FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Cross-Platform Tracking of a 6DoF Motion Controller

Using Computer Vision and Sensor Fusion

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Thomas Perl
Matrikelnummer 0725603

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Priv.-Doz. Mag. Dr. Hannes Kaufmann, Assoc. Prof.

Wien, 09.12.2012

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Cross-Platform Tracking of a 6DoF Motion Controller

Using Computer Vision and Sensor Fusion

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Thomas Perl
Registration Number 0725603

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Priv.-Doz. Mag. Dr. Hannes Kaufmann, Assoc. Prof.

Vienna, 09.12.2012

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Thomas Perl
1050 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

This thesis is dedicated to my parents, who fully supported me (financially and otherwise) throughout my studies, so that I could focus on the important parts of university and real life.

A very special thank you to my supervisor Hannes Kaufmann, who was one of the initiators of the MoveOnPC project, and who provided support, guidance and insightful feedback throughout the implementation and writing phases of this thesis.

Special thanks to Benjamin Venditti, who did the initial implementation of the vision tracker module during Summer of Code 2012. Special thanks also to Douglas Wilson for making use of the library, providing feedback and testing during the early stages of the implementation with Johann Sebastian Joust and the UniMove library.

Thanks also to the various people in and around the PS Eye and PS Move hacking community, as well as people who developed libraries that are used in the PS Move API, or built libraries and applications on top of the PS Move API (in alphabetical order): Pascal B. (linmctool), Viktor Budaházi (Windows pairing), Raphaël de Courville (Processing bindings, iSight exposure), Nicolas Devillard (iniparser), Jules Fennis (Processing bindings), Adam Henriksson (Edgar Rice Frotteur), Patrick Jarnfelt (UniMove), justinb26 (Processing bindings), Martin Kaltenbrunner (TUIO_CPP), Sebastian Madgwick (AHRS algorithm), Alexander Nitsch (MoveOnPC Wiki, HID input report and calibration documentation), Alan Ott (hidapi and feedback on patches), Jim Paris (PS Eye reverse engineering), Mikhail Sapozhnikov (psmoveinput driver) and Kenn Sebesta (PS Move hardware hacking).

Thanks to the people who contributed patches to PS Move API in Git: Stefan Derkits, Santiago Ferreira, Fred Oliveira, Krisztián Szabó and Joshua Yanchar.

Thanks to Stefan Kögl and Martin Wieser for proofreading the drafts and providing useful feedback, which has been integrated into the final version of this document.

Last but definitely not least, I'd like to thank Iris for always being there for me and providing welcome distractions and support, as well as being understanding when deadlines and late-night hack sessions occupied unusually large quantities of my free time.

Typeset in LaTeX using Vim. Diagrams and illustrations created with LibreOffice and The Gimp. The AOPATAD artwork by Richard Hogg (thanks!) is used with kind permission as the PS Move API logo. Parts of the implementation of the PS Move API were sponsored by Google, Inc. as part of Google Summer of Code 2012.

Abstract

There is a lack of software for 6DoF (six degrees of freedom) tracking using affordable off-the-shelf hardware. With the introduction of motion controllers in game consoles, the hardware is easily available these days, but no fully-featured software solutions for 6DoF tracking exist.

This thesis introduces the PS Move API, a cross-platform open source library for multiple programming languages that can be used to track multiple PS Move Motion Controllers via Bluetooth and a USB 2.0 PS Eye camera. The library implements sensor fusion to track all six degrees of freedom: 3-axis position and 3-axis rotation.

The library solves the problems of communicating with the controller via USB and Bluetooth using the HID (Human Interface Device) protocol, pairing the controller with the host computer (for Bluetooth connections) and connecting to the controller in a cross-platform manner.

Vision tracking is implemented using the freely available OpenCV framework and a PS Eye camera (other cameras are supported as well). The PS Move Motion Controller has a sphere at its top that can change its color using RGB LEDs - this is used to track the controller and to distinguish between multiple controllers.

Orientation tracking is implemented using an open source AHRS (attitude heading reference system) algorithm, integrating inertial sensor readings from accelerometers, gyroscopes and magnetometers into a quaternion representation, which describes rotations in 3D space.

Sensor fusion combines data from the visual and orientation tracking to get the controller position and orientation relative to the camera position in world coordinates. This data can then be used for different input mechanisms, such as augmented or virtual reality applications.

An easy-to-use API (application programming interface) is provided as part of the library design, allowing quick prototyping and efficient implementation of solutions incorporating the PS Move Motion Controller. Example applications and integrations into existing frameworks such as TUIO and OpenTracker demonstrate different API use cases and validate the results of the implementation.

Kurzfassung

Derzeit gibt es kaum Software für 6DoF (six degress of freedom, sechs Freiheitsgrade)-Tracking, die sich erschwinglicher Off-the-Shelf-Hardware bedienen. Seit Konsolen-Hersteller in den letzten Jahren vermehrt Motion-Controller in ihre Systeme integriert haben, gibt es ausreichend Tracking-Hardware, aber keine vollständige Software-Lösung, um 6DoF-Tracking-Systeme zu entwickeln.

Diese Diplomarbeit stellt das PS Move API vor, eine Cross-Platform Open Source Bibliothek für mehrere Programmierspachen, die es ermöglicht, mehrere PS Move gleichzeitig per Bluetooth und einer USB 2.0 PS Eye-Kamera zu tracken. Die Bibliothek implementiert Sensor Fusion, um alle sechs Freiheitsgrade zu tracken: 3 Achsen der Position und 3 Achsen der Rotation.

Die Bibliothek löst das Problem der Controller-Kommunikation per USB und Bluetooth mit Hilfe des HID (Human Interface Device)-Protokolls. Weiters wird auch das Pairing (Koppeln) des Controllers für Bluetooth-Verbindungen per USB und das Verbinden auf verschiedenen Betriebssystem-Plattformen unterstützt.

Vision-Tracking wird mit Hilfe des frei verfügbaren OpenCV-Frameworks und einer PS Eye Kamera implementiert - andere Kameras werden ebenfalls unterstützt. Der PS Move Motion-Controller hat eine leuchtende Kugel an der Spitze, die mit RGB-LEDs ihre Farbe verändern kann. Mit dieser Funktion kann man den Controller im Raum tracken, und eine Unterscheidung zwischen unterschiedlichen Controllern treffen.

Das Tracken der Orientierung (Rotation) des Controllers ist mit Hilfe eines Open Source AHRS (attitude heading reference system, Lageanzeigesystem) Algorithmus implementiert, der die Informationen von Beschleunigungssensoren, Gyroskopen und des Magnetometers in eine Quaternion-Repräsentation umrechnet. Diese Quaternion-Repräsentation beschreibt die Rotation im 3D-Raum.

Sensor Fusion kombiniert die Resultate von Vision- und Orientierungs-Tracking, um eine Controller-Position und -Orientierung relativ zur Kamera-Position in Welt-Koordinaten zu bekommen. Diese Daten können für verschiedenste Anwendungsgebiete, wie zum Beispiel Augmented oder Virtual Reality, verwendet werden.

Das Hauptaugenmerk beim Design der Bibliothek liegt auf einer leicht zu verwendenden Programmierschnittstelle (API), mit der man schnelle Prototypen und effiziente Lösungen entwickeln kann, die den PS Move Motion Controller verwenden. Beispiel-Anwendungen und die Integration in bestehende Frameworks, wie zB TUIO und OpenTracker zeigen unterschiedliche Anwendungsfälle auf, und validieren die Ergebnisse der Implementierung.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Nowadays, integration of 3D user input into Virtual Reality applications and games is restricted by the availability of affordable hard- and software. In recent years, new motion controllers designed for use in game consoles have been released. These controllers are affordable and widely available.

In 2010, Sony released their Playstation Move Motion Controller. The controller is initially paired with a host computer using USB, the communication can happen via USB or Bluetooth. Given a suitable camera (for example the PS Eye USB 2.0 camera used with the Playstation 3 system), computer vision can be used to calculate the controller's 3D position relative to the camera position by setting the color of the glowing orb built into the controller.

Using additional data from the inertial sensors (accelerometer, gyroscope and magnetometer) built into the controller and accessible via Bluetooth, the data obtained via computer vision can be augmented for high-quality tracking data.

1.2 Aim of the Work

The goal of this work was the development of a library (PS Move API) that can be used on multiple operating systems to communicate with the PS Move Motion Controller. The library provides an easy-to-use API, abstracting away implementation details and operating system differences.

The solution allows application developers on different platforms (Linux, Mac OS X and Windows) to read the calibrated and preprocessed 3D position of the Move controller and access raw sensor data on a lower level.

These features are made accessible via a low-level C API for integration into existing solutions, as well as a high-level API in a higher, dynamic programming language for quick prototyping of new solutions.

The tracking method works with at least two controllers simultaneously, as this is required for some more advanced 3D position tracking applications.

Parts of the implementation of this project were carried out during Google Summer of Code 2012 by Benjamin Venditti and me. Summer of Code was funded by Google Inc., Vienna University of Technology was the mentoring organization.

1.3 Problem Statement

This section lists high-level tasks solved by the PS Move API implementation.

HID Communication

The PS Move Motion Controller acts as a Human Interface Device (HID, specified in [9]) and can be enumerated when connected via USB. Communication is done using custom HID messages. The structure of the reports has been analyzed by projects such as MoveOnPC [5].

Pairing

To communicate via Bluetooth, the controller needs the Bluetooth host address of the target computer written to it via USB. The operating system's Bluetooth stack requires a custom entry of the controller's address to allow connections, because the PS Move does not use the "classic" PIN-based Bluetooth pairing mechanism. Once connected, the communication is carried out via the Bluetooth version of the HID protocol [11].

Vision Tracking

To get the 3D position of the controller relative to the camera, computer vision can be used on the camera image to determine the size (used for distance calculation) and position of the glowing orb in the camera image. Distortions of the camera image due to the lens and color differences due to lighting conditions have to be taken into account for accurate tracking results.

Orientation

The orientation (rotation in 3D space) of the controller can be obtained by integrating sensor readings from the gyroscope, accelerometer and magnetometer over time. The sensor readings can be obtained via Bluetooth. An AHRS algorithm can be used to convert sensor readings into a quaternion representing the orientation of the controller.

Sensor Fusion

Combining the results of both *Vision Tracking* and *Orientation*, the 6DoF pose (3 DoF position and 3 DoF rotation) can be calculated. Additionally, sensor fusion can be used to improve accuracy of vision tracking using additional data from the inertial sensors. This allows short-term dead reckoning using only sensor data when the camera tracking is lost.

Application Programming Interface

To make it easy for application developers to take advantage of the solution, the API to access all the calculated information should be easy to use and should allow for straightforward integration into existing solutions.

CHAPTER 2

Related Work

This chapter highlights important scientific papers related to the solutions the library requires for sensor fusion. The section is split up into sub-sections dealing with problems in the order that they need to be solved: Camera calibration needs to be done before blob tracking yields useful results, inertial sensors have to be read before sensor fusion (of camera and inertial sensor data) can be carried out. Finally, existing approaches are presented.

2.1 Intrinsic Camera Calibration

Camera optics usually introduce (radial and other) distortions to the captured image. These distortions have to be measured and accounted for, either by adjusting the projection of content rendered on top of the camera image, or by undistorting the camera image. A well-calibrated camera is a prerequisite for accurate blob tracking.

The calibration methods described here usually follow the same procesure: A known object or pattern is captured by the camera, and the distortion is determined by comparing the expected image with the captured image.

The projection from 3D world coordinates to 2D camera image plane coordinates is dependent upon several parameters, these can be categorized as follows (from [12]):

- **Extrinsic parameters:** Translation (3 parameters for all 3 axes in 3D space) and rotation (also 3 parameters) of the camera coordinate system origin relative to the world coordinate system origin
- **Intrinsic parameters:** Aspect ratio of the camera image) (s), focal length of the camera (f), principal point (2 coordinates)

The intrinsic parameters of the camera are camera-specific and do not change when the camera is moved. The extrinsic parameters are environment-specific and change when the camera is moved or rotated. As far as camera calibration for the library is concerned, we deal with calibrating the intrinsic parameters here.

Digital camera calibration methods: considerations and comparisons

As a starting point for digital camera calibration, [28] gives a nice overview of the current approaches taken in both computer vision and (close-range) photogrammetry. Calibration is described by the authors as a "necessary prerequisite for the extraction of precise and reliable 3D metric information from images". Several parameters must be known for a camera to be considered "calibrated" in [28]:

- Principal distance
- Principal point offset
- Lens distortion

Calibration methods are then categorized into different aspects, such as the methods and models used (perspective projection vs. projective camera model), implicit (visually correlating point positions) or explicit (physically interpretable models) models or 3D vs. planar point arrays. Another distinction that is made is between point-based and line-based calibration methods.

Experimental tests were then carried out, comparing different methods and software packages for camera calibration using a 3D test field (with the object stationary and the camera moving) and a planar object (chess board pattern, with the object moving and the camera stationary).

A very interesting aspect of the findings in [28] is the fact that for most off-the-shelf consumer cameras sold today, there can be a difference in the radial distortion (which has to be measured by the camera calibration in order to revert it in the undistorted image) for each color channel in the camera image, due to the way the image sensors are laid out in consumer cameras (usually using a color filter array using a Bayer pattern). Ideally, the radial distortion can therefore be different for each color channel, and must be corrected for each channel separately. This requires that the image data is available in the corresponding RAW format to work on separate channels.

In the case of the PS Move API, we are dealing with low-cost consumer webcams, where getting the RAW sensor image is usually not possible or (due to bandwidth constraints and a minimum required framerate) not feasible, so per-channel radial distortion compensation is something that cannot be implemented easily. However, given that the PS Move tracking relies on separate colors (when tracking multiple controllers) and accurate 3D metric information, it would be desirable to do separate camera calibrations for each channel (and maybe also work on separate color channels for blob tracking).

Camera Distortion Calibration using a Chess Pattern

Because cameras usually have distorted pictures as a result of their optics, a calibration procedure has to be carried out for each camera model to make straight lines in the world appear as straight lines in the camera image. In [33], a chess pattern is used for calibrating the camera.

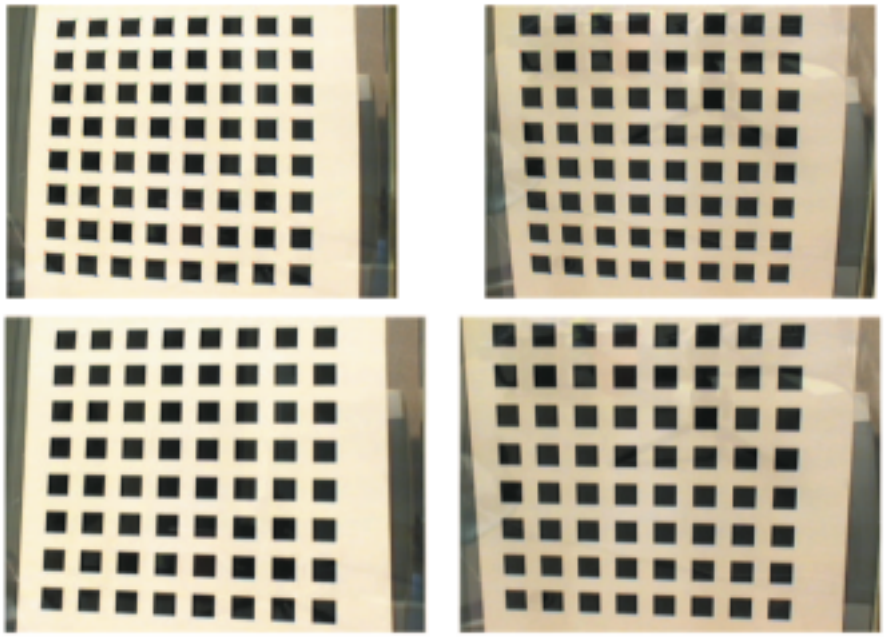


Figure 2.1: The pattern before (above) and after (below) distortion removal (from [33])

An example of the chess pattern in the camera image can be seen in figure 2.1 - the 8x8 pattern of black squares results in 256 different corners for which the location is determined in the camera image. The pattern itself is 17x17cm and has been printed on high-quality paper and placed on glass. Based on the locations of the corners, the intrinsic lens distortion can be calculated and then removed.

The solution proposed in [33] requires at least two images of a planar pattern shown from different angles. To achieve different angles, the camera or the pattern can be moved (in the PS Move API, we'd usually keep the camera steady and move the pattern). An important point here is that the pattern is placed on a "reasonably planar surface", because otherwise the distortion of the pattern is a result of both the camera lens distortion and the non-planarity of the surface (e.g. a bent sheet of paper).

Zhang also lays out a recommended calibration procedure in [33] as follows:

1. Print a pattern and attach it to a planar surface
2. Take a few images of the model plane under different orientations
3. Detect the feature points in the images
4. Estimate the five intrinsic parameters and all the extrinsic parameters
5. Estimate the coefficients of the radial distortion
6. Refine all parameters

The OpenCV library [14] provides built-in functions for doing camera calibration using a similar approach that also uses a printed chess pattern.

Geometric Camera Calibration Using Circular Control Points

Instead of using a checkerboard image, [12] starts out with a 3D object with circular control points. The author also points out that in order to properly calibrate a camera, external error sources must be suppressed as much as possible. The extend of accuracy required depends on the use case: Metrology has much higher requirements for accuracy than robot guidance. In the case of the PS Move API, accuracy isn't as important as ease of use and high frame rate (for the camera undistortion / projection calculations).

The camera model is defined as the mapping of 3D to 2D coordinates on the image plane. Examples of the projection are orthographic and perspective projections. While the orthographic projection isn't well-suited for real-world images, the perspective projection is much better suited, and is often augmented with an additional lens distortion model to match real-world conditions more closely. An example of a pure perspective projection without any lens distortion is the pinhole camera model, which can be used to approximate real-world cameras, but because of the physical properties of image sensors used in consumer cameras, no camera matches the pinhole camera model exactly.

In the PS Move API using multiple controllers, the intrinsic parameters would be the same for each controller, as they are bound to the camera, and the extrinsic parameters would be different, as they describe the translation and rotation of the camera relative to the world coordinate system origin (assuming that for each controller, we map the world coordinate system so that the origin is at the center of the sphere, and the axes are lined up with the inertial sensor axes of the controller).

In [12], the calibration takes advantage of the property of circles that they either become ellipses or (as special case of an ellipse) circles when observed by a camera. Determining the ellipse center in the image gives the location of the circle center, avoiding projection errors that happen when using other kinds of objects. To account for radial distortion and decentering distortion, a total of 8 intrinsic parameters are used.

