REAL TIME THREE DIMENSIONAL MOTION TRACKING USING SMARTPHONES

Tobechukwu Gladson Nwanna

A Thesis

Presented to the faculty

of Princeton university

in candidacy for the Degree

of Master of Science in Engineering

Recommended for acceptance

by the Department of Computer Science

Princeton University

Adviser: Szymon Rusinkiewicz

June 2013

Abstract

Software that requires users to explore virtual three dimensional spaces, such as modeling/CAD programs and videogames, is becoming increasingly common. Despite the prevalence of such software, interfaces have remained stagnant and failed to evolve. Many still attempt to map two dimensional inputs to a three dimensional output which often results in user confusion and an increased learning curve. We propose a new type of interface that utilizes natural gestures to provide a more intuitive way to explore 3D spaces. Using smartphones, we have created a system that is able to track the devices' movement in three dimensions, without the need for additional sensors or prior setup. It takes advantage of the various sensors found in modern phones as well as the camera to estimate motion. We detail the motion estimation pipeline, including its strengths and weaknesses.

Acknowledgements

I would like to thank my adviser Szymon Rusinkiewicz for his continuous support and willingness to answer the questions I had. His enthusiasm helped me get through times when progress was coming along slowly. I would also like to thank all the Princeton Computer Science professors and students who helped provide an environment that allowed me to comfortably discuss my ideas. Lastly, I would like to thank my parents who supported me throughout the duration of my time as a Master student at Princeton.

Contents

Abstract iii

Acknowledgements iv

1 Introduction 1

1.1 An Example Scenario in Google SketchUp 1

2 Related Work 3

2.1 The Nintendo Wii Remote 4

2.2 The Playstation Move Controller 4

2.3 The Microsoft Kinect 5

2.4 Possible Improvements 6

3 System Overview & Design 7

3.1 The Choice of Smartphones 7

3.2 Sensor Overview: Accelerometer 8

3.3 Sensor Overview: Gyroscope 8

3.4 Sensor Overview: Magnetometer 8

3.5 Sensor Overview: Camera 9

3.6 Pose Estimation Pipeline 9

4 Implementation 11

4.1 Data Structures 11

4.2 Sensor Calibration 12

4.3 Determining Orientation 12

4.3.1 Calculating a Global Orientation 12

4.3.2 Handling Noisy Measurements 13

4.4 Determining Angular Velocity 14

4.5 Using a Kalman Filter to Improve Estimates 15

4.5.1 Kalman Filter Parameters 16

4.6 Determining Translational Motion 18

4.6.1 Detecting Feature Points 18

4.6.2 Tracking Feature Points 19

4.6.3 Approach #1: Three View "Structure from Motion" 20

4.6.4 Approach #2: Bundle Adjustment.. 22

4.6.5 Estimating the Fundamental and Essential Matrices 23

4.6.6 Essential Matrix Decomposition into R and t 25

4.6.7 Refining the Motion Estimate with Sensor Data 26

4.6.8 Refining the z-axis Estimate with Convex Hulls 28

5 Results... 29

5.1 Performance 29

5.2 Experimental Procedure... 30

5.2 Discussion of x- and y-axis Translation 35

5.3 Discussion of z-axis Translation.. 38

6 Future Work 39

7 Conclusion 40

References... 42

1 Introduction

Human computer interaction is an area of computer science that explores alternative methods of interacting with digital systems. A subset of HCI is natural user interfaces which take advantage of gestures and physical motions people are familiar with to provide more intuitive interactions. Despite advances made in the field, when it comes to interacting with three dimensional spaces, most interfaces suffer from a fundamental problem. They attempt to map two dimension inputs, from devices such as keyboards, mice and touch screens, to three dimensional outputs. This leads to a level of interface abstraction that can confuse users and detract from the experience.

1.1 An Example Scenario in Google SketchUp

To better illustrate the problem we turn to Google SketchUp as an example. SketchUp is an easy to use modeling program, designed to accommodate both novices and users experienced in modeling software. For this example we will primarily be focusing on the aspects of the program that pertain to how the user explores the space. Looking at **Figure 1** we see the area of the toolbox that contains the camera controls. Note there are three primary ways to manipulate the camera.

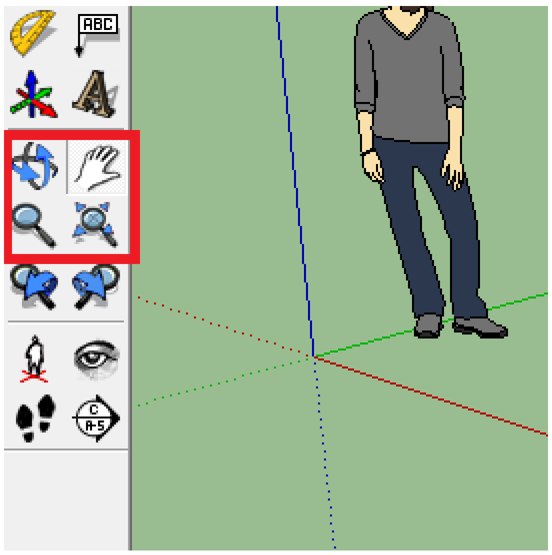


Figure 1: Google SketchUp camera control panel

The first method is the *orbit* function which allows the user to rotate the camera around the central axis. To the right is the *pan* function which allows the user to move the camera along the current viewing plane. We will refer to these motions as “rotate” and “translate”, respectively. The last two functions are *zoom* and *zoom extents*. The first function allows the user to perform a granular zoom in or out while the second function zooms in at fixed magnification intervals. All of these are controlled by various mouse gestures. Orbit and pan are controlled by clicking and dragging in the desired direction of movement. Zoom is controlled by clicking and dragging up to zoom in and down to zoom out. Zoom extends zooms in every time the toolbox button is clicked.

While these gestures work they illustrate the problem with taking two dimensional inputs (mouse gestures) and mapping them to three dimensional outputs. In this case, they require the user to switch modes using the various toolbox buttons and sometimes perform gestures they are not very intuitive (such as dragging to zoom). Ideally, a user should be able to perform all of these without having to switch modes and using an input device that has the same dimensionality as the environment space. We hope to solve this problem by replicating these behaviors with more natural gestures.

2 Related Work

In recent years there have been a number of devices, primarily from the videogame industry, that have attempted to address this issue. In the next few sections we will take a look at three of these devices, briefly explaining the technology behind each and its relative strengths and weaknesses.

Figure 2: Nintendo Wii Remote[1]

Figure 3: Playstation Move Controller[2]

Figure 4: Microsoft Kinect[3]

[1] Image taken from http://www.baixakijogos.com.br/assets/pictures/000/002/616/content_pic.png
[2] Image taken from http://www.compumundo.com.ar/fotos_productos/nuevas/44766/grande/44766.jpg
[3] Image taken from http://cdn2.sbnation.com/products/large/1792/kinect.jpg?1345120439

2.1 The Nintendo Wii Remote

The first device is the Nintendo Wii remote pictured in **Figure 2**. Its primary interface is the motion sensing remote controller and a sensor bar. The remote comes equipped with a 3-axis accelerometer and has an infrared optical sensor at its head. The accelerometer is able to measure forces along three axes, allowing the remote to detect its orientation in three dimensions as well as sense quick motions made in any direction.

The sensor bar is a rigid panel of 10 infrared LEDs clustered into groups of 5 at each end of the bar. The optical sensor on the remote is able to capture the position of each bulb and through triangulation, between a pair of bulbs and remote, is able to determine where the remote is pointed relative to the sensor bar's location. In addition, using the captured size of the LEDs on the sensor (in pixels), the remote is able to roughly determine its distance from the bar [1] [2].

2.2 The Playstation Move Controller

The Playstation Move Controller, pictured in **Figure 3**, similarly comes equipped with a 3-axis accelerometer, but also includes a 3-axis gyroscope and 3-axis magnetometer. At the head of the controller is a sphere containing RGB LEDs. The orientation of the device is sensed in the same way as the Wii remote except the magnetometer now allows it to sense its global heading as it functions as a compass. The gyroscope allows it to more accurately disambiguate rotational motions as it tracks angular velocity.

A webcam, known as the Playstation Eye, comes with the Move and allows tracking of translational movement by tracking the position of the LED sphere. Since the system knows the physical size of the sphere it can determine its distance from the camera by calculating its size in pixels. [3] [4].

2.3 The Microsoft Kinect

The last device is the Microsoft Kinect, pictured in **Figure 4**. Unlike the previous two devices which facilitated motion tracking through the use of handheld controllers, the Kinect is completely hands free. It utilizes a depth camera composed of an infrared projector and infrared sensor.

The projector projects a pattern of infrared lights onto the scene which the sensor captures. The different relative distances of objects to the camera causes the pattern to look deformed when captured by the sensor. Using this deformation the depth of the scene can be estimated. The depth map produced by this process can then be segmented to extract any visible bodies and track their motions. While this allows more robust tracking of translational motion, tracking of rotational motion, especially of smaller limbs like the hands, suffers [5].

2.4 Possible Improvements

Each of these devices achieves the goal of allowing a user to interact with a three dimensional space using a three dimensional input. Instead of trying to approximate an equivalent gesture or requiring the user to switch modes each does a nearly 1:1 mapping of physical motion to virtual motion. However, as general purpose input devices, each of these suffer from various weaknesses.

The Wii remote is unable to sense translational motion, which severely limits its range of uses. In addition, to take full advantage of its functionality it requires the prior setup of the sensor bar. The Playstation Move controller, while being able to sense translation, requires the prior setup of the Playstation Eye for that functionality. Although it can track motion without the Eye its sensors very quickly drift without the Eye providing a means of recalibration. The Kinect is unable to sense subtler gestures and also requires prior setup. In addition, all three have specific range requirements that limit what types of environments they can be setup in.

