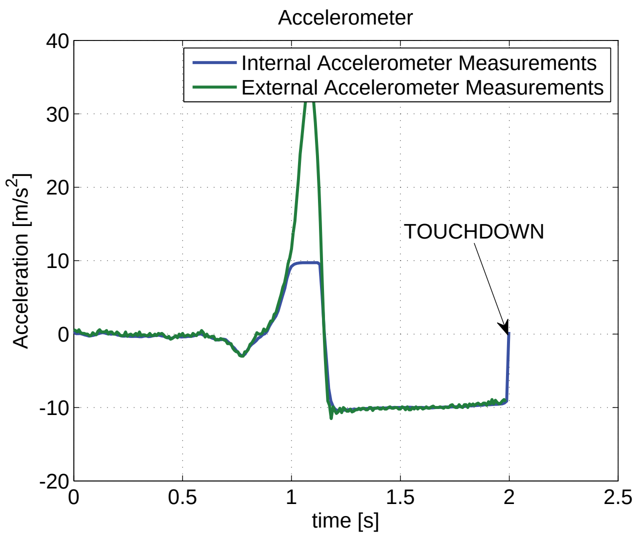


Figure C.5: 5. Touchdown Event. The touchdown event is triggered by a change in the sign of the acceleration.