Tests were carried out using 200 synthetic images and one real-world image that was used as a reference model. The synthetic images were processed with a blur filter to more closely match real-world conditions. As calibration object, two perpendicular planes were used with 256 circular control points on each plane. The position and arrangement of the control points was known, and accurate knowledge of the control point layout is a prerequisite for the calibration method to give good results.

An interesting insight of [12] is the list of error sources in the real world that influence the camera calibration:

- **Real vs. synthetic pictures:** The projection model of the synthetically-generated pictures does not match the real projection of the camera, thereby introducing a projection error - this error is especially visible when using cameras with wide-angle lenses.

- **Illumination changes:** In [12], the calibration was carried out with two different light sources (fluorescent and halogen lamps) and the results compared. Due to the chromatic aberration caused by different lighting, the points might appear out-of-place or magnified in the camera image, resulting in different observations for the control point coordinates.

- **Camera electronics:** This error source was described as line jitter (noise in horizontal placement of pixels).

- **Calibration target:** As described above, the control point coordinates and size/layout must be known exactly for the calibration algorithm to work. Wrong data here leads to wrong calibration results.

Especially the point about illumination changes is an important point to consider for the PS Move API tracker, not necessarily for camera calibration, but for blob tracking in general - different lighting conditions (or exposure settings) will make the sphere appear magnified in the camera image, leading to different radiuses observed (and subsequently to the wrong distance reported).

2.2 Blob Tracking

Given a calibrated camera image, the next step for sensor fusion is to track the controller's sphere in the camera image, which is referred to as blob tracking, or (in the case of circular blobs) sphere tracking. The results of sphere tracking are the 2D position and the size (radius) of the sphere. Mapping the radius to a depth value and combining the 2D position with the depth gives a 3D position.

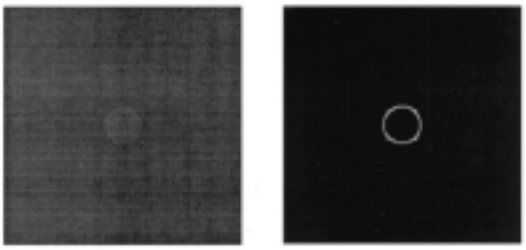
Tracking algorithms usually take advantage of either the shape (e.g. circle) or color of the object to be tracked. In the case of the PS Move API, both the shape (circle) and color (by setting the RGB LEDs) can be used for tracking.

Edge-based Sphere Tracking

One approach of vision-based sphere tracking was implemented by [15] using 2D Hough transforms. In that specific example, circles are detected using edge detection, followed by a voting phase on a 2D search space (the X and Y coordinates of the circle center). The third parameter (the radius of the circle) is calculated using a histogram after candidates for the center point have been determined.

As can be seen in figure 2.2, this approach was mostly used for detecting circles that don't stand out from the rest of the image except for their borders.

This method splits the usual 3D search space (parameters x_0, y_0 and r) into a 2D search space (x_0 and y_0) followed by a 1D histogram search (r). The center search utilizes "voting" - each pixel in the search space (usually $size = image\_width \times image\_height$, but a more fine-grained search space is also possible). A high vote count only provides an indication of a possible center point - this indication has to be verified. The verification step determines the



(a) (b)

Figure 2.2: Example input image and edge-detection result (from [15])

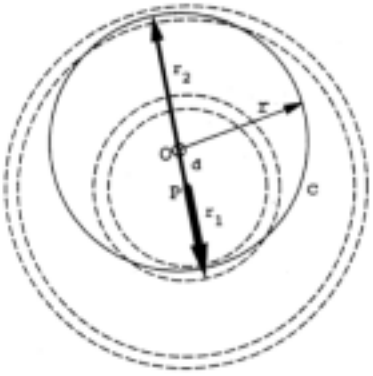


Fig. 5. If an inaccurate estimate of the centre of the circle is provided by the first step, the radius histogram will give two peaks. In this example, the estimated centre is point P, while the true centre is point O. Obviously, these peaks will appear at bins with distance r_1 and r_2 from the estimated centre.

Figure 2.3: Errors resulting from wrong center point (from [15])

radius and if the pixel is really a center point. Having two possible candidates for the radius might suggest that the calculated center point is off from the real center point (see figure 2.3).

Errors in this approach can happen from different sources (from [15]):

- **Digitization errors:** Coming from the pixel raster of the camera and the limited resolution that the camera provides.
- **Distortion:** Coming from the camera optics or from movements while the image is recorded (motion blur).
- **Noisy additional pixels:** Random noise from the sensor or the environment itself.
- **Missing obscured pixels:** Items between the circle and the camera can obscure parts of the circle in the camera image.

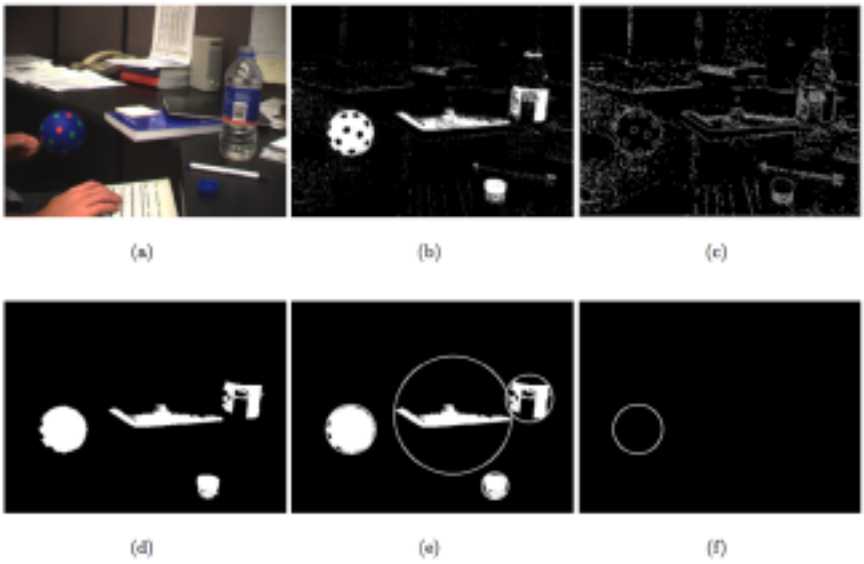


Figure 2: Locating the perspective sphere projection. a) Input image; b) Binary image; c) Set of all contours; d) Filtered contours; e) Minimum enclosing circles; f) Most circular contour (chosen to be the projection of the sphere).

Figure 2.4: Image filters used to determine the sphere location (from [3])

- **Inaccurate center estimation:** When the center point has not been properly determined, two incorrect candidates for the radius will appear in the histogram search instead of the single correct radius (figure 2.3).

Using additional image filters before the voting can increase the quality of the algorithm's output values, especially in noisy environments. Parameters can be used to tune the algorithm depending on how much of the circle's border is visible in the resulting image.

Color-based Sphere Tracking

Another approach for tracking spheres in a camera image has been implemented by [3]. Instead of relying on edge detection and the edges of the circle/sphere, color is taken into account to filter other elements in the camera image (figure 2.4). Additionally, dots with different colors are placed on the sphere to allow tracking the 3D orientation of the sphere (figure 2.5).

The color-based sphere tracking approach has the advantage that partial occlusion is handled well as long as at least half of the sphere is still visible in the camera image. This approach doesn't use 2D Hough transforms at all, and relies solely on color. Compared to other tracking mechanisms that are mentioned in the paper, [3] works with only a single camera and in real-time.

Figure 5: Locating the projections of the dots on the sphere. a) Input image; b) Restricted input; c) Ellipses found for the red (filled) and green (unfilled) dots; d) Exact dot locations.

Figure 2.5: Image filters used to determine the sphere orientation (from [3])

For tracking the orientation of the sphere, 32 colored dots (16 red and 16 green) are placed on the blue sphere in a special layout that avoids two dots being merged together in the camera picture by leaving enough space between each dot.

To locate the sphere and the dots, the camera image is converted from the RGB colorspace into the HSV colorspace, which allows hue-based tracking of the colors that is stable even under different illumination conditions.

In general, this approach provides a good way of detecting colored spheres using the HSV colorspace without relying on 2D Hough transforms. Orientation tracking using colored dots also works well, but in the case of the PS Move we do have inertial sensors that we can use for orientation tracking, so only the first part (3D position detection) is relevant for this project.

Mean-Shift Blob Tracking with Adaptive Feature Selection and Scale Adaptation

When tracking objects against a background, non-occlusion tracking failures can happen because of changes in the background (new objects, removed objects, general color appearance) or because of changes in the object itself (size or appearance). In [19], this problem is analyzed, and a mean-shift algorithm with adaptive feature selection is presented to deal with changes in appearance and size of the tracked object, as well as with changes in the background.

To deal with size changes (scaling), a simple scale adaption method is used: The kernel window is iteratively shifted and adjusted to fix the new object boundaries. In the PS Move API, we also have to shift the sub-area of the camera image that is processed (this makes tracking easier, faster and in general also more stable).

Experiments in [19] have been carried out with three different sample videos: First, a video of a car driving around a parking lot and then accelerating on a highway. In this video, the size and movement of the object changes rapidly. The algorithm presented in [19] successfully deals with these changes - without the algorithm, tracking is lost because of a reflection in the background. In the second video, a different car scene is used, the camera in this case had some problems, so there were duplicate frames and focusing issues, but the adaptive algorithm was able to deal with these issues in the input material. Finally, a soccer scene was used to demonstrate the robustness of the algorithm against major changes in the background image:

The tracked object is a soccer player, and the tracking continues successfully when the player moves from inside the shadow of the stadium to outside the shadow.

In the PS Move API, we also have to make sure to adapt to the changing size and appearance (color) of the Motion Controller in the camera image. Failure to do so might result in slower tracking results (as the tracking has to be re-started from an unknown position instead of just shifting the region of interest).

Another important result of [19] that can be used for the PS Move API implementation is that the results show that pixel values change sharply around object boundaries, making the pixel values at and around the boundaries extremely useful for slowly shifting the tracking area around.

BraMBLe: A Bayesian Multiple-Blob Tracker

Tracking multiple objects is sometimes important in computer vision blob tracking. In [16], the goal is to track multiple persons (an unknown number thereof that will vary over time) with a single camera. As with all blob tracking algorithms, a calibrated camera is a prerequisite for tracking.

The authors describe the usual approach for multiple-blob tracking: First, the background subtraction takes place, then foreground modelling takes place, which usually just uses a simple temporal filter with a constant velocity predictor. Also, only rudimentary occlusion filter is said to be made for multi-object tracking. Avoiding occlusion filtering for multi-person tracking can be achieved by mounting the camera high enough (so that it nearly points down vertically), avoiding occlusions altogether rather than dealing with them in the algorihm.

As the procedure outlined in [16] deals with a bayesian distribution, an "observation likelyhood" measure has to be introduced. This measure describes the likelyhood that a configuration of objects gave rise to an observed image. The configuration itself describes the number, position and size of the objects. For example, a configuration of two persons, one of size A and position B and another one of size C and position D could have a 70 percent likelyhood of giving rise to a certain image E captured by the camera.

To model persons' appearances, a generalised-cylinder object model is used in which a person is modeled as a cylinder with 4 horizontal discs of different diameter. If the camera is mounted sufficiently low, a person in the camera can be described as the area that 4 parallel lines (the side view of the horizontal discs) span in the camera image. In addition to modelling the shape of the objects to track, the appearance and disappearance of objects (persons entering and leaving the room) also has to be described via a distribution.

Problems with this algorithm were false positives at reflective areas of the image, in which the person appeared - normal filters were not able to deal with this problem. Also, tracking was problematic in cases where two people cross in front of a third person - while tracking was not lost, the three objects could not be separated by the tracking algorithm.

In summary, the research in [16] shows that even complex shapes such as the human body can be approximated by rough shapes (such as cylinders) with good tracking results. For the PS Move API, the shapes we need to track have a circular shape, and have distinct colors, so the tracking problem is reduced a bit. While multiple objects have to be tracked in the PS Move API as well, the number of objects to track (and even their color) are known to the algorithm.

Real-time 3D gesture visualisation for the study of Sign Language

Gesture visualisation and recognition is very useful in the context of sign languages. [23] presents "ThirdEye", an interactive visualisation tool for movement analysis. The French sign language was recognized as full language in France in 2005. Signed languages need a different script than written languages, as written languages are linked to the verbal representation, whereas signed languages are more related to the real-world concepts.

The core hypothesis in [23] is described as: "The realization of gestural sign (sic) casts traces in space that have a scriptural quality". In linguistics, segmentation of the traces and possibly automatic translation plays a vital role. The dynamic of the movement is thought to be a good indicator of the automated segmentation (compared to having only the trace without temporal information).

The ThirdEye motion capture system is divided into distinct parts:

- **The devices:** The device consists of two luminous spherical markers (one green, one blue) that can be clipped onto each hand of the signer, a PS Eye camera set to low exposure and a foot switch to trigger the writing
- **The algorithm:** Two threads deal with the data processing: One thread deals with the capture part of the code, another thread deals with rendering the traces

To determine the position of the colored spheres, a simple algorithm is used:

1. Look at random pixels, find one with the right color (the iterations here have an upper limit, to avoid looping forever)
2. Go left/right (up/own) - "draw lines" from the pixel with the right color to find the border of the colored sphere
3. For the drawn lines, take the center of each line and combine the two centers to get the sphere center (this works because the sphere always appears as a circle in the camera image)
4. For the next frame, skip step 1 and use the current center as the starting point for steps 2 and 3

The sphere size is determined by systematically filling the majority of pixels in the color area. From the pixel count (area), the sphere size can be determined, and subsequenty the depth (distance of the sphere from te camera) can can be deduced.

The solution presented in [23] has several aspects in common with the PS Move API's visual tracker part. The algorithm used for determining the position and size of the colored blobs demonstrates an existing and low-cost (in terms of CPU time) solution. The PS Move API also uses the PS Eye camera with low exposure because of the good performance and off-the-shelf availability.

2.3 Inertial Sensors

With the blob tracking step resulting in a 3D position, the 3D rotation of the controller must be determined separately (solutions like [3] can also track the orientation from the camera image, but the PS Move does not provide colored dots on its RGB LED-colored sphere).

Representation of rotations in 3D space is best done using quaternions, to avoid problems with gimbal lock that can happen when using Euler Angles (see [8] for a description of quaternions and their use in computer science and geometry).

The three types of inertial sensors that are available in the PS Move Motion Controller are (all sensors are susceptible to noise in the raw readings):

1. **Accelerometer:** Accelerometers measure acceleration relative to free fall. When the controller is kept steady, the measurement gives the acceleration towards the earth's center (gravity). The gravity vector can be used to determine the controller's orientation, but cannot be used to determine the rotation on the axis parallel to the gravity vector. During movement, the accelerometer measures the combination (sum) of the gravity vector and the movement acceleration direction vector.

2. **Gyroscope:** Gyroscopes measure the angular velocity around a specific axis. When the controller is kept steady, an ideal gyroscope reads zero, but in general, gyroscopes usually have a measurement bias. This bias introduces errors when the readings are integrated over time to give the angular rotation. For this reason, the angular rotation readings drift over time. The advantage of gyroscopes are the quick response times.

3. **Magnetometer:** Magnetometers measure the direction of the magnetic field. This can be compared to accelerometers, but instead of pointing towards the earth's center, the magnetic field points to the magnetic north pole, which (depending on the location on earth) makes it possible to determine the rotation around the axis parallel to the gravity vector. Magnetometers are susceptible to interference from local magnetic fields (electronic devices, metal furniture, ...).

All these inertial sensors are implemented as three-axis sensors in the PS Move Motion Controller. In general, the gyroscope can be used for short-term rotation detection while the accelerometer and magnetometer together can be used to have a stable reference orientation towards which the rotation can be re-adjusted to over time.

An additional use case for inertial sensors is position estimation when the blob tracking algorithm fails to determine the 3D position of the controller (e.g. because the controller is obscured in the camera image). The use of inertial sensors for position prediction/estimation is known as dead reckoning.

Orientation Filter for Inertial Sensor Arrays

Combining the accelerometer, gyroscope and magnetometer sensor readings into an estimation of the controller orientation is a prerequisite for sensor fusion. It also helps with applications

where the orientation of the controller should be used as part of the input data (e.g. pointing applications).

In [21] two new orientation filter algorithms are presented, depending on the availability of sensors and the desired output accuracy:

- **IMU** – This algorithm only depends on a 3-axis accelerometer and a 3-axis gyroscope. The output is a quaternion describing the attitude relative to gravity (see figure 2.6)
- **AHRS** – This algorithm builds on top of the IMU algorithm, but includes an additional 3-axis magnetometer, which measures the magnetic field. The output is a quaternion describing the attitude relative to both gravity and the magnetic pole.

The quaternion representation has been chosen, because it is used as the default representation in many software packages dealing with 3D orientation representations, and because it avoids the gimbal lock of Euler angles. The Euler angles can be calculated from the quaternion representation.

A very simple orientation algorithm simply integrates the gyroscope readings (which measure rotational velocity) over time. This has to be compensated for noise and sensor bias, otherwise the result would drift quickly. Using the accelerometer readings, a vector pointing towards the earth's gravitational center can be obtained (assuming no extra movement), which can be used to determine the orientation relative to earth's gravity, but rotations on the axis parallel to the gravity vector cannot be measured with this method. To also get a reference heading for rotation parallel to the gravity vector, a magnetometer is needed. The magnetometer will return a vector describing the direction of the magnetic pole of the earth, working like a compass.

Fusing the information from the gyroscope (quick response), accelerometer (orientation relative to gravity) and the magnetometer (orientation relative to the magnetic field), a good estimate of the real orientation can be made. Comparing the proposed algorithm with existing algorithms based on Kalman filters (see figure 2.7) shows that the performance compares to and exceeds that of the Kalman-based filters, while having lower computational requirements and lower implementation complexity.

An optimized C implementation of the IMU and AHRS algorithms have been made available as open source and can be adapted for other applications.

Camera Attitude Calibration using Inertial Sensors

When working with cameras for which the focal distance is unknown, one can calibrate the camera parameters by determining parallel lines in the camera image and finding their vanishing point (see figure 2.8 for an example image with vanishing point).

In [20], two cameras and an inertial sensor package are used to determine the horizon and focal distance of a camera system. The proposed solution can work with only one vanishing point, but the authors also used two vanishing points to make their results comparable with other methods.