Ideally, we want a system that satisfies these 3 goals:

1. **A general purpose system** able to track the same types of motions as these devices. In the context of the SketchUp example we should be able to replicate the three types of camera manipulation (orbit, pan and zoom).
2. **A system that does not require the user to install another sensor to use it**. It should be able to act completely independently.

3. **A system that operates at a reasonable speed**. As an interface, the tracking has to be handled in real time and at sub-second speeds.

With these goals in mind, we propose our idea for a new type of motion tracker.

3 System Overview & Design

3.1 The Choice of Smartphones

Our proposed solution is to use smartphones as general purpose three dimensional motion tracking devices. Smartphones are ideal input devices for a variety of reasons. A 2012 survey showed that 49.7% of mobile phone users in the US owned a smartphone with 48% of them owning Android devices [6]. A market penetration that high signifies a large number of users are familiar with using these devices, which lowers the learning curve with using this new type of input device.

Another major reason is that most smartphones now contain a large number of sensors. Although these are mainly used for phone related functions, the Android API allows developers to harness them for applications. The vast majority of Android phones have a 3-axis accelerometer, 3-axis gyroscope, 3-axis magnetometer and a camera capable of recording video. For these reasons, we are developing this system on the Samsung Galaxy S2, running Android 4.0. To better understand how these sensors will be incorporated into the system, we briefly explain the capabilities of each sensor and what their primary purpose will be.

3.2 Sensor Overview: Accelerometer

A 3-axis accelerometer is a sensor capable of measuring the force applied to it in three dimensions. In the absence of any user applied force it measures gravity and can determine its orientation based on this. However, when there is a user generated force applied to the accelerometer, distinguishing between that force and gravity becomes difficult which is why the accelerometer is poor device for measuring rotational motion. In addition, accelerometers are fairly noisy and their values are prone to fluctuate. [7].

3.3 Sensor Overview: Gyroscope

A 3-axis gyroscope, sometimes referred to as an “angular rate sensor”, is a sensor capable of sensing rotational motion along three dimensions. Specifically, it is able to detect its instantaneous angular velocity. Although gyroscopes are supposed to measure an angular velocity of zero when not undergoing any rotation, after prolonged use they will sometimes measure non-zero values in a phenomenon known as drift. As a result, they require regular calibration [7].

3.4 Sensor Overview: Magnetometer

A 3-axis magnetometer is a sensor capable of measuring magnetic field strength and direction in three dimensions. In the absence of any strong magnetic fields nearby it can be used as a compass, able to detect the magnetic field of the Earth. This means it is able to determine its absolute heading. Like the other two sensors, magnetometers are fairly noisy [7] .

3.5 Sensor Overview: Camera

The last sensor is a complementary metal–oxide–semiconductor (CMOS) sensor, more commonly referred to as a camera. The Galaxy S2 is equipped with a camera capable of recording 1080p (1920x1080) video at 30 frames per second, although the framerate varies depending on the lighting conditions. It has a fixed focal length with autofocus [8].

3.6 Pose Estimation Pipeline

Figure 5 shows an overview of our pose estimation pipeline. Pose refers to the *position* and *orientation* of a device (the smartphone in this context). The rectangular boxes in orange show the final outputs—orientation, angular velocity and relative position. We use an idea called *sensor fusion* which states that by combining multiple sensors in an intelligent manner we can produce more accurate results than any sensor could have computed individually.

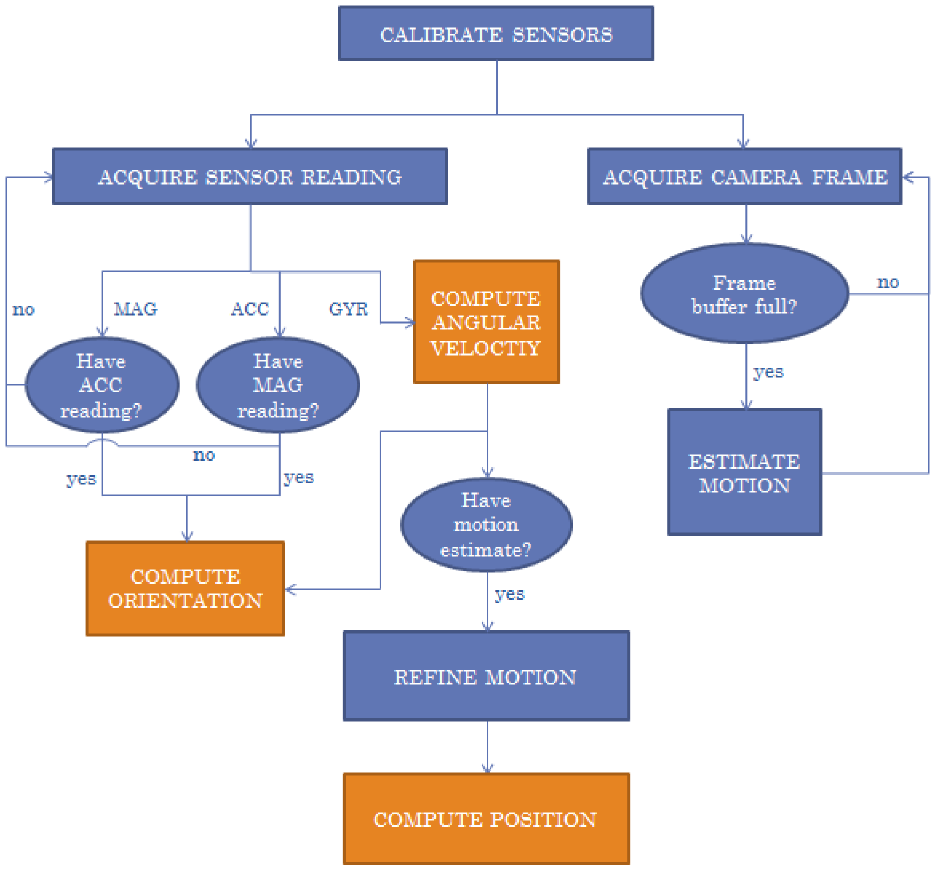


Figure 5: Pose Estimation pipeline

Take note that the pipeline is divided into two sections which we will refer to as *sensor side* (on the left) and *camera side* (on the right). This is because the motion sensors all run on one thread while the camera runs on another. As the sensors poll faster than the camera, even at its highest possible frame rate, the application is driven by the sensor side.

The general algorithm is to:

1. Compute the orientation and angular velocity using the motion sensors
2. Compute an estimate of the translation using the camera

3. Refine the translation estimate using motion sensor measurements
4. Determine the current position of the device relative to when tracking was started

The following sections detail how the pipeline is implemented as well as describe a few small connections not represented on the flow chart.

4 Implementation

4.1 Data Structures

The pipeline is split into two tracks with each side being event driven by the polling of new sensor data. To "synchronize" both sides we use a structure called *SharedData* that encapsulates all the relevant system constants and sensor readings. It uses object level locking to make sure specific buffer pools are accurate across threads and minimize the amount of waiting each thread has to do to access relevant data.

Whenever new sensor data comes in, it is placed into the appropriate buffer. Each sensor has its own special buffer that keeps track of the last 20 values measured along with a timestamp for each value. The specific size of the buffers was simply a number that we decided provided an ample enough history, as it roughly corresponds to the last 2 seconds of measurements. In addition, each sensor has a flag it sets whenever new data is captured. This allows other parts of the system to know when a sensor has new data and which sensor's buffer was most recently updated.

4.2 Sensor Calibration

At the start of every run of the application the user is asked to leave the device stationary for a period of time to allow for calibration. During this phase the raw values of the different sensors are collected and a number of statistics are computed, included the *bias* (average values when stationary), *standard deviation*, *variance*, and *min/max* values. A number of these values are used in calculations meant to help process the data. These are detailed in later sections.

4.3 Determining Orientation

4.3.1 Calculating a Global Orientation

Orientation can be determined solely using the accelerometer values. This is under the assumption that the device is not being affected by any other large forces aside from gravity. This value is only a relative orientation as the accelerometers are not able to discern which direction the device is actually facing. To compute a global orientation, we need the device's heading, which we can obtain using the magnetometer.

Computing the device's heading also serves the purpose of allowing the system to define a "default" orientation that can be maintained through the duration of the tracking process. An example of this idea is illustrated in **Figure 6.**

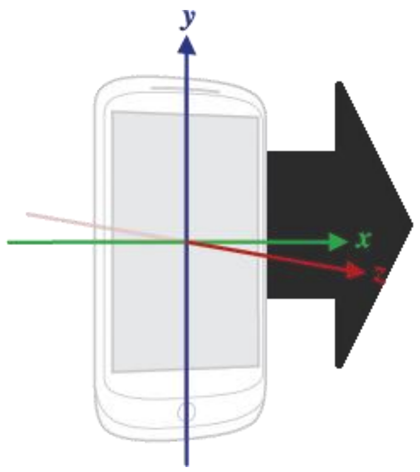


Figure 6a: The device's initial "default" orientation

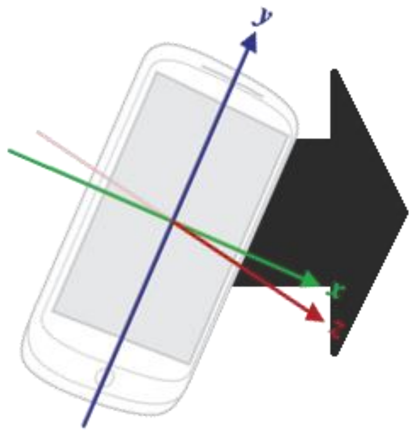


Figure 6b: The device rotated to a new orientation

In **6a**, the movement represented by the black arrow would be interpreted as a pure translation along the x-axis. In **6b** the same movement would be mistakenly interpreted as a translation along both the x- and y-axis. To keep all movements using the default orientation's coordinate frame we simply compute the rotation matrix representing the change in orientation from **6a** to **6b**. We then multiply the translation by that rotation to transform it to the "default" coordinate frame and get the intended translation.

4.3.2 Handling Noisy Measurements

Standard Deviation of Device Orientation When Stationary			
Axis	X	Y	Z
Raw	0.003487	0.002204	0.016163
Filtered	0.003132	0.001676	0.143751
Difference	-0.00035	-0.00053	+0.127588

Table 1: Shows the effects of filtering on noise. The standard deviation of orientation calculations is generally reduced

Although the accelerometer and magnetometer can calculate the orientation with a reasonable accuracy, the sensor measurements are fairly noisy. Determining if the global orientation is correct is difficult so instead we look at the standard deviation of calculated orientation values when the device is stationary in **Table 1**. With perfect sensors these should all be zero. Although these values are not very large they can be improved with filtering. We will refer back to this chart after explaining the filtering process.

4.4 Determining Angular Velocity

Angular velocity can be computed in two ways. The first is to compute the difference between subsequent orientations and divide by the delta between the timestamps. This is a poor measure as the accelerometer values used in that computation are less accurate as the device is moving. A more accurate method is to use the gyroscope. The gyroscope returns in radians/second which is exactly what we want, however, it suffers from drift and noise.

Drift is partially dealt with by subtracting out the initial bias. This solution only works for a short period of time and ideally the bias should be recalculated when possible. Noise is another issue that needs to be addressed. Both of these issues are dealt with by using a Kalman Filter. **Chart 1** displays the average readings from the gyroscope when stationary as well as the current bias. The chart shows that by subtracting out the bias we get more accurate values.

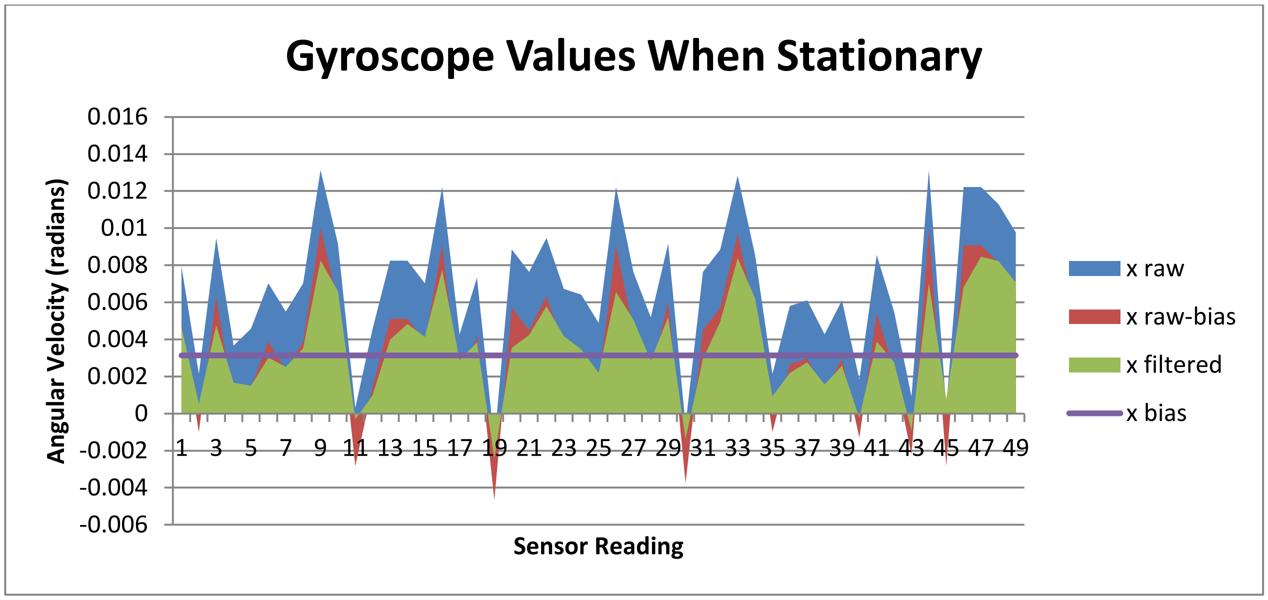


Chart 1: Gyroscope values when stationary. The closer the angular velocity is to 0 the better.

4.5 Using a Kalman Filter to Improve Estimates

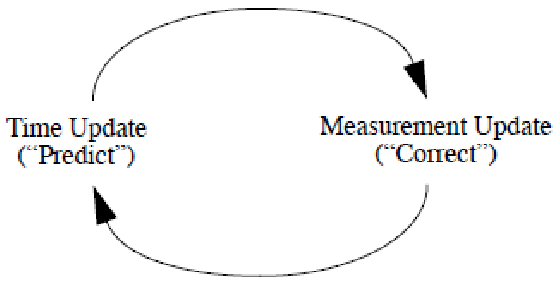


Figure 7: The Kalman Filter algorithm cycle[4]

A kalman filter is a recursive algorithm used to smooth data with uncertainty. It works using a two-stage process. During the first stage, the state variables of the system, represented by an

[4] Figure 1-1 from [17]

internal model, are “predicted” for the next time step using a model of the transition they are expected to undergo. During the next stage, this transition is physically measured and both the prediction and measurement are weighted based on the level of uncertainty in each. The state values are then updated based on the ratio of these weights. **Figure 7** shows a simplified overview of what was just described.

4.5.1 Kalman Filter Parameters

The state of the filter contains the values that are being estimated and tracked by the system. The transition model describes how these values are expected to change between subsequent time steps. And finally, the measurement model describes which state values are being updated by the current measurement. The basic kalman filter assumes a linear model for the system, which allows us to use a linear algebra package to quickly compute values. We use an open source Java library called Efficient Java Matrix Library (EJML).

$$x = \begin{pmatrix} \omega \\ \theta \end{pmatrix} \qquad F = \begin{pmatrix} 1 & 0 \\ \Delta & 1 \end{pmatrix} \qquad H_a = \begin{pmatrix} I \\ 0 \end{pmatrix} \text{ or } H_b = \begin{pmatrix} 0 \\ I \end{pmatrix}$$

Table 2: The state model(x), transition model (F) and measurement model (H)

Each state variable has three components (x, y, z) so x is actually 6×1, F is 6×6 and H is 6×6. The transition model assumes the gyroscope values are constant ($\omega_t = \omega_{t-1}$) and that the orientation is updated by $\theta_t = \theta_{t-1} + \omega_t * \Delta$ with Δ (delta) being the time between current and previous sensor measurement.

The kalman filter allows the system to be updated with partial measurements so H is either H_a or H_b depending on what sensor currently has a reading. Because orientation can only be computed using both the accelerometer and magnetometer, H_b is only used when both have new values as opposed to H_a which is selected whenever we have a gyroscope reading. While ideally we should wait until all the sensors have values, the algorithm is robust enough to handle partial measurements, and, in the interest of speed, it makes the most sense.

Looking at a comparison between raw values and filtered values for orientation in **Chart 2,** we see that the values are effectively smoothed by the filter, resulting in less variance in calculations. Going back to **Table 1** we see this in the comparison of standard deviate values between the raw and filtered computations. In general the kalman filtered values have a smaller standard deviation as noise is reduced. Furthermore, looking at **Chart 1** we see that noise is also reduced in the gyroscope values resulting in more accurate measurements of the angular velocity. As an added effect, the effects of drift are reduced as the system keeps the gyroscope values constrained by the transition model which assumes they stay constant.

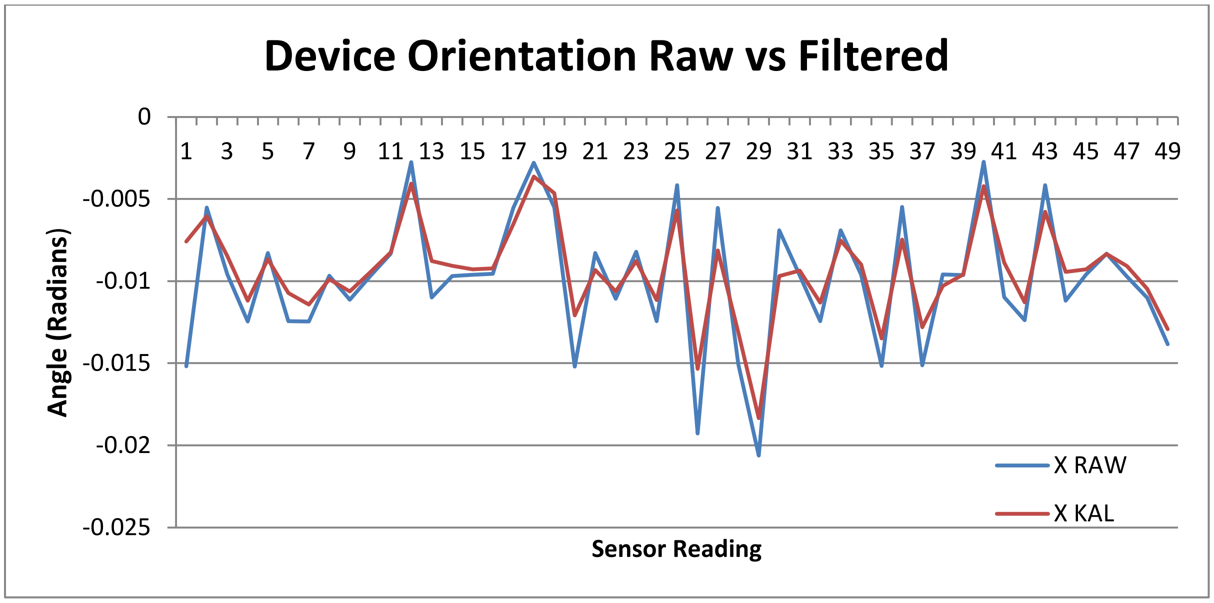


Chart 2: An example of the smoothing that occurs when the orientation is processed by the Kalman Filter

4.6 Determining Translational Motion

Translational motion is primarily computed using the camera side of the pipeline. The approach is to track points of interest and estimate the movement of the camera by observing the relative movement of these points. Since a number of computer vision algorithms are used we take advantage of the popular open source computer vision library OpenCV. The follow sections delve into the process of acquiring motion and briefly explain the various algorithms used and extended from OpenCV.