As far as hardware is concerned, two cameras are used for stereo vision, and in between the two cameras, sensors are mounted: A three-axis accelerometer, three gyroscopes and a two-axis inclinometer. When the device is not moving, the length of the accelerometer vector is the

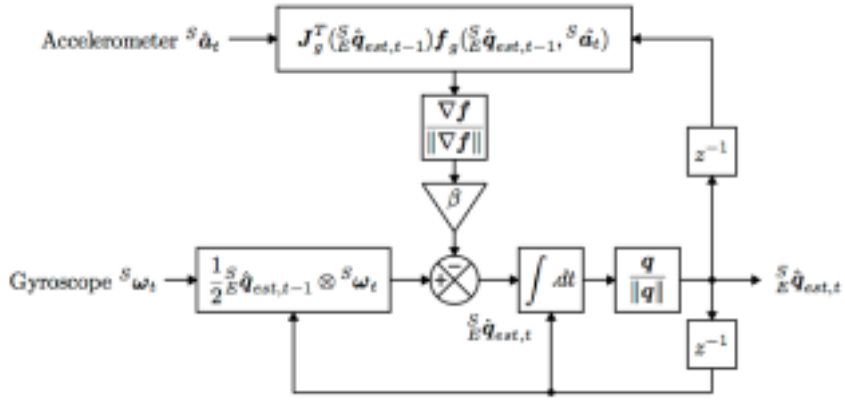


Figure 2.6: Block diagram of the IMU orientation filter algorithm (from [21])

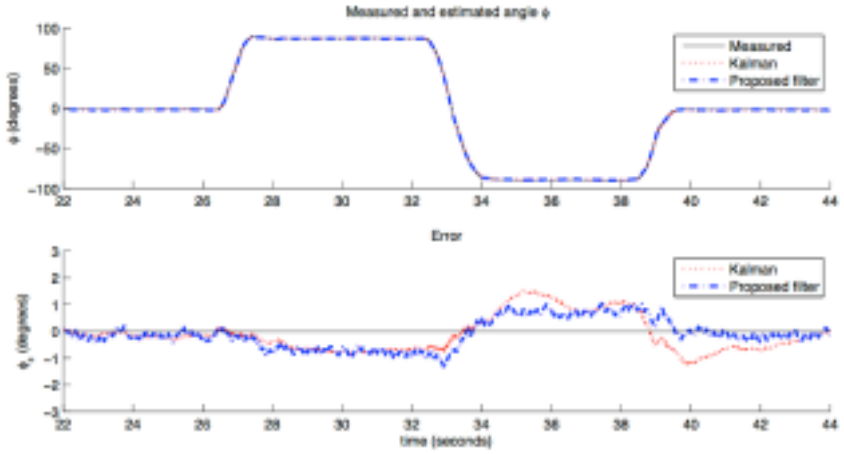


Figure 2.7: Measured vs. estimated angle and estimation error comparison (from [21])

same as gravity - this can be used to determine the attitude of the device. When movements are detected, the attitude is not calculated from the accelerometer readings.

In order to obtain a vanishing point in the camera image, computer vision algorithms have to be applied to the image. Edge detection can be used to find parallel lines on the ground plane. After the edge detection phase, a 2D Hough line transform can be used to find the parameters of the lines. The lines are then sorted by their angle and grouped into pairs of two (see figure 2.9 for an example). From each pair, the vanishing point can be calculated - this vanishing point gives an estimate of the horizon position in the camera image.

The solution shows how to combine inertial sensor data with computer vision output and how to determine properties of the camera optics by just looking at the optical output and using information from the inertial sensors that are mounted together with the camera.

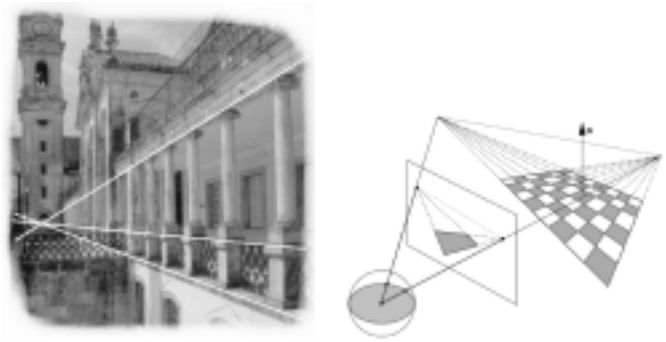


Figure 3: *Via Latina* of Coimbra University with vanishing point, and vanishing line of planar surface.

Figure 2.8: Example of a vanishing point in a camera image (from [20])

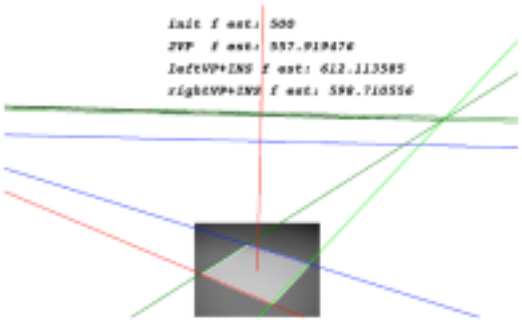


Figure 5: Target image and estimated values of f. The vanishing lines are shown, as well as the nearer vanishing point. The lower horizon is based on an initial estimate of f and $\boldsymbol{n}$, and the others are based on the the left and right vanishing points and $\boldsymbol{n}$.

Figure 2.9: Measured and estimated horizon from two vanishing points (from [20])

2.4 Sensor Fusion

Sensor fusion describes the process of combining (fusion) data from different sources (sensors) into a combined piece of information. In the case of the PS Move API, the visual sensor data (3D position obtained from the camera via sphere tracking) is combined with the inertial sensor data (3D orientation obtained from the accelerometer, gyroscope and magnetometer via an AHRS algorithm). The result is an accurate representation of where the controller is located in 3D space, and how it is rotated (where it is pointing).

In this section, different visual-inertial sensor fusion approaches are presented and compared. At the end of the section, the One Euro Filter [4] is presented as a filter algorithm that can be used at the end of sensor fusion to smoothen the sensor fusion results.

Visual-Inertial Sensor Fusion

In [2], computer vision and inertial sensors are used to combine the benefits of computer vision (accurate, non-drifting tracking) with those of inertial sensors (responsive results with no outliers). The authors use an IMU sending updates at a rate of 100 Hz and a camera with 25 Hz (at a resolution ot 320x240 pixels).

The visual tracking works by rendering a CAD model of the scene and correlating detected features in the CAD model with those of the camera image. If only a few features are found, additional features are extracted from the image. The inertial data is used to augment the filter input and algorithm output of the sensor fusion system.

Two important points are made with regard to sensor fusion in [2] concerning vision and inertial sensor input:

1. **Outlier rejection.** Inertial sensor do not produce outliers (even though they have noisy, drifting output), while vision-based methods sometimes detect sporadic outliers that might affect the algorithm output depending on the filter used. In [2], this problem is dealt with by calculating a scalar based on the state of the system, and using a threshold value to decide whether to accept or reject a given reading.

2. **Divergence monitoring.** When no visual features are detected, the system relies solely on (biased and drifting) inertial sensor data. To compensate for that, the vision image can be compared with the CAD model rendered at the estimated position. If no vision image is obtained, time can be used to increase a divergence threshold. Again, if the divergence monitoring test fails, the system can assume that the current state has diverted, and re-start with the initialization process.

In general, [2] argue that combining inertial sensor data with data from computer vision can yield better tracking quality, while also reducing the demand on the quality of vision measurements (for equal or better results).

A disadvantage of the approach taken by [2] is that they rely on rendering a textured 3D model of the environment in which the system is used. This adds additional set-up work for getting the system to run, and also adds more processing power requirements to the hardware, as it has to render and then analyze the rendered CAD model.

Tightly integrated sensor fusion for robust visual tracking

In [18], inertial sensor data (gyroscope-only) is used to provide predictions for a visual sensor's movement even in the case of motion blur. A model of the scene's content has to be prepared off-line using CAD. The vision-based tracking used only works for slow movements - once the movement between two consecutive frames becomes too big, the local search of the new camera pose fails, and the system requires manual re-initialization.

The fitting of the camera pose estimation is done using edge detection: First, the camera image is obtained, then the edges of the CAD model are rendered using the old (or predicted) camera pose. After that, edge detection is applied to the camera image, and the rendered and detected edges are compared, and a new estimation of the camera pose is calculated.

The gyroscope used in [18] was sampled at a frequency of 171 Hz - as raw data was read from the sensor, calculating and removing the sensor bias had to be done as a calibration step. It was also noted that the gyroscope output was very sensitive to room temperature, and that a good calibration and bias calculation was very important for accurate pose estimation.

Based on the inertial sensor data, motion blur in the camera image could be predicted, and accounted for in the image processing / edge detection step. In the summary, the authors note that accelerometers were not used, but could improve the results (position prediction of the camera).

Compared to [18], the camera in the PS Move setup described in this thesis does not move, has a high frame rate (up to 120 Hz) and gives control over the exposure time, which eliminates most motion blur-based problems. Also, the PS Move setup most (obviously) allow for tracking moving objects, whereas [18] is suited more for moving a camera around a static scene.

As far as gyroscope output and calibration is concerned, the PS Move setup does not depend on perfect gyroscope calibration to the same extend as [18] does - errors accumulating in the gyroscope data processing can be corrected over time using the accelerometer and magnetometer readings in the PS Move Motion Controller (a well-calibrated gyroscope is still useful and desirable, so that the bias error is minimized in the first place).

Hybrid Inertial and Vision Tracking for Augmented Reality Registration

Another approach at combining vision tracking with inertial sensors for use in augmented reality applications is presented in [32]. The solution presented combines the advantages of vision tracking with the advantages of inertial sensors while avoiding the disadvantages of each technology, as the two technologies are complementary.

Maintaining an accurate relationship between the real objects and those rendered on top of the camera image is one of the challenges in augmented reality solutions today. Sensor fusion is one of the ways to improve the quality of tracking.

Different categories of tracking technologies are outlined in [32]:

- **active-target:** powered signal emitters and sensors are placed in a prepared and calibrated environment
- **passive-target:** ambient or naturally occuring signals are measured by sensors (e.g. a compass sensing the Earth's magnetic field, vision systems sensing intentionally placed fiducial markers
- **inertial:** completely self-contained, sensing physical phenomena created by linear acceleration and angular motion

Interestingly, the magnetometer (compass) sensor is not categorized as inertial sensor here, because it measures signals that occur naturally on earth. Similarly, one could argue that the accelerometer also measures (amongst linear acceleration caused by movement) the natural phenomenon gravity. In this thesis, the magnetometer is always categorized as inertial sensor for simplicity.

According to the categorization from [32], the PS Move API can be considered an "active-passive-inertial" tracking system:

- the controller itself is an active-target for the vision tracking (camera), because the color can be changed, and the color tracking only works when the controller is powered
- the magnetometer is a passive-target tracking technology
- the accelerometer and gyroscope are inertial tracking technologies

For the inertial sensor hardware, an Intersense IS-300 three-axis gyroscope is used, which incorporates a gravity sensor and a compass to account for the bias drift.

Four coordinate systems are used in [32]:

1. The world coordinate system: An absolute reference point that describes the origin of world coordinates
2. The camera-centered coordinate system: The center point of the camera - relative to the world coordinate system, it not only describes the position but also the orientation of the camera
3. The inertial-centered coordinate system: This coordinate system is offset a bit from the camera-centered coordinate system (because the inertial sensor is not mounted directly at the camera center), but the orientation is the same as for the camera-centered coordinate system
4. The 2D image coordinate system: The coordinate system of the camera frame as it is captured (projected from 3D space from the camera position)

In the PS Move API, the inertial-centered coordinate system has its origin at the position of the controller, the camera-centered coordinate system works both as camera coordinate system and world coordinate system (in the PS Move API, the camera is usually not moving, so it can be used as a reference point) and the 2D image coordinate system is again the captured camera frame.

One of the conclusions drawn in [32] is that accurate calibration of the two coordinate systems (inertial and vision) is important for proper integration of both subsystems.

One Euro Filter: A Simple Speed-based Low-pass Filter for Noisy Input in Interactive Systems

While all related work in this chapter so far dealt with sensor fusion itself, the One Euro Filter [4] itself does not deal with sensor fusion, but only with filtering of noisy input data. The One Euro Filter is used in the PS Move API in the sensor fusion step to filter the resulting position of the sensor fusion process for more stable position tracking.

The One Euro Filter is a “first order low-pass filter with an adaptive cutoff frequency”. Adaptive means that at low speeds, a low cutoff is used to reduce the jitter (at low speeds, lag is usually not so much of a problem) and as the speed increases, the cutoff frequency is increased as well to reduce Lag (at high speeds, lag is usually more of a problem than the - relatively small - noise). Another advantage of the One Euro Filter compared to more complicated filter

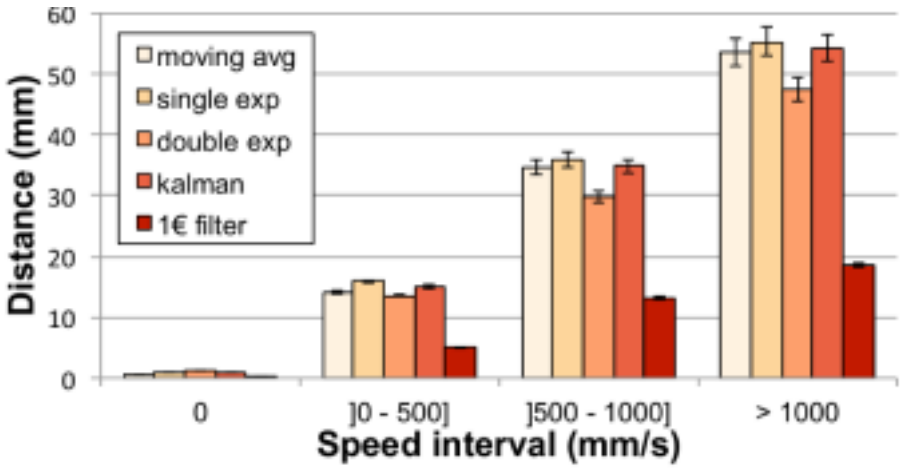


Figure 2.10: Distance from filtered values to real values (smaller is better) (from [4])

algorithms like Kalman filters is that it is easy to implement, it uses very few resources and with only two parameters, it is easy to tune. The One Euro Filter can be used with variable sampling rates (some filter algorithms depend on a fixed sampling rate).

Alternatives to the One Euro Filter would be a simple moving average or a simple exponential filter - however, they introduce either too much lag (the moving average has a higher lag than the exponential filter) or cannot smoothen the jitter (depending on the tuning parameters used). A much more expensive (in terms of implementation and processing power needed) method would be Kalman filters, but process and measurement noises have to be known or determined empirically.

Noisy signals in the PS Move system are introduced by magnetic fields, heat (temperature differences), resolution limits (of the sensors as well as the camera), large scaling factors (to amplify small movements on a big screen) and hand tremor (involunatary movements of the hand). The noise affects the accuracy of the tracking (an offset might be introduced) and the precision of the tracking (jitter).

The problem with currently-available advanced filter algorithms, as stated in [4] is: "[...] implementing and tuning a filter to minimize both jitter and lag is challenging, especially with little or no background in signal processing."

To tune the two parameters of the One Euro Filter, one can start with known values and then tune the first parameter until jitter is removed. With the first parameter fixed, the second parameter can be tuned to minimize lag. In comparison with single-parameter filters such as the simple exponential filter, it is easier to get a good reduction of both lag and jitter rather than just deciding which of the two problems to favor.

A comparison of filtering Desktop mouse interaction with artificially introduced noise with different algorithms is shown in figure 2.10. The high distance differences for the other filters are mostly related to lag, as all algorithm parameters have been tuned to reduce jitter.

2.5 Existing Approaches

This section presents existing approaches that deal with tracking the PS Move Motion Controller in 3D space and compares them with the PS Move API.

Move.me

Sony has released a solution called "Move.me" [30] for developers who want to use a Playstation 3 system together with the PS Eye camera and the PS Move Motion Controller. This solution costs USD 100 and requires a PS3 system, a Playstation Network account, a TV or HDMI display to hook up the PS3 system and network equipment in addition to the development machine.

Compared to the PS Move API developed in this thesis, Move.me lacks the portability, cost efficiency and flexibility: Our library can run on a mobile device and communicate directly via Bluetooth, the minimum setup consists of a controller and a mobile computer and it is able to use any camera input that is supported by OpenCV or that can grab frames in a format that can be passed to OpenCV.

For example, it is possible to use a high definition camera with a higher resolution than the 640x480 pixels provided by the PS Eye camera. In addition, using a networked setup, it is possible to use more than 7 controllers in one application by networking together multiple host computers.

Sony demonstrates that tracking is possible in principle – their implementation utilizes the Cell Broadband Engine on the Playstation 3.

CHAPTER 3

Methodology

This chapter describes the high-level concepts and design of the PS Move API. First, the hardware is described in section 3.1. Then, a list of features is presented in section 3.2. After that the different processes (pairing in section 3.3, tracking in section 3.4) as well as the end user interaction (in section 3.7) are described. The tracking algorithm used to detect and track the controller in the camera image is described in section 3.5. At the end of this chapter, basic design decisions (section 3.8) are listed that are used throughout the library design and this thesis.

3.1 Hardware

This section describes the basic hardware that is used in the PS Move API setup: The PS Move Motion Controller and the PS Eye camera. While other cameras can be used, the PS Move API has been tested with and developed for the PS Eye camera, because it's readily available, affordable and has a good frame rate while allowing total control over the exposure.

Playstation Move Motion Controller

The central piece of hardware used in this project is the Playstation Move Motion Controller (figure 3.1). It is a cylindric game controller with a soft white sphere on top of it. The sphere

Figure 3.1: Playstation Move Motion Controller

is lit from the inside by an RGB LED, allowing for many different colors. The body of the controller contains 9 buttons:

- Four buttons (Square, Triangle, Cross, Circle) on the front, which are color-coded as pink, green, blue and red
- Two buttons (Select on the left, Start on the right) on the sides, which have their names written on them
- One big Move button sitting at the top center of the front, with the PS Move Logo on it
- One small PS button sitting at the middle center of the front, with the PlayStation Logo on it
- One analog Trigger button at the back, with the letter "T" on it

With the exception of the analog Trigger, all buttons are digital. The Trigger can be used as a digital button or as an analog 8-bit button. The PS button is special in that it is used as power button to switch on the controller. Pressing the PS button for about 10 seconds will turn off the controller - this is built into the controller's hardware, and cannot be prevented by software.

On its bottom, the controller has ports for connections:

- A standard Mini USB socket used for pairing, USB connections and for charging from USB host devices. This is a standard connector, and can be used with any standard Mini USB cable - no special cable is necessary.
- Charging connectors for use with Sony's charging dock accessory
- An expansion port for connecting the controller to accessories (e.g. gun attachments, etc..)

The controller has three types of built-in sensors - these are connected to the STM32F103VBT6 microcontroller, and cannot be accessed directly. The communication happens from the microcontroller via a Cambridge Silicon Radio BC4RE Bluetooth module.