4.6.1 Detecting Feature Points

The process of estimating camera motion from image data is called visual odometry. It starts with the identification and tracking of interesting feature points across multiple frames. A *feature point* is simply a point of interest in an image that can be uniquely identified. There are various feature point detectors and descriptors (quantitative descriptions of features) that can help identify them even as an image frame undergoes various changes from camera movement.

In the interest of speed we chose a relatively simple detector, developed by Harris and Stephens, that detects corners [9]. It works by taking an image converted to grayscale, I, then computing the gradients of slightly shifted versions, I_x and I_y. The weighted sum of squared differences between equivalent patches is then computed into a 2x2 matrix. The presence of a corner in that patch can be detected by observing the first and last value of this matrix. If one of the values is large an edge is present and if both are large a corner is present.

4.6.2 Tracking Feature Points

After a reasonable number of features are detected they are then tracked across frames. We use a parallelized implementation of the Lucas-Kanade feature tracker using pyramids [10]. This algorithm works by searching across small windows in a downsampled (low resolution) version of the frame then iteratively projecting those searches up the "pyramid", increasing the resolution, until the flow of features is found at the original resolution. While these algorithms are robust, like most vision algorithms they are highly dependent on the image quality.

If the lighting conditions are poor strong features are hard to identify and track. The problem is compounded by Android's low light compensation which dynamically adjusts the framerate depending on the lighting conditions. When observing darker scenes the framerate can sometimes decrease to 15 fps which, depending on the motion, can result in significant blurring. While this cannot be fully addressed, we filter out features that could not be properly tracked or were occluded in later frames. We use an array of status bits OpenCV generates when optical flow is attempted that indicates whether a particular feature was successfully tracked or not.

Once we had the ability to detect and track features, we tried a variety of algorithms for estimating motion. We found in our attempts that given the 3rd goal we set for the system, specifically for it to be fast enough to operate in real time, a number of algorithms were not viable. We briefly touch on some of the algorithms that we tried and why they were unsuitable for the task.

4.6.3 Approach #1: Three View "Structure from Motion"

Structure from Motion describes a class of algorithms that attempts to recover the 3D structure and motion of a set of observed feature points. In the context of this application, we intended to use the recovered motion. We implemented a well-known algorithm called Tomasi-Kanade Factorization as a first attempt at assessing the accuracy of estimating motion using only the camera [11].

A key assumption of the algorithm that was initially ignored was that the image data was captured using an *orthographic camera model*. An orthographic camera assumes feature points are projected parallel from their physical positions onto the image plane. This means that scale and depth (the distance from the object to the image plane) are unimportant. Modern cameras actually use what is known as a *pinhole/perspective camera model* where depth has an effect on the scaling of observed objects. We chose to ignore this assumption because we assumed that features observed at a reasonable distance could be approximated as orthographic projections. **Figure 8** shows the difference in the two camera models.

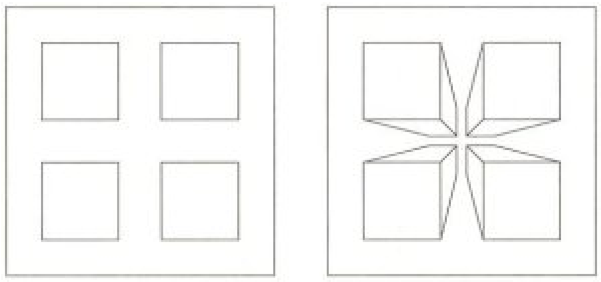


Figure 8: The difference between an orthographic and pinhole camera when observing 4 rectangular prisms.[5]

We performed a series of simple tests where we moved the device along an axis and observed the estimated motion. We found that the estimated rotation and translation were not close to what was expected. The rotation matrix generated did not represent a small rotation (we say small because in the presence of noise expecting absolutely no rotation is unreasonable) and the translation vector did not report a significant translation along the axis we moved the device. We chose not to include the results of these tests, but concluded from running them

[5] Image taken from http://www.seos-project.eu/modules/3d-models/images/ortho_vs_persp.jpg

multiple times that the orthographic assumption was too crucial to ignore or hope to approximate and that this route would not be ideal.

4.6.4 Approach #2: Bundle Adjustment

Bundle adjustment is class of algorithms that simultaneously refines the estimate of the 3D geometry of a scene as well as the motion [12]. What makes these algorithms particularly effective is that they operate globally over all features to minimize an objective function describing the error in estimations between pairs of frames. In our system true bundle adjustment is not possible as we only have frame data up to the current point in time. However, there have been bundle adjustment algorithms that attempt to perform this process incrementally.

The algorithm we explored and partially implemented was by Zhang and Shan. It attempted incremental motion estimation through local bundle adjustment [13]. The algorithm is initialized by computing an estimate for the second camera's motion and the current 3D structure and then they are refined as new frame data is obtained. Its minimization function attempted to minimize the distance between a tracked feature point's location in a frame and its projected location (based on its estimated 3D position and camera pose). It is called a "local" bundle adjustment because it uses the set of frames, $(frame_{t-2}, frame_{t-1}, frame_t)$. Unfortunately, we found that even with a relatively small number of feature points the minimization step is very computationally expensive, causing the algorithm to operate on the order of seconds. For a real time system this is far too long.

4.6.5 Estimating the Fundamental and Essential Matrices

We ultimately settled on a two view approach that involved computing the essential matrix, a matrix describing the relationship between two views from a calibrated camera, and decomposing it into the rotational and translational components. The next few sections define the key terms and concepts used to compute the essential matrix and how it is decomposed.

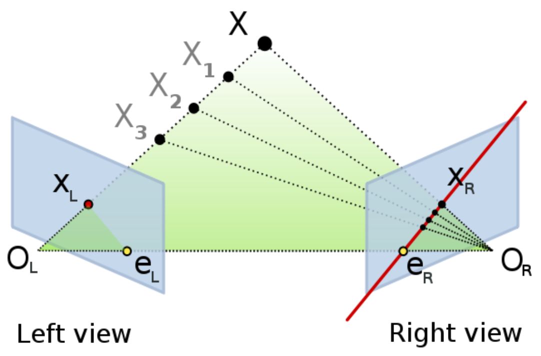


Figure 9: The relationship represented by the fundamental matrix. The epipolar constraint[6]

A fundamental matrix (F) is a 3x3 matrix that describes the relationship between observed points in different views. In **Figure 9** we see a visual of this. In the context of our problem, left view and right view refer to the position of a camera before and after undergoing an arbitrary motion. The observed 3D point X is seen by both views, projected onto position

[6] Image taken from http://en.wikipedia.org/wiki/File:Epipolar_geometry.svg

X_L in the first view and X_R in the second. If we draw a line from the optical centers of each view through X and project this line onto the opposite view we get what is known as an *epipolar line* (the red line). This line defines a set of constraints on where the observed point can appear in the second view and where the observed point could potentially exist. Mathematically, this is captured by the equation $xFx' = 0$, where x and x' are matching points in the first and second view [14].

OpenCV computes this using RAndom SAmple Consensus (RANSAC) which tries to fit models to random subsets of the data until it gets an accurate model above the specified confidence level (we used .99). Because this process is highly dependent on the order of points we employ a scheme where we randomize the feature points before performing the computation to generate a set of possible fundamental matrices.

The essential matrix, like the fundamental matrix, describes the relationship between sets of matched feature points in different views. It differs in that it uses normalized feature points, points that have been adjusted based on the parameters of the camera used to capture them. These parameters are the *focal length*, and image *centers*. They are normally referred to as the *camera intrinsics* and arranged in the matrix $K = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}$. If we have a fundamental matrix and know the camera intrinsics we can compute the essential matrix with $E = K^T F K$.

4.6.6 Essential Matrix Decomposition into R and t

According to Theorem 2 in [15] we can use singular value decomposition (SVD) to decompose the essential matrix into a set of possible rotation matrices and translation vectors. We say "possible" because a given E can describe two rotations and two translations for a total of four possible motion combinations. The correct rotation and translation pair can be chosen fairly easily using a chirality check also detailed in [15]. Decomposed values for the translation are normalized by the magnitude of the translation.

In the previous section we briefly mentioned that we compute multiple fundamental matrices and choose the best one out of the set. This is because the order of points can effect what subsets are generated and tested by the RANSAC algorithm. In detail, this process is as follows:

1. Compute the average displacement of pixels, d_x and d_y, between the two frames
2. Generate 3 possible Fs.
3. Compute their associated E and decompose to get a potential (R, t) pair
4. Eliminate the E whose ratio t_x/t_y differs most from the other two
5. Select the E whose t has a ratio t_x/t_y closest to d_x/d_y

Using this process we are able to get a somewhat better estimate for R and t and choose the E that is the most likely to represent the actual movement of the camera. We refine this estimate using sensor data.

4.6.7 Refining the Motion Estimate with Sensor Data

The essential matrix can be decomposed into its rotational and translational components because of the relationship $E = R[t]_x$ with $[t]_x$ being the skew-symmetric matrix (represents the cross product of t and any other vector) of the translation vector t [16]. Using this, instead of decomposing to find these values we can instead substitute R into the equation and solve for $[t]_x$.

Whenever we compute an essential matrix and save it into its respective buffer we also include the two frames involved. Using these frames and their timestamps, we can find the period when the motion was performed and filter the gyroscope buffer to find the readings recorded during that period. Using those readings we can compute an R representing the total *measured* rotation. From there we use a linear solver to solve for $[t]_x$ and get the components t_x, t_y, and t_z. We find that in many cases this produces better results than using the estimate from decomposition.

We tested this by performing a pure translational motion along the x-axis and computing the average ratio ${t_x}/{t_y}$ of all the translations calculated for the motion. A higher value was desired as it meant the system was better able to estimate that motion was only along the x-axis. While the decomposed value does a good job at estimating the translation when we computed the average ratio, that had a ratio of 65.69543 while the solved for translation had a ratio of 66.61833. While this might seem minor, considering the frequency translations are computed (roughly 10 times a second) any small errors quickly accumulate and any small increases in accuracy can be significant across long periods of time.

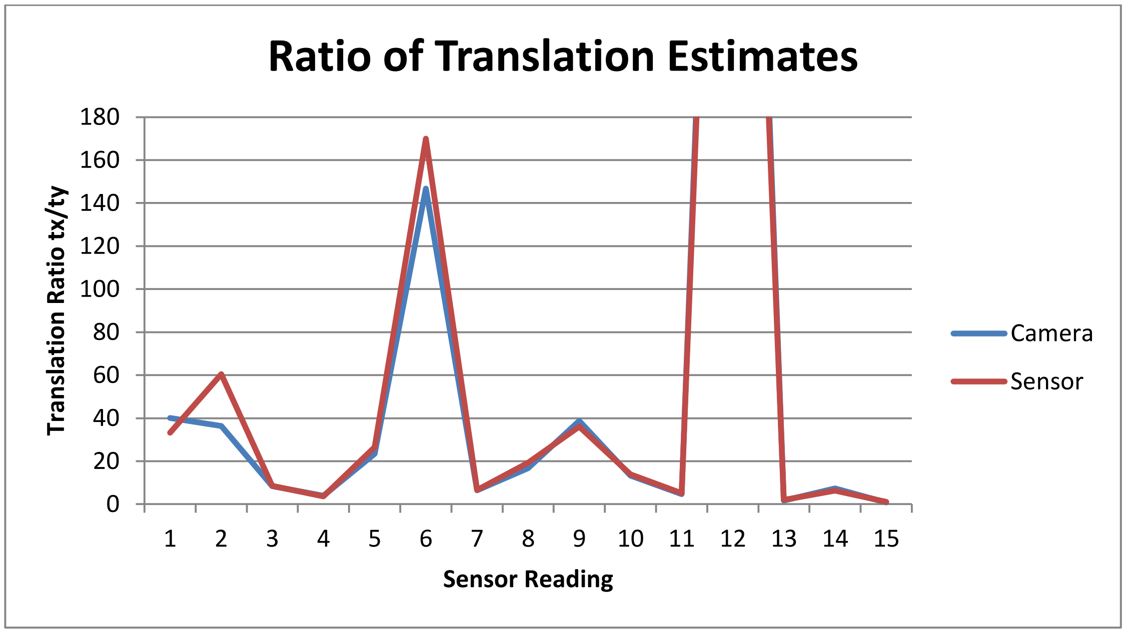


Chart 3: Ratio of t_x/t_y during a translational motion along the x-axis. Higher ratios are better.

One minor issue we encounter is that the recovered t using this method is arbitrarily scaled, sometimes even with a negative scalar. To address this we simple use the signs of the estimated t, which explains why computing it is still necessary even though we only use E. Because the translation vector is arbitrarily scaled, we normalize the values by the magnitude of the translation.

4.6.8 Refining the z-axis Estimate with Convex Hulls

An issue we encountered was that t_z is often not a useful measure of translation in that direction. Referring back to **Figure 6a** we see that movements perpendicular to the plane of the phone's surface are along the z-axis. Every time a translation vector is estimated, whether by decomposition or by solving, the z value drastically differs. We speculate that this is caused by scale factor differences when estimating the 3D depth of feature points between computations of F. Although this does not seem to adversely affect t_x and t_y, it seems to be quite significant with t_z. As a result we do not use that value in any of our calculations. Even when selecting the appropriate t estimate, we use a ratio involving just t_x and t_y.

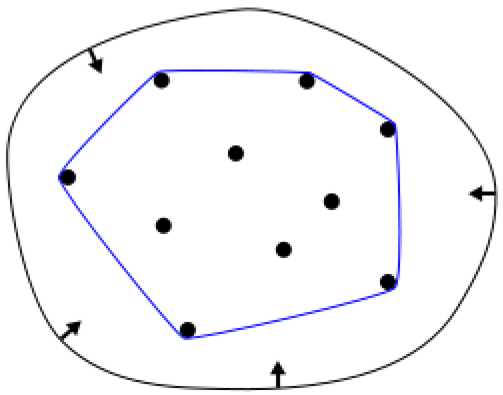


Figure 10: A visual representation of the convex hull of a set of points[7]

We instead use an alternate method to estimate t_z involving convex hulls. Looking at **Figure 10** we see that a *convex hull* is the smallest convex shape that encapsulates all

[7] Image taken from http://en.wikipedia.org/wiki/File:ConvexHull.svg

observed feature points. When the camera is undergoing a pure translation in the z direction we observe that matched feature points either converge or diverge from the centroid of all the points. We take advantage of this behavior by computing the hull area of matched features in both frames and finding $ratio_{hull} = {}^{area_{second}}/_{area_{first}}$. We can then see that if,

$ratio_{hull} \approx 1$: no movement in the z-direction

$ratio_{hull} > 1$: zooming in

$ratio_{hull} < 1$: zooming out

To covert this value to a consistent "unit" we use $t_z = ratio_{hull}$ if the ratio is above 1 and $t_z = -{}^{1}/_{ratio_{hull}}$ otherwise. This way, complementary ratios produce opposite but equal translations (i.e. a 2x zoom and a .5x zoom are the equal in magnitude and opposite in sign). With that computation we have successfully computed the full translation vector t.

5 Results

5.1 Performance

Our system was designed with smartphones in mind so it is important that it is computationally inexpensive. We took a few statistics by observing the numbers Android Task Manager reported during a run of the application. As we see in **Chart 4**, the small buffers keep the RAM usage fairly low (compare this to an average of 45.3 MB for the browser with 4 tabs open) and CPU usage, while variable, is usually under 10%.

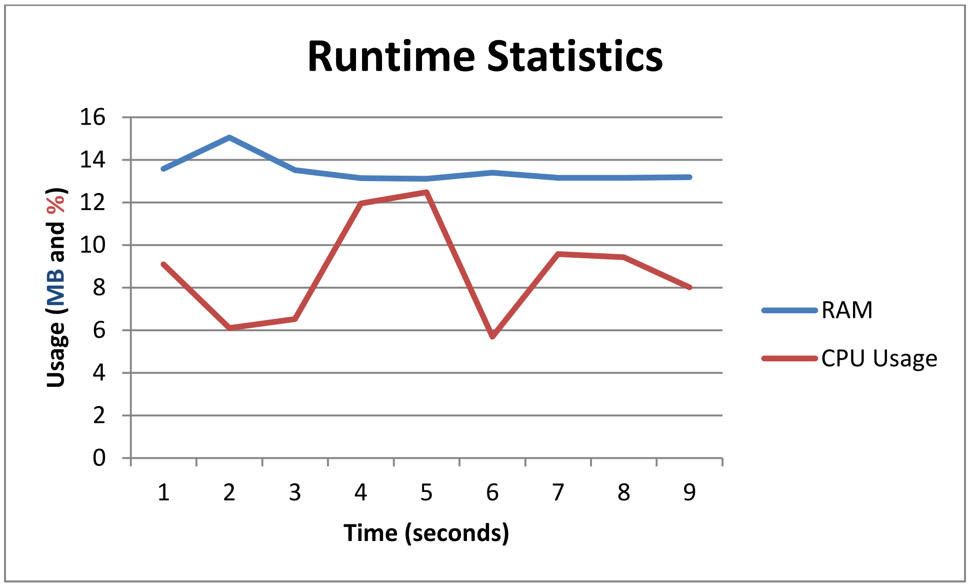


Chart 4: Statistics recorded during a run of the system

5.2 Testing Procedure

For this interface to be of any viable use it needs to be robust to different sets of environments. As feature detection and tracking is a crucial step in the estimation of motion, we conducted our experiments by testing the exact same motion with different sets of features. We ran tests in 5 types of environments, testing the basic motions (pure translation on the x-axis, y-axis and z-axis). The set of tests, described primarily by the "kind" of features the camera was primarily exposed to, were as follows:

1. **Roughly planar features:** Features roughly at the same physical depth/distance

2. **Random & busy:** Areas with lots of potential features at random depths

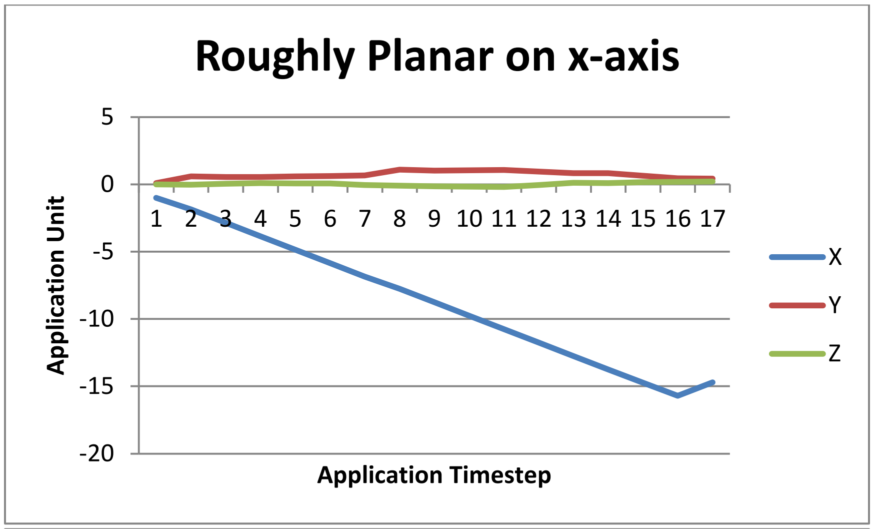
3. **Random & sparse:** Areas with only few potential features