The following sensor chips are used in the PS Move Motion Controller (this information has been reverse-engineered directly from the hardware by Kenn Sebesta in [29]):

- **Accelerometer:** Kionix KXSC4 10227 2410 three-axis accelerometer
- **Gyroscope:** Two chips, unknown origin (a two-axis gyroscope for the X and Y axes, identified by [29] as STM LPR425AL and a single-axis Y5250H 2029 K8QEZ gyroscope for the Z axis)
- **Magnetometer:** AKM AK8974 magnetic compass

The controller also contains a temperature sensor (make and model unknown). The temperature value is included in the input report, but is not yet used in the sensor calibration algorithm (the raw value can be obtained via the PS Move API, though).

In the input report, the accelerometer and gyroscope values are reported as signed 16-bit values, the magnetometer fields are 12-bit signed values.

The PS Move Motion Controller sends up to 85 updates per second via Bluetooth on a Mac OS X host, and up to 60 updates per second on a Linux host using the same hardware (it is not yet determined why the two operating systems differ so dramatically in update rate). Each update contains two readings for the accelerometer and gyroscope, giving an effective maximum update rate of 170 Hz for the accelerometer and gyroscope, and 85 Hz for the magnetometer and buttons.

Playstation Eye

For tracking the motion controller in 3D space, the Playstation Eye camera (figure 3.2) is used. Any other camera could be used as long as it has a decent enough resolution, frame rate and exposure control.

The PS Eye camera is a very good choice as a camera, because of its low cost, its USB 2.0 connectivity and the frame rate of 60 FPS at 640x480 and 120 FPS at 320x240. In addition to having a good frame rate, the PS Eye camera also allows access to the exposure duration that the camera uses. Turning off auto-exposure and turning down the exposure duration is important for getting good tracking results and avoiding problems caused by motion blur on other cameras.

The camera has been released in 2007, independent of the Playstation Move Motion Controller, but it is part of the Playstation Move system on the Playstation 3 (a PS Eye is required for Move input on the PS3, but the PS Eye can be used for other applications without the Move Motion Controller).

Another advantage of the PS Eye is that it's supported out of the box on Linux since kernel version 2.6.29[1]. A proprietary driver[2] for Windows is available from Code Laboratories. On Mac OS X, the camera is supported by the open source macam[3] driver, but it does not work together with OpenCV in recent (64-bit) versions of Mac OS X.

An alternative to macam on OS X would be to create a userspace interface using Jim Paris' original C code[4] for interfacing with the camera using libusb.

Because of its price, availability and technical specifications, the PS Eye camera will be used for the reference implementation and for testing the tracker library. Where it makes sense, the library should prefer the PS Eye camera over other cameras connected to the system (e.g. a built-in webcam in a notebook computer). Of course, other cameras such as HD cameras or built-in webcams are also supported by the library, but the tracking performance might not be as good as with the PS Eye camera, with which the library has been tested.

[1] Linux 2.6 commit fbb4c6d20f29f2b10daad31cc6238d91f93d70d4, November 2008

[2] http://codelaboratories.com/products/eye/driver/, retrieved 2012-09-25

[3] http://webcam-osx.sf.net/, retrieved 2012-09-25

[4] https://jim.sh/svn/jim/devl/playstation/ps3/eye/test/eye.c, retrieved 2012-09-25

Figure 3.2: Playstation Eye USB 2.0 Camera

The PS Eye camera uses an Omnivision OV7725 $6\mu m$ VGA sensor [5] with an image area of $3984\mu m$ x $2952\mu m$, a signal-to-noise ratio of $50dB$ and a dynamic range of $60dB$ (source: [25]).

3.2 Features Overview

The PS Move API provides the following core features:

- Pairing the PS Move Motion Controller with the host computer
- Setting LEDs and the rumble motor intensity
- Reading button states and the state of the analog trigger
- Reading raw sensor values (accelerometer, gyroscope, magnetometer)
- Reading the factory-supplied calibration data via USB during pairing
- Reading miscellaneous device state information (temperature, battery charge level, etc..)

The following features utilize calibration data and the orientation algorithm from [21][6]:

- Converting raw sensor values to calibrated sensor values
- Getting the device orientation in quaternion representation

[5] http://image-sensors-world.blogspot.com/2010/10/omnivision-vga-sensor-inside-sony-eye.html, retrieved 2012-12-01

[6] as the orientation algorithm reference implementation that is used in this project is licensed under the GNU General Public License, some users of the library might want to build the PS Move API without this algorithm

The following camera tracker features are available when building with OpenCV[7]:

- Getting the X/Y position and the sphere radius of tracked controllers
- Mirroring the camera image horizontally
- Auto-updating LEDs of connected controllers
- Deinterlacing of camera video input
- Retrieving the camera image as 24-bit RGB image

The following sensor fusion features are available when both the orientation algorithm and the vision tracker are used:

- Getting the projection matrix for 3D rendering on top of the camera image
- Getting the model-view matrix for 3D rendering with the controller's sphere center as the coordinate origin
- Getting the 3D position of the controller

3.3 Pairing Process

This section describes the pairing process from a high-level point of view. Details about the implementation on different platforms can be found in section 4.8.

In order for the Motion Controller to be able to communicate with the host computer, it has to be "paired". In traditional Bluetooth applications, this happens by pressing a button on the device to make it discoverable, and then pairing the device with the host by entering either a fixed PIN (for devices without a way of entering data, e.g. headsets, mice, ...) or by entering the same PIN on both devices (for mobile phones, tablets, etc...). More recently, pairing via Bluetooth could also be established via the same PIN-based technique, but instead of entering the same PIN twice, the devices auto-generated a common PIN and display it - the user only had to confirm that the PIN displayed is the same on both devices.

The official Playstation 3 controller (SixAxis, DualShock 3) and subsequently the PS Move Motion Controller use a different method of pairing. In order to make the process less tedious for gamers, pairing happens over USB. When the controller is connected via USB, the PS3 sends its Bluetooth address via a HID feature report and the controller stores the Bluetooth address in non-volatile memory. When the controller is turned on, it connects to the Bluetooth address in memory, effectively establishing a connection with the desired host.

[7] some users might want to avoid the OpenCV dependency - the core library can be built without Tracker support

Pairing with the PS3

The pairing process happens automatically on the PS3 system, and users usually do not realize that pairing takes place, as simply connecting a controller for charging will initialize the pairing process:

1. Connect the controller to the PS3 system (the PS3 must be switched on)
2. Wait for about 2 seconds
3. Disconnect the controller from the PS3 system
4. Press the PS button on the controller

Pairing with PCs (Linux, Mac OS X, Windows)

The steps for pairing the Playstation Move Motion Controller with a PC are:

1. Connect the controller to the host computer via USB
2. The host computer opens the controller as HID device
3. The host computer reads the host address and the address of the controller (feature report 0x04)
4. The host computer determines its own Bluetooth address (this part is operating system-dependent) - alternatively, the user might want to specify a custom Bluetooth address to pair the controller to (this is especially useful for pairing with mobile devices that do not have USB host mode capabilities)
5. The host computer compares the desired host address with the currently-set host address (to avoid unnecessary writes to the controller's memory)
6. In case the addresses differ, the host computer updates the host address in the controller (feature report 0x05)
7. The host computer can now optionally add an entry for the controller to the operating system's Bluetooth stack in order for the Bluetooth stack to accept connections from the controller - this essentially "fakes" the traditional PIN-based pairing process

3.4 Tracking Process

To track the controller with 6 degrees of freedom, the data from the camera image and the inertial sensors must be combined, as can be seen in figure 3.3.

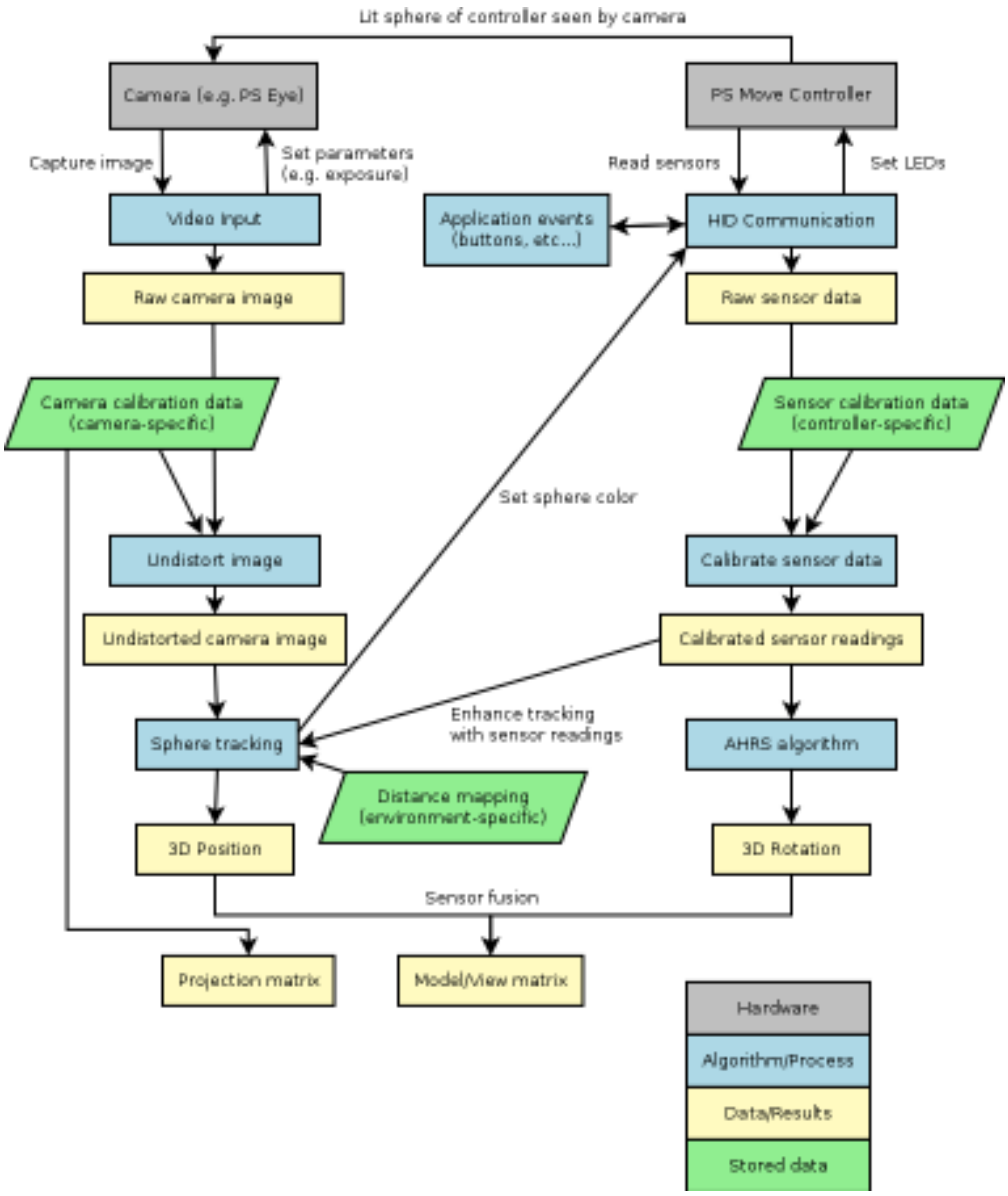


Figure 3.3: How the Tracking Process works in PS Move API

Calibration data

To transform raw data into a format usable by the library and to calculate the right distance from the camera, three types of calibration data must be used:

1. **Camera calibration:** This calibration must be done once per camera type. It provides the necessary parameters for reversing the lens distortion as described in [33]. Users can either choose to create their own calibration data or use the camera in uncalibrated mode,

which might result in inaccurate tracking results. These inaccuracies might be acceptable in certain use cases - calibration is not enforced.

2. **Sensor calibration:** Each controller's inertial sensors have different output ranges. In the factory, the controller is calibrated and the calibration data is saved in the controller's nonvolatile memory. It can be retrieved via USB HID. The PS Move API takes care of saving the calibration data as file to the filesystem while pairing takes place. The calibration data is used when the controller is connected via Bluetooth, and mapped using the Bluetooth address of the controller.

3. **Distance mapping:** In order to provide more accurate distance measurements, the user can calibrate the distance values (mapping of sphere radius to distance values) and save this mapping in a special file. If available, this distance mapping will be used instead of the default mapping, which will result in better depth perception. In some cases it might be necessary to carry out a new distance mapping calibration if the lighting conditions change - this is also why the distance mapping is refered to as "environment-specific" in figure 3.3.

Sphere Tracking

Sphere tracking can be done in two ways:

1. **Edge-detection and 2D Hough transforms** – This method is utilized by [15] and gives good results even if the circle isn't highly visible in the camera image.

2. **Color-based detection using contours** – This method has been used by [3] and had very good results using hue-based filtering in HSV colorspace.

As the PS Move is basically a colored sphere, using the hue-based detection using contours in HSV colorspace has advantages for this implementation. While 2D Hough transforms can be used with a color filter before the edge detection as well, the contour-based detection algorithm should provide better results and also work in cases where the controller's body (which itself has a circle-shaped edge) obscures parts of the glowing RGB LED sphere.

The orientation tracking method described in [3] is not used in this work, as orientation tracking is done using inertial sensors built into the controller itself. This makes the computer vision implementation simpler and makes good use of the high inertial sensor sampling frequency.

Advantages of using the RGB LED

In all presented sphere tracking algorithms, the tracking algorithm had no control over the color of tracked objects. In case of the PS Move Motion Controller, the color of the sphere can be set via Bluetooth, and (ignoring any delay from the Bluetooth traffic and the delay of grabbing a frame from the camera) immediately observed in the camera image.

This has two nice side-effects:

- **Determining the position** – The position of the controller can not only be determined by using a color filter. Switching the LEDs on and off and taking snapshots, then computing the difference between these snapshots allows us to determine the position of the controller in the camera picture without manual user intervention.
- **Selecting a suitable color** – By analyzing the environment, a suitable color can be picked. Colors that are already present in the camera image prior to tracking can be skipped and avoided when choosing a suitable color to assign to the RGB LEDs.

Again, these properties of the hardware in use allow for some simplifications and improvements to the tracking that were not possible with static spheres.

Considerations for the Camera Calibration Method

For the PS Move API, the type of camera calibration that is needed should deal with lens distortion, as it is important to have an undistorted image for accurate 3D position calculation (based on the radius of the tracked sphere).

The approach in [33] seems simple enough to be carried out by developers wanting to use new cameras with the library. Also, a similar implementation of this approach is shipped with OpenCV, and so can be readily used on a wide variety of machines and cameras.

Camera calibration using inertial sensors as in [20] might be useful for some future improvements, but might not immediately apply to the problem at hand. Its use of inertial sensors to augment the vision data is interesting, and provides some inspiration for the sensor fusion part of the PS Move API.

Importance of the Orientation Algorithm for Sensor Fusion

A good orientation filter (e.g. the algorithm in [21]) is a prerequisite for estimating the controller position in the camera picture when visual tracking is not available for short amounts of time (e.g. because the sphere is fully obscured during tracking). Given the orientation, a movement registered by the accelerometer (relative to the controller's frame) can be rotated into the world's frame and from there to the camera's projection.

3.5 Tracking Algorithms

This section describes all algorithms used during the visual tracking process. For an end-user perspective of the tracking, see section 3.7, for a high-level overview of the tracking process (including inertial sensor reading and sensor fusion), see section 3.4.

Assignment of Colors to Controllers

In the current implementation, a list of known "good" colors is hardcoded in the PS Move API. As controllers are added to the Tracker object, the Tracker assigns a unique color to each

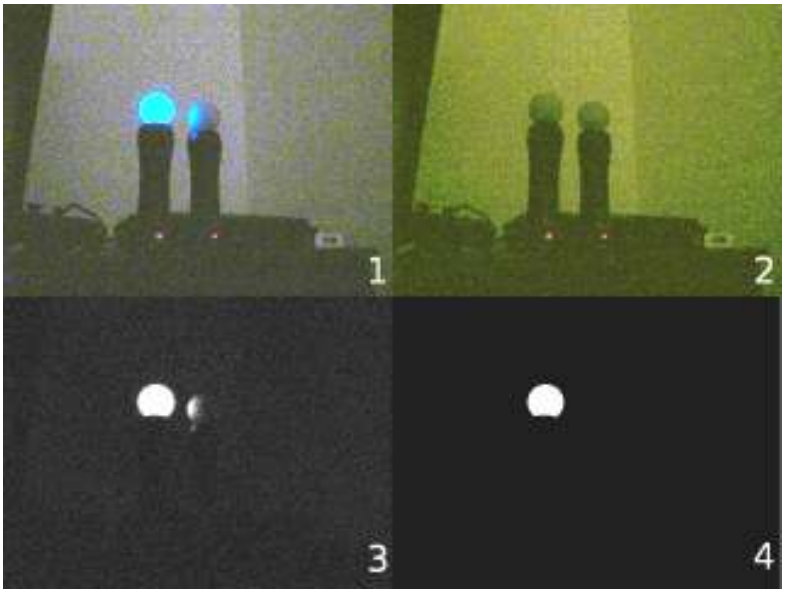


Figure 3.4: Blinking calibration without background movement: Image of the controller switched on (A-frame, 1), image of the controller switched off (B-frame, 2), difference image of 1 and 2 (3) and thresholded difference image (C-frame, 4). The color of the controller can be determined by using the C-frame (4) as a mask for the A-frame (1) and calculating the average hue (in the implementation, the mask is an AND-combination of all captured C-frames to improve reliability).

controller. For advanced use cases, the API provides a function to specify the color for each controller - in that case, tracking results can be worse than with auto-assigned colors.

Future improvements of this algorithm could take the current camera image (environment, background) into account and avoid colors that already appear in the environment. In the case of using the PS Eye with a low exposure, this is usually not necessary, because only the (actively-lit) spheres are visible in the camera image.