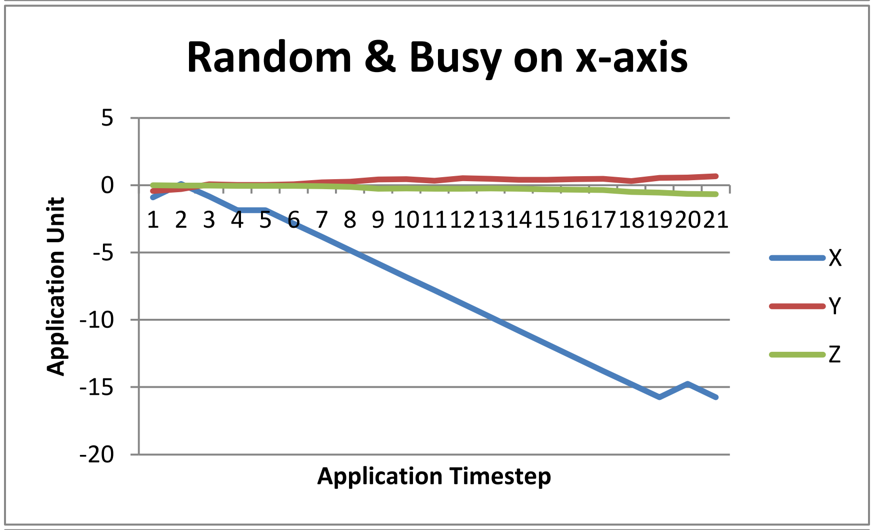
4. **Random & far**: Areas where features were at far depths (> 4m)

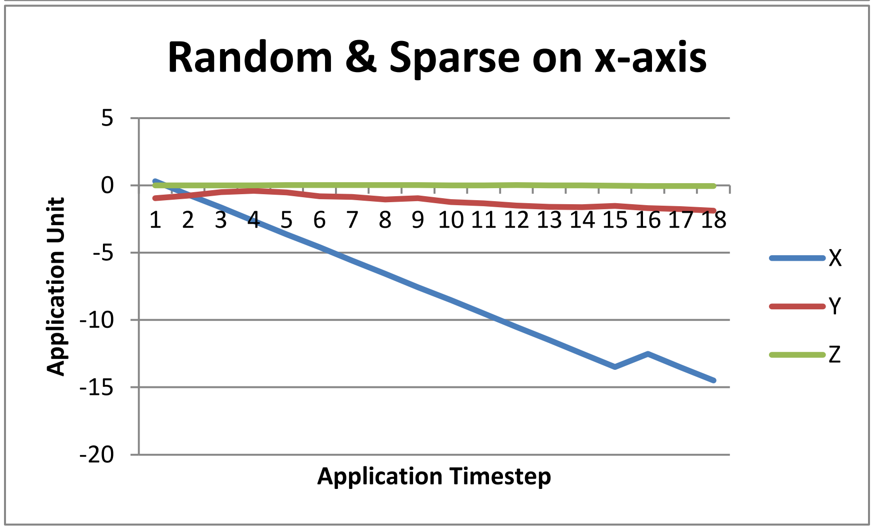
5. **Completely random:** Random features at random depths

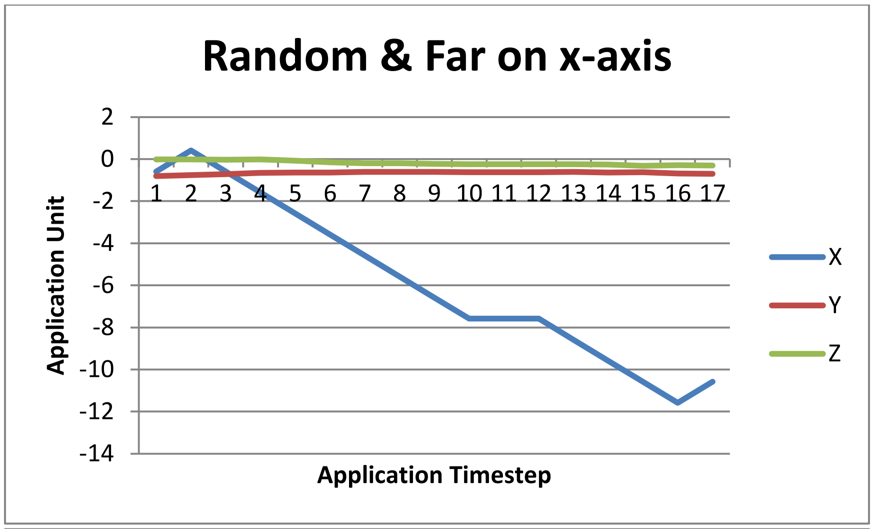
For each test the device was moved a fixed distance for roughly the same amount of time to approximate a constant velocity. While the x and y tests involved movement in a single direction the z test involved moving forward the fixed distance then back that same distance.

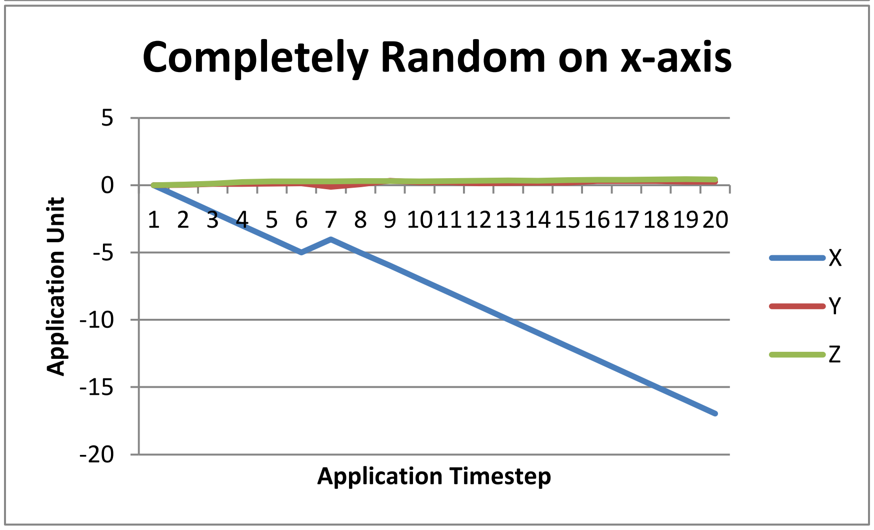
The graph axes are labeled "Application Unit" and "Application Timestep". The first label is because values are normalized so they don't use physical units of measure. The second label refers to the fact that motion estimates are computed at a roughly fixed interval, although it changes slightly based on the current load on the phone (determined by what other applications running at the time). Although the unit of measure is a time difference, it has no real world significance as it is simply a delta we found empirically that works well.

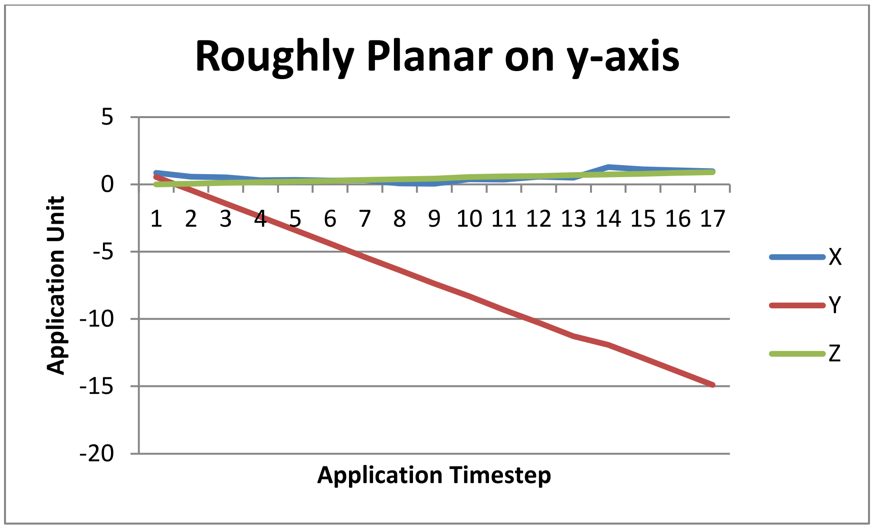

Roughly Planar on x-axis
Application Unit
Application Timestep
5
0
-5
-10
-15
-20
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
X
Y
Z

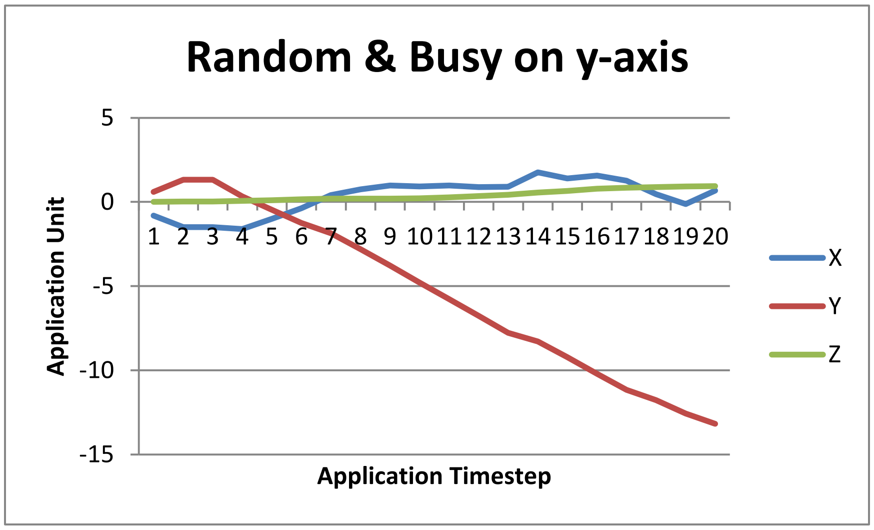

Random & Busy on x-axis
Application Unit
Application Timestep
5
0
-5
-10
-15
-20
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
X
Y
Z

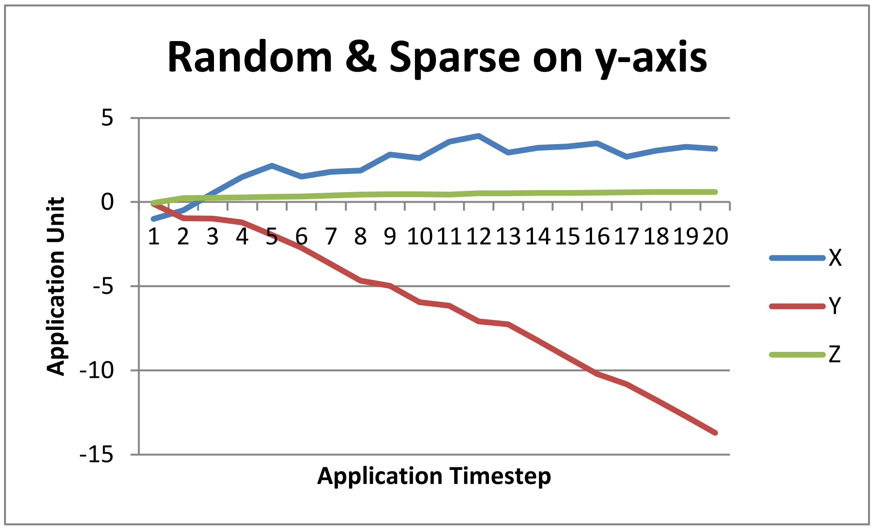

Random & Sparse on x-axis
Application Unit
Application Timestep
5
0
-5
-10
-15
-20
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
X
Y
Z

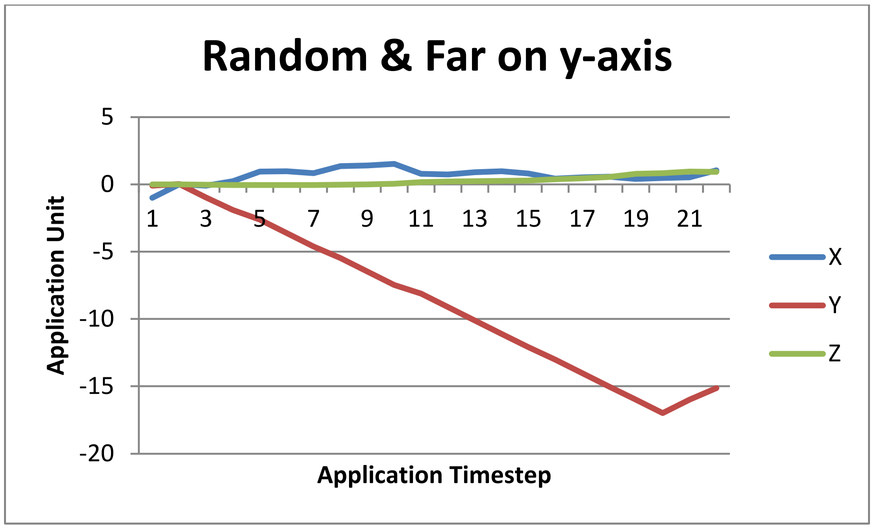

Random & Far on x-axis
Application Unit
Application Timestep
2
0
-2
-4
-6
-8
-10
-12
-14
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
X
Y
Z

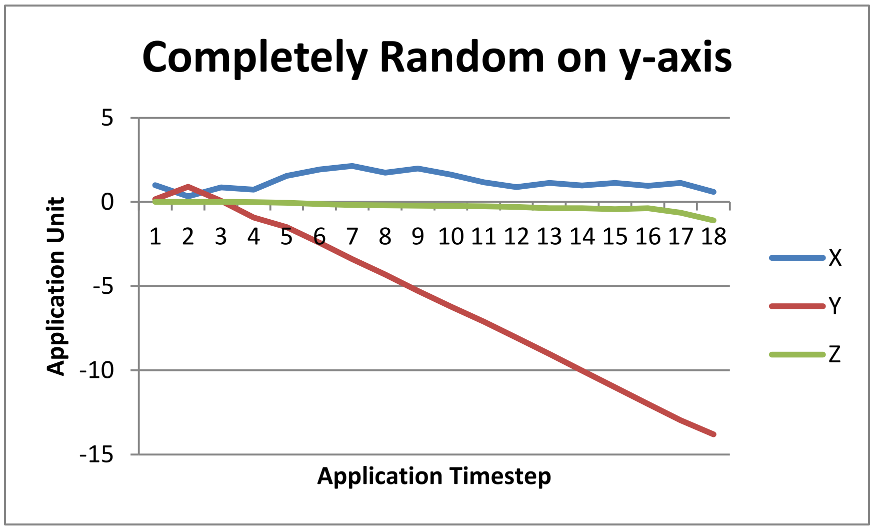

Completely Random on x-axis
Application Unit
Application Timestep
5
0
-5
-10
-15
-20
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
X
Y
Z


Roughly Planar on y-axis
Application Unit
Application Timestep
5
0
-5
-10
-15
-20
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
X
Y
Z


Random & Busy on y-axis
Application Unit
Application Timestep
5
0
-5
-10
-15
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
X
Y
Z


Random & Sparse on y-axis
Application Unit
Application Timestep
5
0
-5
-10
-15
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
X
Y
Z

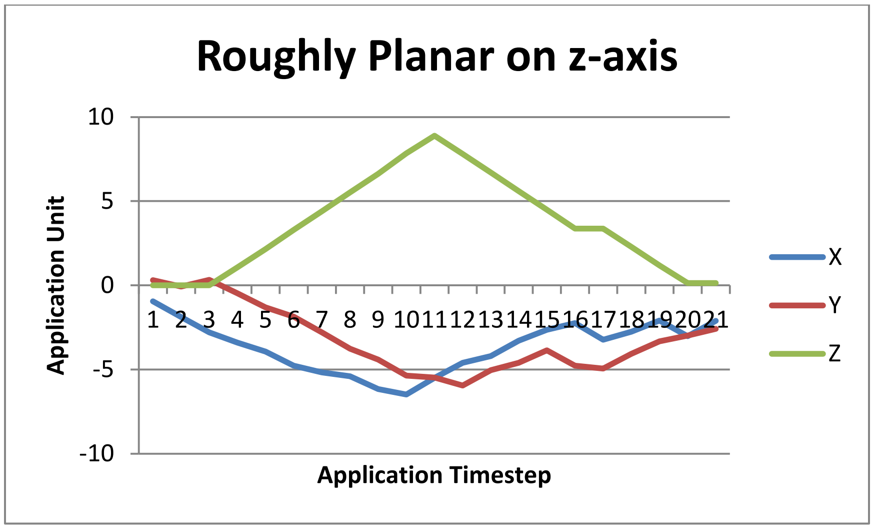


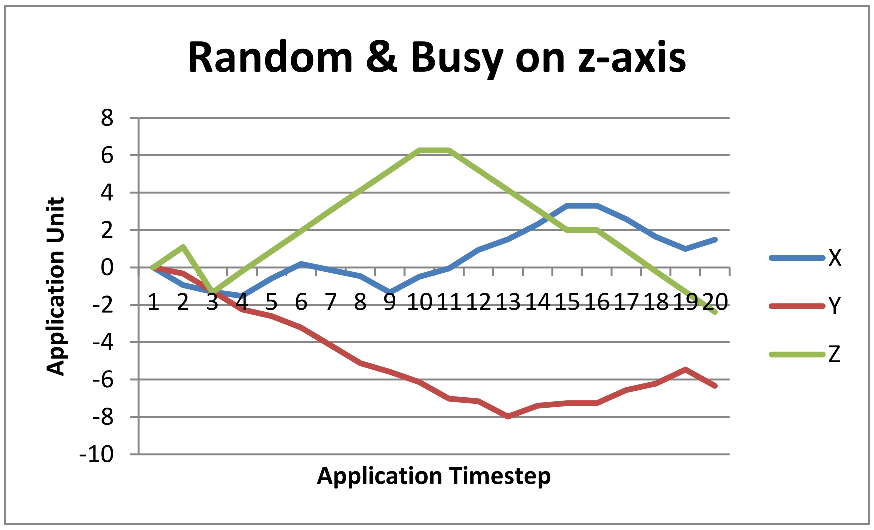


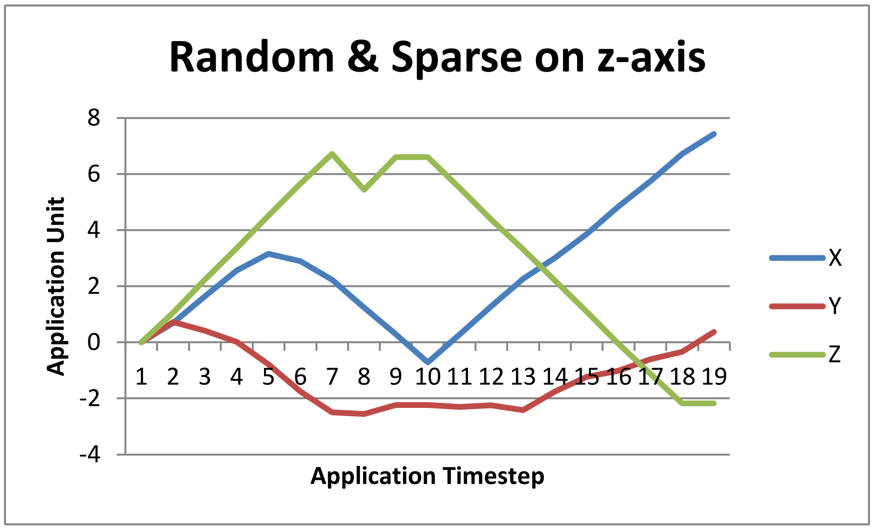
5.3 Discussion of x- and y-axis Translation