Blinking Color Calibration

To determine a mapping from the assigned color of the controller to the color of the controller in the camera image (which, depending on the camera, environment and exposure settings, can be quite different), the following algorithm is used:

1. Switch the controller's RGB LEDs to the controller's assigned color
2. Wait 100 ms (capture and discard frames from the camera during that time)
3. Capture a frame from the camera and store it (A-frame)
4. Switch the controllers' RGB LEDs off

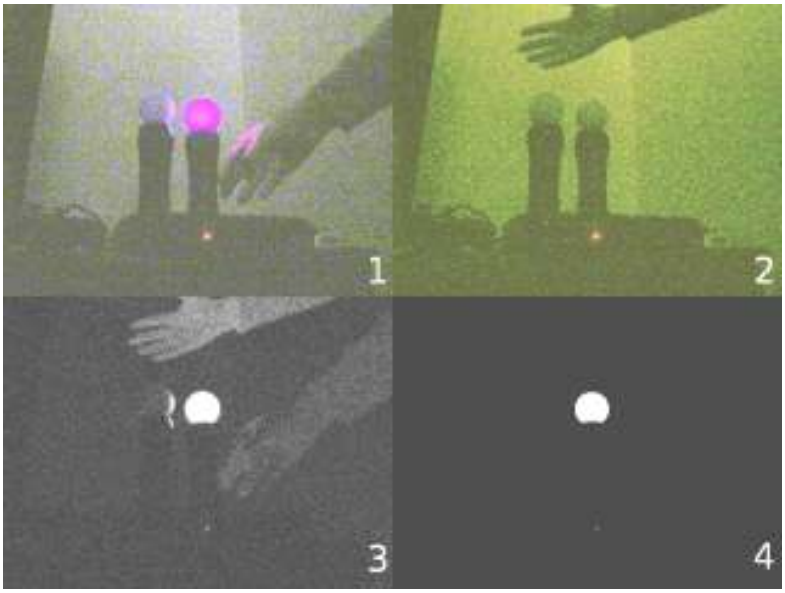


Figure 3.5: Blinking calibration with background movement: The hand position changes from frame 1 to frame 2. This is visible as two shadows in the difference image 3. The thresholding (image 4) usually removes these errors.

5. Wait 100 ms (capture and discard, as above)

6. Capture a frame from the camera and store it (B-frame)

7. Generate a difference image (C-frame) of the A-frame and B-frame captured in steps 3 and 6 by creating the difference image and then thresholding and eroding/dilating the difference image to remove any noise in the difference image

8. Repeat steps 1-7 for 3 more times, which results in 4 A-frames and 4 C-frames

An example of the blinking calibration can be seen in figure 3.4, and an example how this algorithm deals with slight changes in the background between the A-frame and B-frame can be seen in figure 3.5.

The resulting C-frames (difference images) are then added together into a single image (the mask) using the AND operation. The pixels in the mask now describe the areas in all 4 A-frames that changed during blinking. From this mask, only the biggest contour is taken and the average color in that contour is determined from the first A-frame. The resulting color is the color of the controller in the camera image. Some additional checks are carried out that make sure that the hue of the color in the camera is in the same range as the hue of the assigned color, which makes the algorithm more robust against false positives.

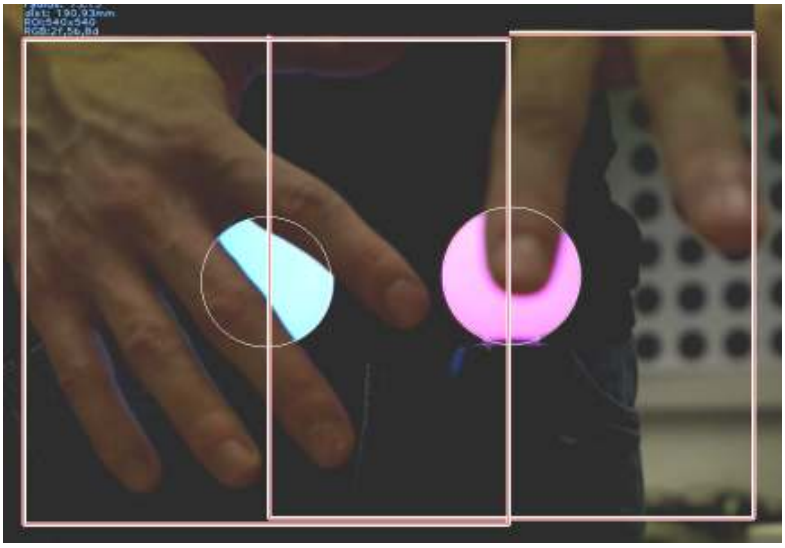


Figure 3.6: The sphere sizes and centers are properly calculated even though they are only partially visible (the red/white squares show the region-of-interest for each tracked sphere, the white border around the spheres is based on the center and size of the detected sphere)

Sphere Size and Center Calculation

Determining the sphere size is important for proper depth detection. Radius detection should also work in cases where the sphere is only partially visible (see figure 3.6). The following algorithm implements the sphere size detection:

1. Start with the biggest detected contour in the image given the color of the controller determined from the Blinking Color Calibration algorithm
2. Set the current maximum distance to 0
3. For each pixel in the contour, calculate the distance to all other pixels in the contour
    a) If the distance is greater than the current maximum distance, store the two pixel coordinates and save the distance as the new maximum distance
    b) If the distance is less than the current maximum distance, continue
4. The radius of the sphere will be half the maximum distance
5. The center will be the average of the two pixel coordinates stored in step 3.a (the center point of the line between the two pixels with the greatest distance)

This algorithm works because a sphere always appears as a perfect circle in the camera image (assuming a calibrated camera), and the upper limit for the distance of any two points in the circle is always the diameter of the circle. In the extreme case, only two opposite pixels

on the edge of the circle need to be detected for the algorithm to still function correctly. Only when no opposite pixels are visible anymore will the algorithm fail and report a smaller radius. In particular, the center of the sphere does not have to be visible, as long as the diameter can be determined from the sphere border.

To avoid unnecessary calculations, the "current maximum distance" is the squared distance. Only at the end of the algorithm will the square root be calculated. To smoothen the detected radiuses and centers, a filter is used in the algorithm after the sphere size and position detection step.

Region of Interest Size Calculation

In the tracking algorithm, the Region of Interest (ROI) is a square sub-image of the camera image that is centered on the current sphere image. The current implementation uses 4 differently-sized ROIs. Using ROIs instead of analyzing the whole image on each frame gives a performance boost, and also avoids wrong blobs to be mis-detected as the sphere. The ROI sizes are determined as follows:

1. If the environment variable **PSMOVE_TRACKER_ROI_SIZE** is set, the biggest ROI size will be the integer value of the environment variable
2. Otherwise, the biggest ROI size will be half of the shorter side of the camera image: $roi\_size_0 = 0.5 \cdot min(width, height)$
3. Subsequent ROI sizes will be 70% smaller than the size of each preceding ROI: $roi\_size_{n+1} = 0.7 \cdot roi\_size_n$

For 4 ROI sizes and a 640x480 image, the ROI sizes will thus be calculated as:

- ROI 0: $roi\_size_0 = 0.5 \cdot min(640, 480) = 240$
- ROI 1: $roi\_size_1 = 0.7 \cdot roi\_size_0 = 168$
- ROI 2: $roi\_size_2 = 0.7 \cdot roi\_size_1 = 117$
- ROI 3: $roi\_size_3 = 0.7 \cdot roi\_size_2 = 81$

State transitions can only happen to adjacent ROI sizes. ROI 0 is always used at the beginning (and when tracking is lost) - when the sphere size becomes smaller, smaller ROIs are used to better track the sphere.

Region of Interest Panning

The ROI for each controller will be centered around the contour detected for the controller. If the contour happens to be outside of the ROI (or at the edge of a ROI), the ROI will be moved in the direction of the contour for best tracking results. If panning does not help, the next bigger ROI is used and tracking will be retried. If after switching (and panning) to the biggest ROI no contour is found, the recovery procedure described below is started.

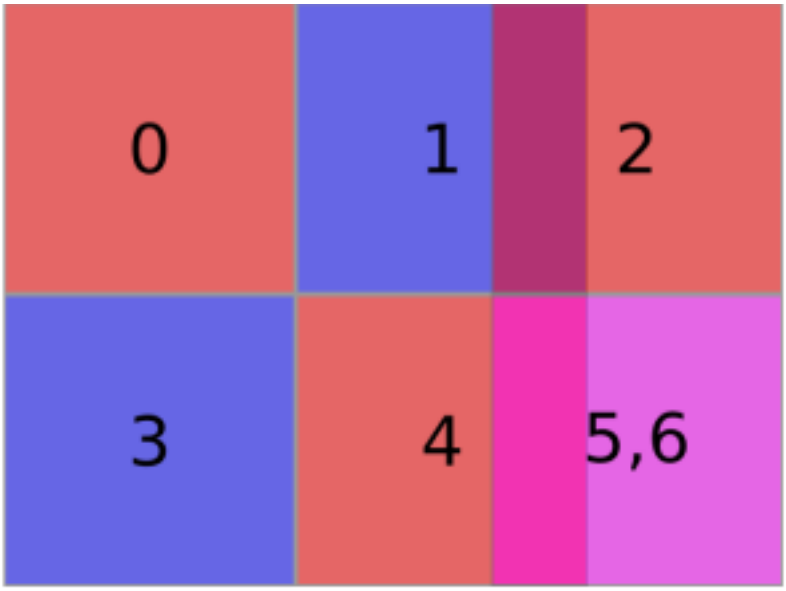


Figure 3.7: Tiling for a 640x480 camera image ($roi\_size_0 = 240$): Red (0, 2, 4, 6) and blue (1, 3, 5) tiles form separate groups, the last column (2, 5, 6) overlaps with the second-to-last column (1, 4) and the last tile (5) is duplicated (6) so that the algorithm can swap between the two groups without any additional checks using: $new\_tile\_id = (tile\_id + 2)\ mod\ (tile\_count)$

Recovery after Tracking Loss

The recovery algorithm takes places at the beginning of tracking and every time the controller is lost. At the start of tracking, after the ROI size has been determined, the image area is split up into n tiles:

1. The image is divided into columns and rows by tiling the biggest ROI size - the last column (the last row) might overlap with the second-to-last column (second-to-last row) if the image width (height) is not a multiple of the ROI size

2. Tiles are numbered column-first, starting from zero

3. If the tile count ends up even (if rows or colums (or both) are even) it is increased by one (by duplicating the last tile) - this is necessary in order for the algorithm to alternate between odd and even fields

4. The resulting grid of tiles will be split into odd and even fields, depending on the tile index

An example of how tiling is accomplished for a 640x480 image with a ROI size of 240 is shown in figure 3.7. The recovery procedure happens for each controller separately (so one controller can be tracked while the other controller is in recovery mode):

1. If the controller is not tracked, its search tile number will initially be set to 0 (corresponding to the top left tile) - this is the initial value of the search tile number, and it will also be set to 0 every time the controller is found

2. The biggest ROI will be placed at the position of the search tile, and the controller's color will be searched in this area

    a) If the controller color is found, the ROI will be centered on the controller image, and the controller will be marked as found

    b) If the controller color is not found, the search tile number will be updated: $new\_tile\_id = (tile\_id + 2)\ mod\ (tile\_count)$

3. The new search tile number will only be used in the next frame, so only one tile search is carried out per captured frame - this avoids excessive blocking of the main thread during recovery

Because only one tile is searched per frame during recovery, the tracker algorithm has roughly the same processing requirements during recovery as it has during tracking. Also, the worst-case for recovering after tracking loss (using the example from figure 3.7) would be 6 frames:

- Assuming the controller is completely inside tile 3
- The tiling algorithm starts at tiles 0, 2, 4, 6 (4 frames)
- After wrapping, tiles 1, 3 are visited (2 frames)
- In tile 3, the controller is found - tracking has recovered

In the general case, however, the controller usually overlaps with at least two tiles (e.g. if it is in the image center and of suitable size, it is contained in tiles 1, 2, 4, 5 and 6). In that case, recovery can happen in only 2 frames:

- Assuming the controller overlaps with tiles 1, 2, 4, 5 and 6
- The tiling algorithm starts at tiles 0 and 2 (2 frames)
- In tile 2, the controller is found - tracking has recovered

This algorithm finds the controller faster than linearly searching for the controller from top left to bottom right. More advanced search patterns could be used (e.g. an archimedean spiral from the center of the image), but in my experiments, I found the current solution easy to implement, efficient and quick to converge in the average case.

One improvement that could be implemented easily would be to not duplicate the last tile, but instead add e.g. a centered tile at the end of the list.

Figure 3.8: The sphere radius is 64.58 pixels at 20 cm distance from the camera (left) and 6.57 pixels at 200 cm distance from the camera (right) using a PS Eye camera.

3.6 Distance Function

For some use cases, it is desirable to not only get the radius of the sphere in the camera, but also a real-world distance measurement. This distance has to be calculated from the sphere radius.

The tracking algorithm returns the size (radius) of the sphere in the camera image. The closer the sphere comes to the camera, the bigger it will appear in the image, and the farther away from the camera, the smaller it will appear in the image. The mapping from radius (or even diameter) to distance is not linear or quadratic - the mapping function has to be determined experimentally by using curve fitting on experimental data.

Empirical Measurements

The tool **distance_calibration** (included in the PS Move API source) has been used to obtain the radiuses of the controller for different distances. The desired distance is shown on-screen, and the user has to place the controller at the displayed distance and press a button to confirm. The tool takes 50 measurements at 5 cm increments, starting at 20 cm. The results are saved as "distance.csv" (comma-separated value text file) and as screenshots "distance_*N*.jpg" (where *N* is the distance in cm, e.g. 020 for 20 cm). Two example screenshots at 20 cm and 200 cm saved by the tool can be seen in figure 3.8, the colored blobs below the controller are reflections of the controller from the tape measure and the floor.

The resulting pixel values from the experimental measurements can be seen in figure 3.9. The distance is measured from the red ring of the PS Eye camera.

Curve Fitting and Function Selection

To determine the most suitable function for mapping radius values to distance values, the open source curve-fitting tool Fityk [31] was used. The most suitable function for the distance map-

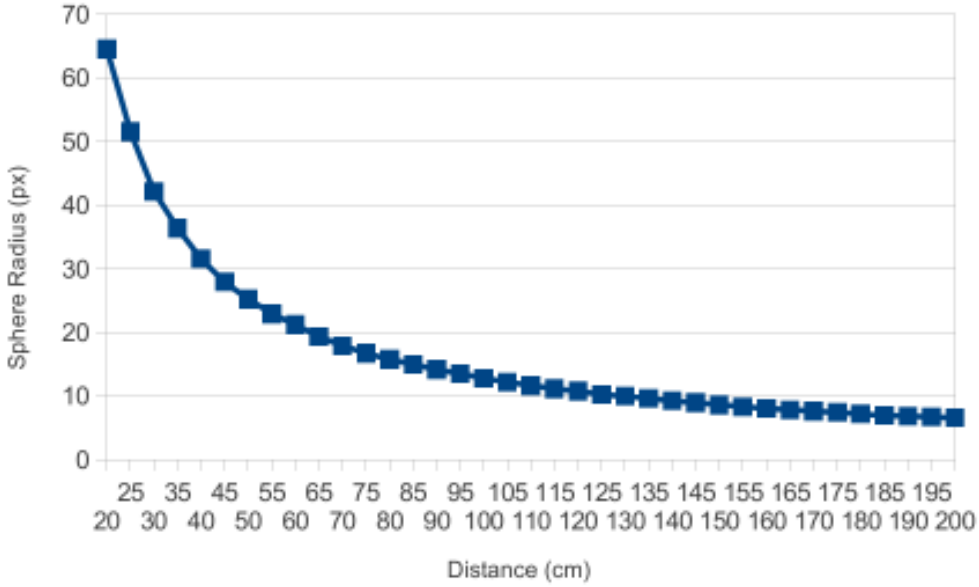


Figure 3.9: Measured sphere radiuses (in pixels) from 20 cm - 200 cm, measured by the PS Move API **distance_calibration** tool using a PS Eye camera in wide angle view with a low exposure (Exposure_LOW setting) and a PS Move using magenta as the sphere color.

ping data set was found to be a Pearson VII distribution, which is called "Pearson7" in Fityk. In the Fityk documentation[8], the function is defined as:

$$y(x) = height \cdot \left(1 + \left(\frac{x-center}{hwhm}\right)^2 \cdot \left(2^{\frac{1}{shape}} - 1\right)\right)^{-shape}$$

The distribution has four configurable parameters: $height$, $center$, $hwhm$ and $shape$.

Parameters for the PS Eye Camera

Loading the "calibration.csv" file generated by the **distance_calibration** tool into Fityk, and fitting a "Pearson7" function to the data (see figure 3.10) gives the parameters for the PS Eye camera that are used in the PS Move API:

$height = 517.281$
$center = 1.297338$
$hwhm = 3.752844$
$shape = 0.4762335$

[8] http://fityk.nieto.pl/model.html, retrieved 2012-12-01

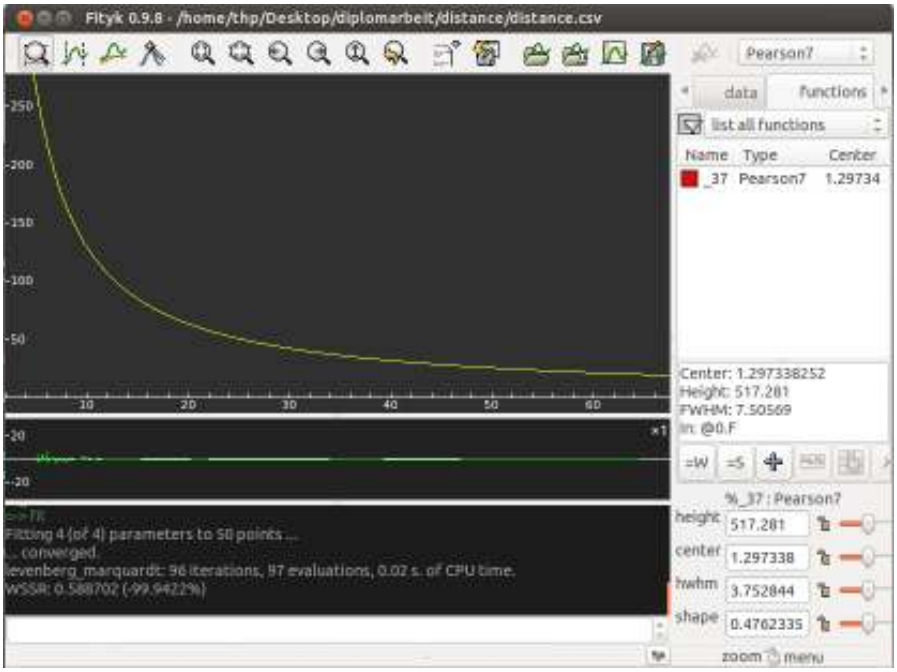


Figure 3.10: Curve fitting of distance data using the Pearson VII distribution in Fityk.

Parameters for Other Cameras

These parameters have been verified to work well with a PS Eye camera in wide angle mode, but will not give the right mapping for other cameras. The distance calibration tool and the Fityk application are freely available and the process is documented here, so users can use the same process (distance calibration followed by curve fitting of a Pearson VII distribution function in Fityk) to obtain the 4 required function parameters. The PS Move API provides a function to set the four parameters for custom camera setups. By default, the above parameters are used for distance calculation, so PS Eye users can use the distance mapping function out of the box without any extra calibration steps.