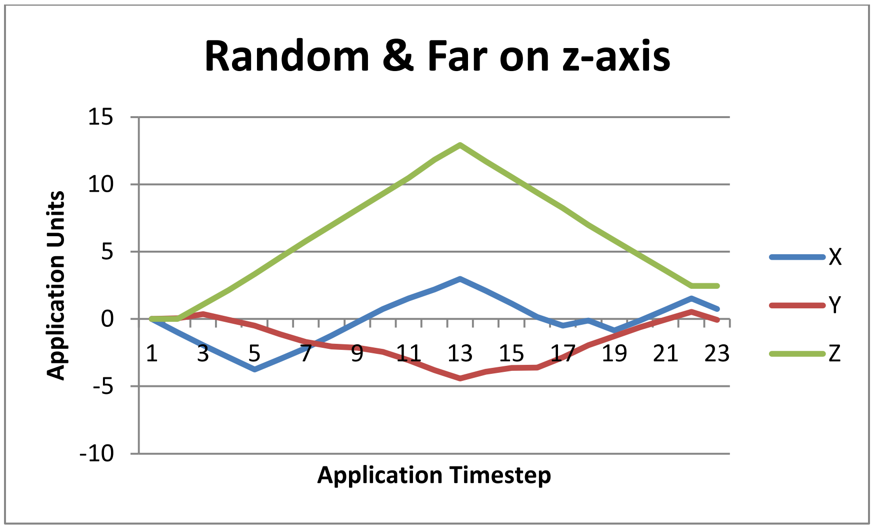
We see that the system is fairly robust to different types of environments and performs well regardless of the types of features it is exposed to. Initially we thought that in sparse enviornments, without many features, it might have issues accurately determining the correct motion, but that did not seem to be the case.

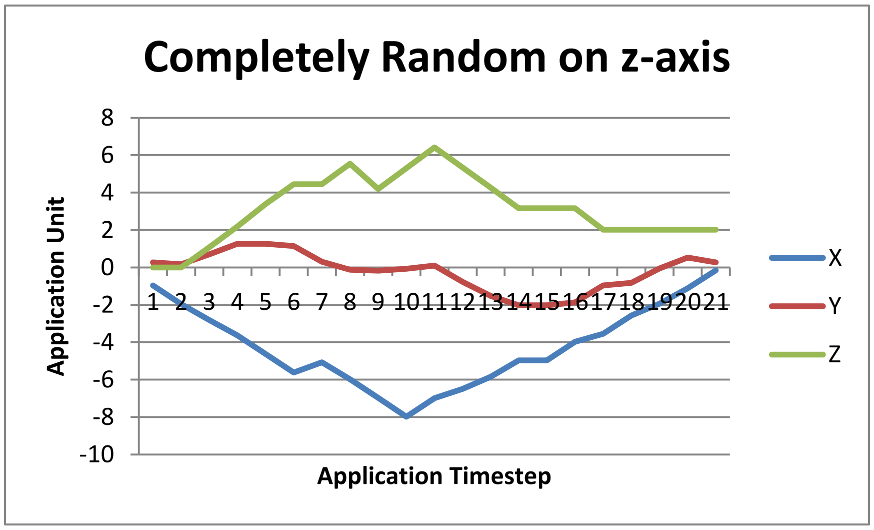
We noticed that on average movements in the y direction are noisier than those in the x direction. In terms of the methods used both are handled in the exact same way so there should not be a difference. We speculate that the difference can be attributed to experimental error. While the x-translation tests were performed by sliding the device across a flat surface for the y-translation it was sliding it vertically along a vertical flat surface which is harder control. Despite that, in every test the system was able to very clearly determine the correct motion.


Roughly Planar on z-axis
10
5
0
-5
-10
Application Unit
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
Application Timestep
X
Y
Z


Random & Busy on z-axis
8
6
4
2
0
-2
-4
-6
-8
-10
Application Unit
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Application Timestep
X
Y
Z


Random & Sparse on z-axis
8
6
4
2
0
-2
-4
Application Unit
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Application Timestep
X
Y
Z





5.4 Discussion of z-axis Translation

Movement in the z direction was not nearly as easily distinguished as in the x and y directions. This can be attributed to feature points not behaving as consistently as a whole. Specifically, when translating in those directions, feature points on average move in the direction of translation (as evidenced by the fact that we can use the average displacement to help with determining a translation) whereas with translations in the z direction this is not the case. In addition, as mentioned earlier, distortion effects (caused by slight rotations) can have a much larger impact on z direction calculations as the convex hull of features will not only change size, but change in shape as well.

Looking at the graphs, the motion seemed to be best interpreted during the *Roughly Planar* test and *Random & Far* tests which intuitively make sense. When features are roughly planar, moving along the z-axis causes them to generally all converge/diverge in a consistent manner. Differences in depth that would normally result in features displacing different distances and the hull shape being distorted are not an issue if they are all roughly at the same depth. As for the *Random & Far*, these same principles hold. At a sufficient distance features are approximated as being roughly planar, as the distances that they move relative to one another are tiny compared to the distance from the camera, so they undergo similar movement and thus we get similar performance in that area.

6 Future Work

There are a number of potential improvements that could be made to the system to make it even more robust and widely applicable.

1. **Tracking non-rigid features:** The current optical flow algorithm we use assumes features are rigid and remain so when computing a translation. This can potentially

limit where the camera can be pointed. There are algorithms that are able to track non-rigid features that could be explored in the future.

2. **Using a different estimation of the essential matrix:** We currently estimate the essential matrix by computing the fundamental matrix and applying the camera intrinsics. There are algorithms that can provide potential more accurate estimates by computing it directly given that feature points have been normalized by the intrinsics.
3. **Testing out different kalman filter models:** We model the orientation and gyroscope using linear models because that is what the basic kalman filter requires. There are other types of filters, such as the Extended Kalman Filter, that accept non-linear models and could possibly model those sensors more accurately.

7 Conclusion

Trying to track motion without any external sensors is far from a trivial task. External sensors offer a number of advantages such as aiding in recalibration and providing a constant reference point. However, we felt that the restrictions imposed by them, mainly that they required setup and could only be used in specific environments, were enough to warrant our attempt at tracking without using them. Although our system might not be quite as robust, it achieved the 3 goals we initially proposed. It is able to replicate the movements from our initial example, it can operate completely independently of any other sensor and it tracks at sub second speeds which are fast enough to be considered real time.

References

[1] Nintendo. [Online] http://www.nintendo.com/consumer/systems/wii/en_na/ts_system.jsp?menu=wiiremote.

[2] **Castaneda, Karl.** Nintendo and PixArt Team Up. *Nintendo World Report.* [Online] http://www.nintendoworldreport.com/news/11557.

[3] **Kumar, Mathew.** Gamasutra. *Develop 2009: SCEE's Hirani Reveals PS Eye Facial Recognition, Motion Controller Details.* [Online] http://www.gamasutra.com/php-bin/news_index.php?story=24456.

[4] Motion Controller Update Part II: Interview with R&D – The Sequel. *Playstation Blog.* [Online] http://blog.us.playstation.com/2009/09/08/motion-controller-update-part-ii-interview-with-rd-%E2%80%93-the-sequel/.

[5] The PrimeSense 3D Awareness Sensor. *PrimeSense.* [Online] http://www.primesense.com/wp-content/uploads/2012/12/PrimeSense-3D-Sensor-A4-Lo.pdf.

[6] **Hardawar, Devindra.** The magic moment: Smartphones now half of all U.S. mobiles. *VentureBeat.* [Online] http://venturebeat.com/2012/03/29/the-magic-moment-smartphones-now-half-of-all-u-s-mobiles/.

[7] Motion Sensors. *Android Developer API Guide.* [Online] http://developer.android.com/guide/topics/sensors/sensors_motion.html.

[8] Samsung Galaxy S II - Specs. *Imaging Resource.* [Online] http://www.imaging-resource.com/PRODS/GALAXY-S2/GALAXY-S2DAT.HTM.

[9] **Harris, Chris and Stephens, Mike.** *A COMBINED CORNER AND EDGE DETECTOR.* Roke Manor, United Kingdom : The Plessey Company, 1988.

[10] **Bouguet, Jean-Yves.** *Pyramidal Implementation of the Lucas Kanade Feature Tracker.* Santa Clara, CA : Intel Corporation Microprocessor Research Labs.

[11] *Shape and Motion from Image Streams under Orthography: a Factorization Method.* **Tomasi, Carlo and Kanade, Takeo.** 2, s.l. : International Journal of Computer Vision, 1992, Vol. 9.

[12] *Bundle Adjustment - A Modern Synthesis.* **Triggs, Bill, et al.** London, United Kingdom : ICCV '99 Proceedings of the International Workshop on Vision Algorithms: Theory and Practice , 1999.

[13] **Zhang, Zhengyou and Shan, Ying.** *Incremental Motion Estimation.* Redmond, WA : Microsoft Research, 2001. MSR-TR-01-54.

[14] *What can be seen in three dimensions with an uncalibrated stereo rig?* **Faugeras, Olivier D.** Valbonne, France : INRIA-Sophia, 2004.

[15] *An Efficient Solution to the Five-Point Relative Pose Problem.* **Nister, David.** Princeton, NJ : Sarnoff Corporation, 2003.

[16] **Longuet-Higgins, H. Christopher.** A computer algorithm for reconstructing a scene from two projections. *Nature.* 1981, 293.

[17] *An Introduction to the Kalman Filter.* **Welch, Greg and Bishop, Gary.** Chapel Hill, NC : Department of Computer Science University of North Carolina at Chapel Hill, 2006.