3.7 End User Interaction

This section describes the way the end user sees and interacts (indirectly) with the library. It describes the steps necessary to calibrate the camera (technical details in [33]), to pair the controller (technical details in section 3.3), to calibrate the magnetometer and finally to start the tracking process (details in sections 3.4 and 3.5).

Calibrating the Camera

Calibrating the camera is important if the user requires high-quality tracking accuracy or if the camera in use has noticeable lens distortion (which is true for almost all consumer cameras).

This step is **optional** - if it is not carried out by the end user, the camera image will be processed as-is, otherwise the undistortion step will be done for every frame captured from the camera by the library.

The PS Move API includes an utility **tracker_camera_calibration** (source code in examples/c/tracker_camera_calibration.c) that users can use to calibrate the camera. For this step, the user must print out the chess board pattern found in **contrib/pattern.png** in the PS Move API source:

1. Print the pattern from **contrib/pattern.png**
2. Start **tracker_camera_calibration**
3. Place the printed pattern in front of the camera
4. Move / tilt the pattern until it gets recognized by the application, and a colorful grid gets drawn on top of the pattern
5. Press the space bar to capture the image
6. Repeat the previous two steps 9 more times to capture a total of 10 images that will be used for calibration - make sure to always place the pattern in a different position/angle from the camera
7. The calibration data will be calculated, and a preview of the undistorted camera image will be shown
8. If the output looks good, the user can close the application and start using the PS Move API
9. If the output looks wrong, the user must re-do the calibration steps and capture 10 more images until the calibrated picture looks good

It is possible to carry out the camera calibration for specific camera models and lens combinations (if the camera has exchangeable lenses). In most tests, the PS Eye camera gave acceptable tracking results even without using the camera calibration described here.

As the camera calibration is independent of the PS Move Motion Controller, this calibration can be carried out without any PS Move Motion Controller paired or connected.

Pairing Process

The following steps have to be carried out by the end user:

1. Connect the controller to the host computer via USB.
2. Run the “psmovepair” utility and wait until it has finished.
3. Disconnect the controller and press the PS button.

While pairing should be hassle-free and a simple three-step process as outlined above, depending on the operating system and version used there might be some caveats to the pairing process:

- On Mac OS X 10.7 (version 10.7.3 and up), pairing might not work reliably.
- On Mac OS X 10.8, initial pairing will take about 42 seconds while the library waits for the OS X "blued" process to shutdown. The user might have to enter the system password. Subsequent pairing will be instantaneous.
- On Linux, pairing requires root permissions. Restarting the Bluetooth Daemon is currently only implemented for Linux distributions using upstart (this includes Ubuntu Linux and Debian).
- On Mac OS X 10.7.3 and up (including OS X 10.8) and on Linux, the Bluetooth system will be deactivated during the initial pairing when the controller's Bluetooth address needs to be inserted into the system's Bluetooth stack. This only happens once for each controller and OS installation, even if the controller is paired with another machine and then paired again with that installation.
- On Windows, a trial-and-error procedure (see section 4.8) has to be carried out to get pairing to work. It usually works after several tries and has been tested on Windows 7 and on Windows 8 installations.

Background information on how pairing is implemented and the reasons for the limitations listed above can be found in section 4.8.

Calibrating the Magnetometer

To calibrate the raw magnetometer ranges, the controller to be calibrated must be connected via Bluetooth. The calibration is environment-dependent, so users might want to carry out the calibration process every time the setup is moved to a different location.

The magnetometer readings are only used as a 3D vector that points to the magnetic north pole in [21], so the length of the vector is not important, only the direction. As such, the magnetometer readings on each axis are mapped to a value range of $(-1, +1)$, including both endpoints.

In principle, the magnetometer can be calibrated by pointing every axis exactly towards and exactly away from the direction of the magnetic north pole. As this requires special equipment in practice, this solution simply asks the user to rotate the controller in all directions, and a threshold value is used to determine when the observed values are "good enough" (i.e. the value range is big enough to calculate a good direction vector from it). The steps are:

1. Connect one or more controllers via Bluetooth
2. Start the **magnetometer_calibration** tool

3. For each controller, the tool will display on-screen instructions on how to calibrate the controller:

    a) The controller starts out with a red sphere

    b) The user must rotate the controller into all directions, so that the minimum and maximum raw values for each axis can be determined

    c) As calibration progresses, the progress is shown on-screen, and the controller's sphere slowly fades to green

    d) When calibration is done, the user must press the Move button on the controller to finish the process

The magnetometer calibration will be updated dynamically at runtime. As such, the calibration is not mandatory, but a warning will be printed if no calibration data was found. If no calibration is available, the value range will be adjusted dynamically at runtime, as new minimum and maximum values are observed.

Tracking Process

This section assumes that the pairing process has already been carried out, and that the controller connects to the host machine when the PS button is pressed. See the previous section on how the pairing process is carried out.

- Press the PS button on the controller - the red LED starts blinking
- Wait until the red LED on the controller stops blinking and becomes lit (if the LED switches off after blinking, the connection was not successful and pairing might need to be carried out again)
- Start any of the example applications (e.g. **test_tracker**) and hold the controller in front of the camera
- The controller will blink a few times[9] - do not move the controller during that time, or the calibration will not succeed and start again
- After the blinking calibration, the sphere of the controller will be lit up and the tracking output will be shown on screen
- In case the tracking output is not satisfactory, start the application again to re-initialize the calibration

For some applications that utilize the magnetometer for orientation tracking, it might be necessary to carry out a magnetometer calibration first.

When using multiple controllers, connect all the controllers first and then start the example application - the applications usually determine the number of connected controllers at startup,

[9] technical details are described in section 3.5, "Blinking Color Calibration"

so any controller connected after startup is not used (the API provides features for re-initializing the hidapi and discovering newly-connected devices, in case of **moved** this happens automatically under Linux by listening for udev events).

In general, tracking works best with the exposure turned down as much as possible - this avoids motion blur and avoids any false positives from other light sources or colored objects in front of the camera. For situations and use cases where it makes sense to have a higher exposure (e.g. augmented reality applications where the user should be able to see herself in the camera image), the intensity of the controller will be turned down by the blinking calibration algorithm. Without dimming the LED sphere, the tracking algorithm might not be able to detect its color, because its brightness makes it appear as white color in the camera image.

3.8 Design Decisions

This section lists some of the design decisions and best practices that are used in the implementation of the PS Move API library as well as in this thesis.

Language

The implementation language chosen is C. It provides high performance at a low level that is suitable for writing code dealing with high sampling frequencies. Choosing C has the added benefit of allowing other languages to link to it, as most programming language runtimes provide some way of bindings C libraries.

Layering

The library will be split up into different layers (low-level, calibration, orientation, tracking). This eases maintenance and allows different developers to work on different modules during maintenance and development time. It also allows users of the library to choose to not use some layers, while still being able to use the rest of the library (e.g. for some use cases it might make sense to use only the lower-level APIs without using the vision tracking part).

Computer Vision

For tracking the controller in the camera image, OpenCV will be used, as it is widespread, portable and stable. Integrating with OpenCV will also allow other developers to use their existing OpenCV skills and apply them to the vision part of the library, should the need arise.

Orientation Tracking

An *Attitude Heading Reference System* is used to determine the orientation of the controller in 3D space by means of a rotation quaternion. This will allow this component of the library to be indepentent of the other code, and will also allow users to exchange the supplied AHRS algorithm with another one should the need arise.

Build System

CMake has been chosen as the build system of choice, as it provides all the necessary abstractions for doing cross-platform builds. While in theory other build systems might be used to build the library, the reference implementation and core utilities will make use of CMake.

Language Bindings

Bindings for higher-level languages are provided to make the library more accessible and allow beginner-level programmers to quickly develop solutions with the library.

API Documentation

API documentation is provided explaining the list of functions, the parameters and return values as well as all data types that are public. Usage examples should be provided directly in the documentation where the usage is not immediately clear from the pure parameter description.

Profiling

Because reading and writing input/output reports to the HID device has to be done as quickly as possible to reach the maximum update rate of the controllers (which results in the best possible tracking achievable with this hardware), read and write performance should be measureable. Similarly, the performance of tracking the controller position in the camera image must be made measurable, as a high frame rate here is also an important factor for some use cases.

Functional Tests

For each feature of the library, a small test application should be provided that can be used to test that specific feature and provide human-readable output (where it makes sense, this output should also be graphical).

In addition to serving as a way of verifying the correct function of any given part of the library (which helps in tracking down problems using the library), this also allows the developer and other contributors to verify that changes to the core library do not cause regressions.

Debugging Output

For maintainers of the library and for debugging purposes, the library should have an option to be built with additional debugging information. This is very helpful in spotting bugs and diagnosing unexpected behaviour.

Information Hiding

Between the library and user code, only an opaque handle (a pointer) is passed, making sure that all modifications of internal data structures are done via API calls. One advantage of this approach is that the library developer can be sure that all modifications to the state of its objects are only carried out by the library functions, and not by random user code. Another advantage is

that the developer of the library has the freedom to change the internal structure (and allocation size) of objects managed by the library without breaking the API (source interface) or even the ABI (binary interface) for end user applications.

Object-Oriented Programming

Even if the language in which the core library is written doesn not have object oriented features built-in, the API is designed with object-oriented patterns, which makes it easier for developers to see the structure behind the library's types. This design makes it easier to create higher-level language bindings for languages that are object-oriented. Another feature of using object-oriented design is that functions that belong together have the same prefix in their name.

Defensive Programming

Assertions should be used at critical positions in the code (such as at the beginning of a library function to check parameters). With many assertions, code may not be as fast as it could be (without assertions), but it also means that errors are detected much earlier. When assertions fail, the error message should point out which assertion failed to ease debugging and troubleshooting.

Modular Programming

The core library and the tracker part are built as two separate libraries, allowing developers to use only the core library when the tracker part is not needed - this is especially useful for projects that do not want to ship many megabytes of unused dependencies with their application. In the concrete case of the PS Move API, not linking against the tracker library means that OpenCV does not have to be linked and shipped with the final application.

CHAPTER 4

Implementation

This chapter describes the implementation of the PS Move API library. In section 4.1, the high-level architecture of the implementation is given. Section 4.2 will list all third-party dependencies that the library needs or incorporates. The modules of the implementation are described in section 4.3 and 4.4: Section 4.3 describes the public modules that are available to the user of the library, whereas section 4.4 describes the modules internal to the PS Move API, which are not exposed as part of the public API. Bindings for additional programming languages are introduced in 4.5, and are documented with short code examples. For distributed deployments, the Move Daemon (described in section 4.7) can be used. Finally sections 4.8 and 4.9 deal with platform-specific customizations for pairing and camera access, respectively.

Library API Documentation

The API documentation for the modules described here can be found in appendix A, which also includes instructions on how to build the API documentation from source using Doxygen.

Obtaining the Source Code

The source code of the PS Move API is available under a BSD-style license from

http://github.com/thp/psmoveapi

4.1 Architecture Overview

The PS Move API consists of a C library (figure 4.1), language bindings on top of that library (figure 4.2) and core utilities that make use of the C library and are essential for using and setting up the PS Move API (figure 4.3).

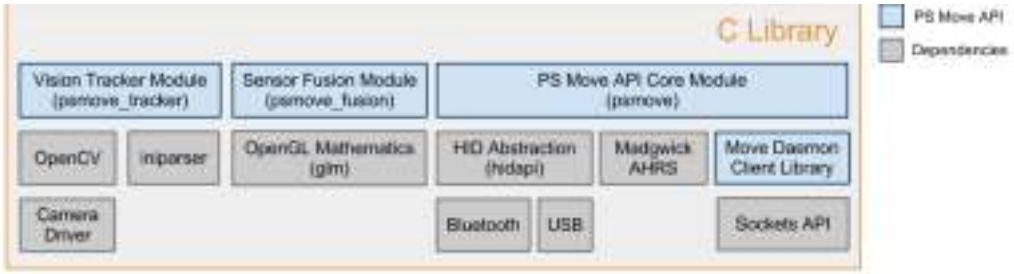


Figure 4.1: PS Move API Architecture: Libraries and Dependencies

C Library

The C library includes the three main modules of the library (figure 4.1). Items in blue are described in sections 4.3 and 4.4, items in grey are dependencies and described in section 4.2.

The PS Move Fusion library implements sensor fusion, and depends on both the PS Move Tracker library and the PS Move API Core. Both the PS Move Tracker and the PS Move Fusion libraries are optional. If a module is used, however, all the dependencies listed below it become a required dependency.

Bindings

Bindings for existing frameworks and languages are built on top of the C library (figure 4.2): The OpenTracker integration is described in detail in section 5.8. The Simplified Wrapper and Interface Generator SWIG [22] is used to generate the language bindings for Java, Python and C#. The Processing bindings are basically a specially-packaged form of the Java bindings (so that Processing can pick them up) and have been contributed by the open source community.

The old C# bindings (UniMove) and Qt/C++ bindings are still available, but are not maintained anymore. Users of the old C# bindings should switch to the new C# bindings based on SWIG, and users of the Qt bindings should switch to the C library.

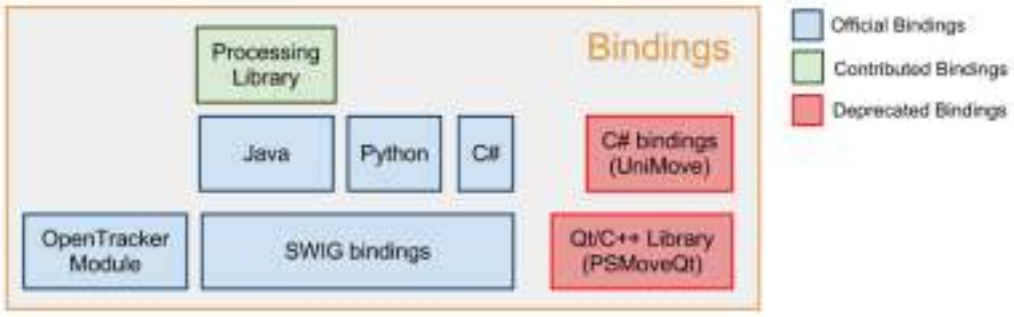


Figure 4.2: PS Move API Architecture: Language Bindings

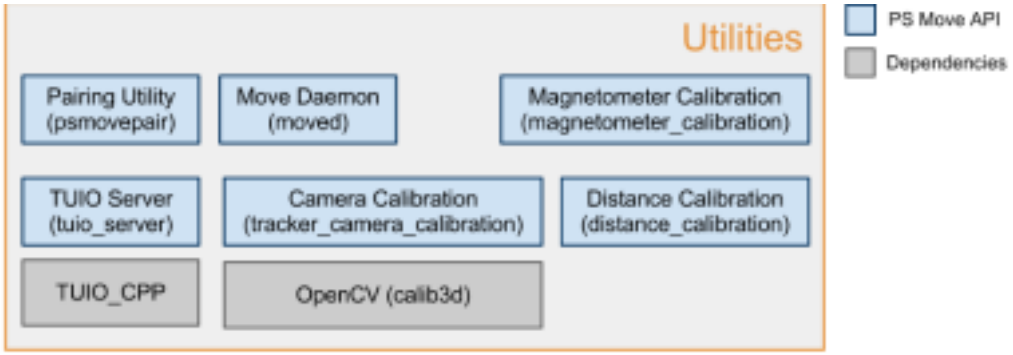


Figure 4.3: PS Move API Architecture: Essential Utilities

Utilities

Some core utilities are provided with the PS Move API (figure 4.3), which can be used to carry out tasks related to pairing, calibrating and communicating with the controllers. The Pairing Utility (psmovepair) is used to pair a controller to the host computer using USB (see section 3.3 for details). The Move Daemon (moved, see section 4.7) is used to publish locally-connected controllers via a local network. The Magnetometer Calibration utility is used to calibration the magnetometer reading range for a specific environment (see section 3.7). The Distance Calibration utility is used to calibrate the radius-to-distance mapping for a specific camera (see section 3.6 on how to use this utility).

Other utilities add support for optional calibration features and for the TUIO protocol. Grey items are again library dependencies.

4.2 Dependencies

This section lists the mandatory and optional dependencies of the PS Move API from third parties. For each dependency, the usage and requirement is listed, as well as the license under which the dependency is available or used. If customizations or changes to the libraries have been carried out (either locally or contributed upstream), they are also described here.

hidapi

- **Mandatory:** yes
- **Usage:** Cross-platform HID communication (USB/Bluetooth)
- **License:** 3 choices: GNU GPL v3, BSD or HIDAPI license

Alan Ott's hidapi [26] abstracts away the platform-specific differences in opening HID devices and reading/writing data from/to them. In the PS Move API, we use it to communicate both via USB during the pairing process and via Bluetooth during the tracking process. The

hidapi abstracts away enumerating and opening devices. In situations where enumeration was not working as expected, patches were contributed upstream[1].

OpenCV

- **Mandatory:** no (required for Tracker module)
- **Usage:** Camera input and image processing in Tracker module
- **License:** BSD

The OpenCV library [14] is used for abstraction of the camera input and image manipulation operations necessary to track the controller in the camera image. The modules **core**, **imgproc** and **highgui** are used by the PS Move API implementation. For the camera calibration application, the **calib3d** module is used, but it is otherwise optional (and not used in the tracker library itself).

Madgwick's AHRS Algorithm

- **Mandatory:** no (required for orientation)
- **Usage:** Convert inertial sensor data into 3D rotation quaternion
- **License:** GNU GPL

Sebastian Madgwick's open source AHRS algorithm [21] is used for orientation tracking using the inertial sensor data from the controller. The code was slightly modified to allow for multiple controllers to be tracked (the reference implementation assumes that only one IMU/AHRS is used), and therefore is shipped with the PS Move API source code in **external/Madgwick-AHRS/**.

OpenGL Mathematics

- **Mandatory:** no (required for sensor fusion)
- **Usage:** Matrix operations for sensor fusion
- **License:** MIT

The OpenGL Mathematics library [7] provides a header-only C++ implementation of OpenGL data types (matrices, vectors, etc...). This library is used in the sensor fusion module to carry out calculations of the modelview and projection matrices without depending on the OpenGL implementation (fixed function pipeline) itself. In the sensor fusion module, the modules used are: **glm.hpp**, **gtc/matrix_transform.hpp**, **gtc/quaternion.hpp** and **gtc/type_ptr.hpp**.

[1] hidapi pull request 62, https://github.com/signal11/hidapi/pull/62, retrieved 2012-11-26

iniparser

- **Mandatory:** no (required for Tracker module)
- **Usage:** Backup and restore of camera settings
- **License:** MIT

This small C library written by Nicolas Devillard can read and write .ini-style configuration files. It is currently used only in the camera control part of the tracker module to save and restore the camera configuration parameters.

TUIO_CPP

- **Mandatory:** no (required only for TUIO server application)
- **Usage:** TUIO protocol implementation
- **License:** GNU GPLv2 or later

For the TUIO server example application, an implementation of the TUIO protocol is needed. The reference C++ implementation by Martin Kaltenbrunner [17] is available for this purpose, and is used in the TUIO server. The PS Move API libraries don't link against the TUIO library, only the TUIO server application.

4.3 Public Modules

This section describes all modules that have public headers for use by developers utilizing the API. The libraries are listed in order of dependence: The Core module does not depend on any of the other modules, the Tracker module depends on the Core modules and the Fusion module depends on both the Core and the Tracker modules.

PS Move Core (include/psmove.h, src/psmove.c)

The API documentation of this module can be found in appendix A (section A.1).

- **Provides:** Controller connectivity, PSMove handle
- **Library dependencies:** hidapi, Bluetooth
- **Internal dependencies:** Calibration, Orientation

This module provides the **PSMove** object type, as well as essential structs, typedefs and symbolic constants. The PSMove object type is an opaque struct representing a connection to a single controller. The module also provides core functions for pairing and communicating with the controller, as well as setting LEDs and reading sensors.

This module makes use of the internal modules for calibration and orientation (see section 4.4 for details), and exposes an API on the PSMove object type for using calibration and orientation data (it is usually not useful for users to deal with separate calibration and orientation objects manually).

As far as library dependencies are concerned, the PS Move Core module has a direct dependency on hidapi and the platform-specific Bluetooth libraries (for getting the host controller Bluetooth address). The Core module also includes platform-specific code (in src/platform/) for certain features, such as camera detection on Linux or platform-specific registration of controllers in the Bluetooth stack (see section 4.8).

This module also implements the Move-specific HID Protocol for receiving input reports and sending LED updates. A detailed description of this protocol can be found in appendix B.

PS Move Tracker (include/psmove_tracker.h, src/tracker/psmove_tracker.c)

The API documentation of this module can be found in appendix A (section A.2).

- **Provides:** Camera connectivity, PSMoveTracker handle, vision tracking
- **Library dependencies:** OpenCV, iniparser
- **Internal dependencies:** Core

This module links against OpenCV and the PS Move API Core and encapsulates all features required to track motion controllers using a camera by providing the **PSMoveTracker** object type. It takes care of detecting the connected camera, setting up the camera source (configuring exposure and other parameters) and doing the tracking of the camera image itself. It returns the size and coordinates of detected spheres. This library is optional and only used when 3D positioning is required.

The camera parameters are saved before being modified (using iniparser), and are restored when the application quits (this makes sure that the low exposure settings used in the PS Move API are not affecting the normal webcam usage of the camera device when the PS Move API is not running).

The Tracker module implements the tracking algorithms described in section 3.5.

PS Move Fusion (include/psmove_fusion.h, src/tracker/psmove_fusion.cpp)

The API documentation of this module can be found in appendix A (section A.3).

- **Provides:** Sensor fusion (position and rotation matrices), PSMoveFusion handle
- **Library dependencies:** OpenGL Mathematics (glm)
- **Internal dependencies:** Core, Tracker

This module depends on both the Core and Tracker libraries, and implements the sensor fusion part of the library by providing the **PSMoveFusion** object type. The library can be used to create an OpenGL-based 3D application, and already provides functions to get the projection matrix (for the intrinsic camera parameters) and the modelview matrix (for the controller orientation and position) to do OpenGL rendering. When using the controller-specific modelview matrix returned by this module, the coordinate system origin will be at the controller sphere's center, and will be aligned with the controller's position.

Matrix calculations are carried out by the OpenGL Mathematics library, which becomes a dependency if the Sensor Fusion module should be used. Internally, this module depends on both the Core and Tracker modules.

4.4 Private Modules

The following modules are only used internally, their functionality is exposed via the Public APIs above. Their documentation here serves as guide for developers wanting to modify or improve the PS Move API itself.

Calibration (src/psmove_calibration.h, src/psmove_calibration.c)

- **Provides:** Saving, loading and interpretation of inertial sensor calibration data (accelerometer + gyroscope)
- **Library dependencies:** None
- **Internal dependencies:** Core

This modules takes care of saving and loading the calibration blob (it can only be saved during a USB connection, which usually happens during USB pairing). Also, the module implements the interpretation of the calibration blob, and provides mapping functions from raw accelerometer and gyroscope values to the calibrated values.

Orientation (src/psmove_orientation.h, src/psmove_orientation.c)

- **Provides:** 3D orientation tracking from inertial sensors
- **Library dependencies:** Madgwick AHRS
- **Internal dependencies:** Core, Calibration

The Orientation module wraps the AHRS algorithm and provides means to reset the orientation to a known "good" orientation (e.g. when the controller points towards the camera). It depends on the Calibration module for getting calibrated sensor readings, and returns the orientation in quaternion representation.

Move Daemon Client (moved_client.h, moved_client.c in src/daemon/)

- **Provides:** Client-side interface to a remote Move Daemon
- **Library dependencies:** Sockets API
- **Internal dependencies:** None

This library implements a low-level C library for interfacing with remote Move Daemon instances (see section 4.7) via the Move Daemon Protocol. The Move Daemon Protocol is a UDP-based protocol that can be used to forward HID requests from a client application to a remote host.

The PS Move API Core supports connecting to controllers using this protocol, and the Move Daemon application is supplied as part of the PS Move API package.

Move Daemon Monitor (moved_monitor.h, moved_monitor.c in src/daemon/)

- **Provides:** Connection event monitoring (Linux only for now)
- **Library dependencies:** Linux Input, libudev
- **Internal dependencies:** None

This library implements a connection monitor that can detect newly-connected and removed devices while an application is running. Right now, the implementation is only available in Linux using libudev, but the API has been kept general enough that future improvements could add implementations for other systems using the same function signatures.

When running the Move Daemon on Linux, this library takes care of informing the Move Daemon when new devices connect or existing devices disconnect, so that the Move Daemon can re-enumerate and add the device on the fly during runtime. This method works on all Linux hosts where moved is running, even if the client machines themselves do not run Linux.

4.5 Language Bindings

Only the SWIG-based language bindings are described in this section. The OpenTracker module is described in section 5.8 and the deprecated bindings (UniMove and PSMoveQt) are not documented here, as they should not be used in new projects.

Most of the code examples in this section are also available (in slightly extended form) in the folder **examples/** in the PS Move API Git source tree - these examples can be used as a starting point for custom applications.

Java

The Java bindings generated by SWIG utilize JNI [6] to expose the C Library functions to the Java Virtual Machine. As Java does not support features such as reference parameters or pointers for primitive data types, single-element arrays have to be used as a workaround where

pointers are used in the C library (e.g. multi-value functions such as the gyroscope functions). Also, module-global functions such as **psmove_count_connected()** are exposed as static member functions of the class **io.thp.psmove.psmoveapi**. Apart from these technical differences, the Java library maps the C API closely.

The following two code examples show you how to read sensor values from the controller with single-element arrays, and how to use the Tracker library together with the Core library to get the controller position in the camera image. Advanced features such as obtaining the camera image are also possible, but they will be demonstrated in the Processing code examples.

Code Example: Sensor Reading

This example shows how to count the connected controllers, how to connect to a PS Move controller and read calibrated inertial sensor values from the controller into the Java application:

```
1  import io.thp.psmove.PSMove;
2  import io.thp.psmove.Frame;
3  import io.thp.psmove.psmoveapi;
4
5  public class SensorReading {
6      public static void main(String [] args) {
7          int connected = psmoveapi.count_connected();
8          System.out.println("Connected controllers: " + connected);
9
10         PSMove move = new PSMove();
11         float [] ax = { 0.f }, ay = { 0.f }, az = { 0.f };
12         float [] gx = { 0.f }, gy = { 0.f }, gz = { 0.f };
13         float [] mx = { 0.f }, my = { 0.f }, mz = { 0.f };
14
15         while (true) {
16             while (move.poll() != 0) {
17                 move.get_accelerometer_frame(Frame.Frame_SecondHalf,
18                         ax, ay, az);
19                 move.get_gyroscope_frame(Frame.Frame_SecondHalf,
20                         gx, gy, gz);
21                 move.get_magnetometer_vector(mx, my, mz);
22                 System.out.format("ax: %.2f ay: %.2f az: %.2f ",
23                         ax[0], ay[0], az[0]);
24                 System.out.format("gx: %.2f gy: %.2f gz: %.2f ",
25                         gx[0], gy[0], gz[0]);
26                 System.out.format("mx: %.2f my: %.2f mz: %.2f\n",
27                         mx[0], my[0], mz[0]);
28             }
29         }
30     }
31 }
```

In line 7 and 8, the number of connected controllers is obtained and printed. Line 10 connects to the first available Move controller. Lines 11 to 13 create single-element arrays for the three-axis accelerometer, three-axis gyroscope and three-axis magnetometer. Lines 15 and 16 are the main loops - line 15 is an infinite loop - in it you could do other processing, like updating the

GUI. Line 16 makes sure that all queued updates are read from the controller (it will return 0 when no more updates have been read). Lines 17 to 21 use the single-element arrays to obtain the calibrated sensor readings, and lines 22 to 27 print out these values on the console.

In Java, there is no need to close the controller connection, as this will be done automatically when the PSMove object is garbage-collected.

Code Example: Vision Tracker

This example shows how to get a PSMoveTracker object and how to interface it with an existing PSMove object by enabling tracking and getting the current position and size of the controller in the tracked image. Advanced features such as mirroring the camera image from the tracker are also demonstrated, these are optional, but make sense for use cases where the user interacts with the on-screen picture. Mirroring changes the appearance of the tracker image, and also the X coordinate of the controller in the position results:

```
1  import io.thp.psmove.PSMove;
2  import io.thp.psmove.Frame;
3  import io.thp.psmove.psmoveapi;
4
5  import io.thp.psmove.PSMoveTracker;
6  import io.thp.psmove.Status;
7
8  public class Tracker {
9      public static void main(String [] args) {
10         int connected = psmoveapi.count_connected();
11         System.out.println("Connected controllers: " + connected);
12
13         PSMoveTracker tracker = new PSMoveTracker();
14         PSMove move = new PSMove();
15
16         // Mirror the camera image
17         tracker.set_mirror(1);
18
19         while (tracker.enable(move) != Status.Tracker_CALIBRATED);
20
21         while (true) {
22             tracker.update_image();
23             tracker.update();
24
25             if (tracker.get_status(move) == Status.Tracker_TRACKING) {
26                 float [] x = { 0.f }, y = { 0.f }, radius = { 0.f };
27                 tracker.get_position(move, x, y, radius);
28                 System.out.format("x: %5.2f y: %5.2f radius: %5.2f\n",
29                         x[0], y[0], radius[0]);
30             } else {
31                 System.out.println("Not tracking.");
32             }
33         }
34     }
35 }
```

As in the previous code example, lines 10 and 11 output the number of currently-connected controllers. Line 13 creates a new PSMoveTracker object that manages the camera connection. Even if multiple controllers are used, only one tracker object is needed.

Line 14 creates a new PSMove object (for the first connected PS Move Motion Controller). Line 17 tells the tracker to mirror the camera image and all calculations based on it. Line 19 registers the controller with the tracker, which starts the blinking calibration. When the blinking calibration is successful, the enable() function returns Status.Tracker_CALIBRATED. If not, the application will try again until the controller is calibrated.

Line 21 starts the main loop. In the main loop, we have to grab a new image from the camera (line 22) and use that image to find the controller (line 23). If the controller is currently tracked (line 25), the position is obtained and printed (lines 26 to 28). If not, tracking is not available currently (e.g. because the controller is not visible), an information message is printed (line 31).

Also here, both the tracker and move objects do not have to be explicitly closed or freed - the connection will be closed when the objects are garbage-collected.

Processing

The Processing library uses the Java bindings, and adds a special packaging layout on top of it that allows the PS Move to be used in Processing [10]. The bindings have been contributed by the open source community (justinb26, Jules Fennis and Raphaël de Courville) based on the Java bindings.

The bindings have been successfully tested on Mac OS X and Linux, and are expected to work on Windows as well. Version 1.5.1 of Processing was used to create and test the example code listed below - it should also work with version 2.0, but at the time of writing, only beta versions of Processing 2.0 have been available.

Code Example: Displaying the Camera Image

To demonstrate the usage and advantage of the Processing bindings here, a simple Processing application using the PS Move and the Tracker module is shown on the next page.

First, the PS Move API Java bindings are imported (line 1). Then, the objects for the controller (line 3), the tracker (line 4), the pixel data (line 5) and the Processing-specific image object (PImage, line 6) are declared.

Line 8 starts the setup() block - this will be called once by the Processing runtime when the application is started. The window size is set to 640x480 (line 9), and the connection to the controller and tracker are set up (lines 10 to 13). Finally, in line 14 the blinking calibration is initialized and will be tried until the calibration is successful.

Line 17 starts the draw() block - this block will be called every time a new frame needs to be drawn from Processing. In line 18 and 19, the camera image is updated and processed inside the tracker. In line 21, an object describing the camera RGB image is obtained. Lines 22 to 23 allocate a new byte array the first time it is needed. Then, the pixel data is retrieved from the RGB image object and stored into the byte array. Lines 26 to 28 create a new RGB image in Processing the first time it is needed. Lines 29 to 33 demonstrate how to load the pixel data from the byte array into the Processing PImage. Finally, in line 34, the camera image is displayed.

```
import io.thp.psmove.*;

PSMove move;
PSMoveTracker tracker;
byte [] pixels;
PImage img;

void setup() {
    size(640, 480);
    move = new PSMove();
    move.set_leds(255, 255, 255);
    move.update_leds();
    tracker = new PSMoveTracker();
    while (tracker.enable(move) != Status.Tracker_CALIBRATED);
}

void draw() {
    tracker.update_image();
    tracker.update();

    PSMoveTrackerRGBImage image = tracker.get_image();
    if (pixels == null) {
          pixels = new byte[image.getSize()];
    }
    image.get_bytes(pixels);
    if (img == null) {
       img = createImage(image.getWidth(), image.getHeight(), RGB);
    }
    img.loadPixels();
    for (int i=0; i<img.pixels.length; i++) {
        img.pixels[i] = color(pixels[i*3] & 0xFF, pixels[i*3+1] & 0xFF,
            pixels[i*3+2] & 0xFF);
    }
    img.updatePixels();
    image(img, 0, 0);
}
```

Python

Python is an interpreted high-level programming language with dynamic typing. It is especially useful for quickly prototyping code, but it can also be used for real-world production code. Like Java, Python does not support reference parameters of local primitive variables, but does support multi-value returns by means of tuples. This is used in the SWIG-based bindings to return multiple values to the caller, such as for the gyroscope readings.

Code Example: Fading Colors

This examples fades the LED color between red and green, it uses only the core library of the PS Move API, and therefore even works with the controller connected via USB (no sensors used):

```
1  import time
2  import math
3  import psmove
4
5  move = psmove.PSMove()
6
7  i = 0
8  while True:
9      r = int(128 + 128*math.sin(i))
10     move.set_leds(r, 255 - r, 0)
11     move.update_leds()
12     time.sleep(0.1)
13     i += 0.2
```

Lines 1 to 3 import the required modules: The time module is used for the sleep() method, the math module is used for the sine function. The psmove module is the binding module for the PS Move API. In line 5, a connection to the first available controller is created. Lines 7, 8 and 13 describe a loop with the loop variable i going from 0 to infinity, with a step of 2. In the loop body (lines 9 to 12), a color value is calculated and then sent to the controller. In line 12, the program sleeps for 0.1 seconds to control the speed at which LED updates are sent.

Code Example: Calibrated Sensor Values

Similar to the Java code examples, this code example basically does the same thing - it retrieves the calibrated sensor values from the controller and outputs them. An important aspect here is that instead of using single-element arrays by reference like in Java, Python has the ability to return multiple values as tuples, and can unpack these tuples using the syntax shown in the code example. This method is used throughout the SWIG-based Python bindings:

```
1  import psmove
2
3  move = psmove.PSMove()
4
5  while True:
6      if move.poll():
7          ax, ay, az = move.get_accelerometer_frame(psmove.Frame_SecondHalf)
8          gx, gy, gz = move.get_gyroscope_frame(psmove.Frame_SecondHalf)
9
10         print 'A: %5.2f %5.2f %5.2f ' % (ax, ay, az),
11         print 'G: %6.2f %6.2f %6.2f ' % (gx, gy, gz)
```

In line 1, the PS Move API module is imported. Line 3 creates a connection to the default controller. Lines 5 and 6 describe the main loop. Line 7 retrieves the second half-frame of the calibrated accelerometer reading from the controller and stores it as three float values in ax, ay and az. Line 8 does the same thing for the gyroscope values. Lines 10 and 11 then print the readings on standard output.

C#

The C# bindings are in a way similar to the Java bindings, except that C# does have support for out parameters and reference parameters, so these features are supported by the PS Move API bindings for C#. As everything else is similar to the Java bindings, only one code example is shown here, demonstrating the out parameter usage for the Tracker module.

Code Example: Vision Tracker

This example is mostly equivalent to the vision tracker example of Java, but uses C#-specific features such as out parameters:

```
1  using System;
2  using io.thp.psmove;
3
4  public class Tracker {
5      public static int Main(string [] args) {
6          PSMoveTracker tracker = new PSMoveTracker();
7          PSMove move = new PSMove();
8          while (tracker.enable(move) != Status.Tracker_CALIBRATED);
9
10         while (true) {
11             tracker.update_image();
12             tracker.update();
13
14             if (tracker.get_status(move) == Status.Tracker_TRACKING) {
15                 float x, y, radius;
16                 tracker.get_position(move, out x, out y, out radius);
17                 Console.WriteLine(string.Format("Tracking: x:{0:0.000}, " +
18                             "y:{1:0.000}, radius:{2:0.000}", x, y, radius));
19             } else {
20                 Console.WriteLine("Not Tracking!");
21             }
22         }
23
24         return 0;
25     }
26 };
```

In lines 1 and 2, the required modules are imported. Lines 6 to 8 create the tracker instance and the controller instance, and start the blinking calibration for the controller until it succeeds. Lines 11 and 12 update the controller tracking information. If the controller is currently tracked (line 14), the position and radius of the controller are obtained via out parameters of the get_position() function call (lines 15 to 17) and output. If the controller is not tracked, an information message is printed on the console.

4.6 Build System

CMake [13] is used as the build system for the PS Move API. It is an open source meta-build system that can target multiple build environments such as make, Xcode or Visual Studio. For

official PS Move API builds, GNU make is used on Linux and Mac OS X – on Windows, MinGW make is used.

To build the PS Move API, it is usually sufficient to issue the following commands in a shell in the PS Move API source directory:

```
mkdir build
cd build
cmake ..
```

After that, CMake will look for required dependencies and output the build configuration:

```
  Build configuration
    Debug build:       No
    Tracker library:   Yes

  Language bindings
    Python:            Yes
    Java:              Yes
    C#:                No (disabled)
    Processing:        Yes
    Qt:                No (disabled)

  Tracker
    PS Eye support:    Yes
    HTML tracing:      Yes
    Use CL Eye SDK:    No (Windows only)

  Additional targets
    C example apps:    Yes
    OpenGL examples:   Yes
    C test programs:   Yes
    C++ TUIO server:   Yes

-- Configuring done
-- Generating done
-- Build files have been written to: /home/thp/src/psmoveapi/build
```

You can use the "ccmake" (CMake Curses UI) or "CMake GUI" applications to modify this configuration. If options are enabled, but their dependencies can not be satisfied, the build configuration screen will point out the missing dependencies.

Some configuration options change the behavior of the resulting libraries:

- **PSMOVE_USE_CL_EYE_SDK**: On Windows, use the CL Eye SDK instead of the normal OpenCV-based image capture mechanism.
- **PSMOVE_USE_DEBUG**: Print additional verbose debug output at runtime. This is useful for development and debugging.
- **PSMOVE_USE_DEBUG_CAPTURE**: Display captured camera images with OpenCV (for debugging exposure and dimming code). Do not use this option when using another GUI toolkit, such as Qt.

- **PSMOVE_USE_LOCAL_OPENCV**: Prefer a local OpenCV checkout and build (in $PSMOVEAPI_SOURCE_DIR/opencv/) over a system-wide OpenCV installation (this is used for the release builds).

- **PSMOVE_USE_PSEYE**: Compile with additional code for the PS Eye camera (e.g. camera detection code on Linux). Disable if you do not plan on using the PS Eye camera.

- **PSMOVE_USE_TRACKER_TRACE**: Compile the tracker module with HTML tracing support. When this option is enabled, the tracker will store screenshots of the blinking calibration in $HOME/.psmoveapi/.

Other options to disable or enable building language bindings and test and example programs, as well as disabling the building of the tracker module are also available and are described in the CMake GUI application or by using cmake in wizard mode using "cmake -i".

4.7 The Move Daemon (moved)

The Move Daemon - **moved** - acts as a UDP-based network service that will expose the controllers connected to the local machine to a local network. It was originally developed to keep L2CAP connections open on Linux before the current way of going through Bluez/hidraw was used, and has been enhanced in January 2012 during the Nording Game Jam for developing an application that interfaces with more than 7 Motion Controllers in parallel (7 devices is the theoretical maximum of controllers connected to a single Bluetooth adapter, on some more recent MacBook Pros on Mac OS X, we were able to connect up to 9 controllers to a single machine, whereas older laptops maxed out at 5 devices).

The PS Move API has built-in support for interfacing with moved instances running on remote hosts, see appendix C for details on the protocol.

On Linux, moved utilized **libudev** to get real-time notifications of removed and newly-added controllers. New controllers will show up to remote hosts as soon as they are connected, devices connected via USB will be paired automatically (using **psmove_pair()**). This allows the Move Daemon to run as unattended server on a headless system, exposing connected devices via the local network.

On all other operating systems, moved will have to be restarted when controllers are added and removed. Because the UDP protocol is stateless, clients will usually continue working, but the order of controllers might change, and clients should be implemented in a way that can handle a reordering of controllers.

A very common use case of the Move Daemon is to pair controllers to a Linux or Mac OS X host (for which known "good" pairing procedures exist), and use the services of that host on a Windows machine via an Ethernet connection (currently there is no reliable procedure for pairing the controller on Windows).

The "moved-hosts.txt" File

The client library uses the file **moved-hosts.txt** in the PS Move API data directory (.psmoveapi, which is located in the $HOME directory on Linux and Mac OS X, and in %APPDATA% on Windows) to determine which (if any) remote hosts to try to connect to using the moved protocol. The file has a simple text file, with one host per line. On Windows, this must be an IP address, on Linux and Mac OS X you can use either an IP address or a hostname that will be resolved at runtime.

4.8 Controller Bluetooth Pairing via USB

In the library, the pairing process is implemented in the **psmove_pair()** function. For most use cases, users will want to use the supplied **psmovepair** utility that takes care of pairing all controllers currently connected via USB. It also supports pairing to a different host by accepting an alternative Bluetooth host as command-line argument. The API function for using a custom address is **psmove_pair_custom()**, but in general using the **psmovepair** utility is sufficient.

On Linux, the Move Daemon (**moved**) has built-in support for pairing USB controllers once they are connected by monitoring connects and disconnects via **libudev**. It is possible to have the same behavior on other operating systems as well, but it has not yet been implemented as part of this project.

The "psmovepair" Utility

The pairing utility is used for pairing a controller via USB to the host machine, so that the controller will connect to the correct machine when switched on. By default, the host address that will be written to the controller will be determined by a method that is specific to the operating system in use (on Linux: libbluetooth, on Windows: libbthprops, on Mac OS X: IOBluetooth). If a command line argument is supplied, it should be a host address in the format "AA:BB:CC:DD:EE:FF". This address will be used as the host address for pairing (see section 3.3 for more information).

On Linux, the **psmovepair** utility has to be run as root or using sudo, because it automatically restarts the Bluetooth Daemon and writes the pairing information into bluetoothd's state files - this makes sure that connections from the controller are always accepted and work properly (no manual steps for pairing are necessary after pairing on Linux with this method).

On Mac OS X, the utility will add the right entries to the system configuration. This has been tested on Mac OS X 10.8, and is not needed for Mac OS X 10.6. On Mac OS X 10.7, it should also work, but might require pairing the controller twice before it works.

Pairing on Mac OS X

The following section applies to Mac OS X 10.7.3 and newer. Older versions of Mac OS X do not require special entries when pairing - as soon as the controller has been pair via USB, OS X will allow HID connections without any additional warnings/questions. This has most likely

been seen as a security issue, and this is why pairing on OS X 10.7.3 and newer requires the steps outlined in this section.

Mac OS X ships with its own proprietary Bluetooth stack. Bluetooth is well-supported in Mac OS X, and there are no alternative Bluetooth stacks that are used. The system-wide configuration file that stores information about the paired HID devices is

/Library/Preferences/com.apple.Bluetooth.plist

This file is stored in Binary PList format. On Mac OS X, the command-line "plutil" application can be used to convert it to an XML PList file that can be opened and edited with a normal text editor. In addition to editing the file directly, the "defaults" command can be used to view and modify the values directly. For example, the following command lists all paired Bluetooth HID devices[2]:

```
defaults read \
    /Library/Preferences/com.apple.Bluetooth \
    HIDDevices
```

An example output of this command could look like this:

```
(
    "e0-ae-5e-00-00-00",
    "e0-ae-5e-aa-bb-cc",
    "00-06-f7-22-11-00",
)
```

For adding entries to this list, it is important that the Mac OS X Bluetooth Daemon (**blued**) is shut down, otherwise it will not load the updated configuration and even overwrite it when it is shut down or restarted. There are several ways to shut down the Bluetooth Daemon:

- Use the Bluetooth Applet in the menu bar to turn Bluetooth off
- Kill the "blued" binary from the command line
- Use the private API function IOBluetoothPreferenceSetControllerPowerState in the IOBluetooth framework (only on OS X 10.7 and newer)

The first two options usually require user interaction, while the third option is ideal for automatically shutting down and re-activating Bluetooth in a C application. The PS Move API implementation therefore uses the private API functions from the IOBluetooth framework.

After Bluetooth has been shut down, the Bluetooth address of the controller needs to be added to the HIDDevices section of the com.apple.Bluetooth.plist file. This can be accomplished by running the following command as privileged (root) user:

```
defaults write \
    /Library/Preferences/com.apple.Bluetooth \
    HIDDevices -array-add aa:bb:cc:dd:ee:ff
```

[2] the "defaults" command omits the .plist extension

Where “aa:bb:cc:dd:ee:ff” is the Bluetooth address of the controller to be added. After that, Bluetooth can be started again and the controller will be able to connect to the Mac OS X computer.

To summarise, the steps required to register a controller in Mac OS X are:

1. Use “defaults read” to check if the controller is already paired - if so, we’re done
2. If not, shutdown Bluetooth using the IOBluetooth private APIs
3. Wait for the “blued” process to shutdown (this is required on Mac OS X 10.8 to avoid a race condition with caching)
4. As superuser, add the controller’s Bluetooth address using “defaults write”
5. Switch Bluetooth on again using the IOBluetooth private APIs
6. Press the PS Button to connect the controller

The implementation of this process and OS X-specific helper functions can be found in the PS Move API source tree in **src/platform/psmove_osxsupport.m**

Pairing on Linux

The default Bluetooth stack on Linux is Bluez [1]. Bluez consists of various utilities and libraries - for PS Move pairing, the important components are:

- **bluetoothd** - The Bluetooth Daemon, handling connections from Bluetooth devices and taking care of authentication and setting up connections. When pairing a PS Move Motion Controller, we need to add its Bluetooth address to bluetoothd’s configuration in order for bluetoothd to accept connections and set up the controller as HID device.
- **libbluetooth** - A userspace library that exposes Bluetooth functions to end-user applications. We use it to determine the Bluetooth host address of the computer for the pairing process.

The state of **bluetoothd** is usually stored in **/var/lib/bluetooth/** in a separate directory for each Bluetooth host adapter on the system (usually, there’s just one host adapter installed). The adapter-specific directory contains various plaintext files, with one line per entry. For the purposes of pairing a PS Move Motion Controller, the important files are:

- **classes** - This file has to have an entry with the value “0x002508” for each paired PS Move Motion Controller.
- **did** - This file has to have an entry with the value “0000 054C 03D5 0000” for each paired PS Move Motion Controller. The values represent the HID product and vendor ID, and are used in the hidapi enumeration process to determine if a given hidraw device is a PS Move Motion Controller or some other HID device.

- **features** - This file has to have an entry with the value “BC04827E08080080” for each paired PS Move Motion Controller.
- **names** - This file has to have an entry with the value “Motion Controller” for each paired PS Move Motion Controller.
- **profiles** - This file has to have an entry with the UUID of the Bluetooth HID profile for each paired PS Move Motion Controller: 00001124-0000-1000-8000-00805f9b34fb
- **sdp** - This file has to contain an SDP entry string for each PS Move Motion Controller (see below).
- **trusts** - This file has to have an entry with the value “[all]“ for each paired PS Move Motion Controller. If this entry is not present, the Bluetooth Agent will ask for permission when the controller tries to connect to the host computer.

The entry in the **sdp** file has to have the following value (as the bluetoothd state files are one-line-per-entry plaintext files, this entry has to be typed all in one line, with a space only between “#00010000” and the rest of the entry):

```
#00010000
3601920900000A000100000900013503191124090004350D35061901000
900113503190011090006350909656E09006A0901000900093508350619
1124090100090000D350F350D350619010009001335031900110901002 51
3576972656C65737320436F6E74726F6C6C6572090101251357697 2656C
65737320436F6E74726F6C6C6572090102251B536F6E7920436F6D70757
4657220456E7465727461696E6D656E74090200090100090201090100 09
0202080009020308210902042801090205280109020635 9A35980822259
405010904A101A1028501750895011500 26FF008103750195131500 2501
350045010509190129138102750195 0D0600FF8103150026FF000501090
1A1007508950435004 6FF0009300931093209358102C005017508952709
018102750895300901910275089530090 1B102C0A102850275089530090
1B102C0A10285EE750895300901B102C0A10285EF750895300901B102C0
C009020735083506090409091000902082800090209280109020A28010
9020B09010009020C093E8009020D280009020E2800
```

In summary, a properly-paired controller with the Bluetooth address **AA:BB:CC:DD:EE:FF** would show up in the configuration of a host adapter **00:11:22:33:44:55** like this:

```
/var/lib/bluetooth/00:11:22:33:44:55% grep -r AA:BB:CC:DD:EE:FF *
profiles:AA:BB:CC:DD:EE:FF 00001124-0000-1000-8000-00805f9b34fb
manufacturers:AA:BB:CC:DD:EE:FF 10 3 6735
lastused:AA:BB:CC:DD:EE:FF 2012-09-14 13:37:07 GMT
classes:AA:BB:CC:DD:EE:FF 0x002508
features:AA:BB:CC:DD:EE:FF BC04827E08080080
names:AA:BB:CC:DD:EE:FF Motion Controller
sdp:AA:BB:CC:DD:EE:FF#00010000 3601920900000A0001000...
did:AA:BB:CC:DD:EE:FF 0000 054C 03D5 0000
trusts:AA:BB:CC:DD:EE:FF [all]
```

Note that the entries in **manufacturers** and **lastused** have not been mentioned above, because their values can be determined and saved by Bluez itself when the controller connects. Setting these values is not required for proper pairing of the controller.

Android also uses Bluez, but the state files are stored in **/data/misc/bluetoothd/**.

In general, users on Linux do not have to care about adding these entries manually, as the library will take care of checking for these entries at pairing time and modifying/adding wrong/missing entries. This only works if the **psmovepair** utility is run as root (or via sudo), as the utility might need to start/stop **bluetoothd** and update files in **/var/lib/bluetooth/**, both of which are usually not allowed for non-root user accounts.

Pairing on Windows

On Windows, we depend on the Microsoft Bluetooth Stack. Unfortunately, compared to the Mac OS X and Linux stack, this stack does not store its state in files, but uses the Windows registry instead. This makes it difficult to automate the task of pairing the controller - there is no known way to automate the pairing of the PS Move Motion Controller on Linux. Instead, the following trial-and-error method[3] can be used:

1. Pair the PS Move via USB
2. Open Bluetooth settings ("Change Bluetooth settings") and make sure that "Allow Bluetooth devices to find this computer" is enabled
3. Search for new Bluetooth devices, push the PS button and pair "without code"
4. Open the properties of the device, go to the "Services" tab and tick the checkbox for the HID service - don't hit Apply or OK yet!
5. Press the PS button and immediately hit Apply
6. If pairing doesn't work at first try, retry the last two steps. If the device disappears, redo the whole process
7. Once the device is connected via Bluetooth, the red LED on the bottom of the controller will stay lit

Usually when the pairing has been completed successfully once, the Windows Bluetooth Stack will remember the controller and will always allow the connection at first try. This is even the case when the controller gets paired (via **psmovepair**) to another host computer and then to the same computer again, as the PS Move Pairing functionality doesn't really affect the state of the Windows Bluetooth Stack.

This method has been tested and known to work at least on a 32-bit Windows 7 installation and a 32-bit Windows 8 installation. In all these cases, it might make sense to use a Linux system to do the Bluetooth communcation, and expose the controllers via the Move Daemon (**moved**) over an Ethernet network to a Windows machine.

[3] http://thp.io/2010/psmove/, retrieved 2012-09-25, originally proposed by Viktor Budaházi

Other Bluetooth stacks and HID device drivers do exist for Windows (e.g. the BlueSoleil stack or the MotioninJoy driver), but these are proprietary and have not been tested with the PS Move API. In general, it's best to use the native trial-and-error pairing method in Windows, because it doesn't depend on any closed source third party software, and is very reliable once pairing succeeds.

4.9 Camera Detection and Configuration

This section describes platform-specific detection and configuration issues with the PS Eye camera (on Linux and Windows) as well as the procedure for using an iSight camera on Mac OS X as a fallback solution.

Camera Integration on Windows

On Windows, two drivers exist for the PS Eye camera: The normal "CLEye" driver that supports a single PS Eye camera and uses normal registry-based configuration parameters and the CLEye SDK, which supports multiple cameras and has a separate API for grabbing frames and controlling parameters.

In the PS Move API, we support both approaches, with the default one being the CLEye camera driver, because it can be obtained by users easily, and does not require registration on the CodeLaboratories website. Our CMake build system provides the configuration option "PSMOVE_USE_CL_EYE_SDK" that can be enabled to enable support for the PS Eye SDK. The user must provide the necessary SDK files and headers in order to build against the PS Eye SDK. Apart from the tighter integration and better control over parameters (directly in the SDK instead of relying on setting registry values), the results of using the CL Eye SDK or the CL Eye driver are the same.

PSEye Detection on Linux

On Linux, special code is needed to detect and work with the PS Eye camera when multiple cameras are available (this is usually the case on notebook computers with built-in webcams). Special code has been added to the Tracker module implementation to detect the presence of a PS Eye camera and prefer that over other cameras in the system. The code is in src/tracker/platform/ in the Git repository in the file psmove_linuxsupport.c: All available video devices have their driver name checked (the PS Eye camera uses the "ov534" driver). When a video device with the driver name "ov534" is found, it is preferred over other video devices - otherwise, the first available camera device is used.

Using the iSight Camera in Mac OS X

For this project, a solution was developed with the help of Raphaël de Courville to use the built-in iSight camera of the MacBook Pro for tracking, requiring more user interaction, but working around the driver problems of the PS Eye on recent versions of Mac OS X.

On Mac OS X when using the built-in iSight camera of MacBooks and iMacs, the calibration method is a bit more complicated, because the APIs used to capture the camera image do not allow third party developers to set the exposure. The only setting that third party applications can do is "lock" the exposure, so that it will not auto-adjust during capture, but only once every time the camera device is opened. Because the exposure adjustment happens when the device is opened, the user must hold the PS Move Motion Controller very close to the camera at application startup, so that the exposure will be set to a low level, increasing tracking quality.

The steps for using the iSight camera on Mac OS X are:

- Connect the controller
- Cover the iSight camera with the sphere of the controller
- Start the example application
- The sphere lights up in white - keep it in front of the camera (the exposure auto-adjustment is taking place)
- When the sphere turns dark, move it away from the camera to a distance of at least 50 cm for the blinking calibration
- In some cases, the exposure adjustment does not work correctly[4] - when this happens, re-run the application and re-do the calibration (you can detect this case by looking at the color of the sphere in the camera image - if it is very different from the real-world color of the sphere, re-doing the calibration steps might be a good idea)

[4] https://github.com/thp/psmoveapi/issues/33

CHAPTER 5

Results

This chapters presents experimental performance and accuracy results, as well as example applications and results of integrating the library with other frameworks.

5.1 Evaluation Setup

This section describes the hardware and software used in the experiments in this chapter. For evaluation, we use Mac OS X and Linux. Windows performance has not been tested, as the Linux and Mac OS X ports are more stable and have a stable method of pairing Bluetooth devices.

The point of the evaluation is to compare the performance of the algorithms implemented on two different systems with different CPUs. The fact that the systems have different operating systems is used only for verification that the library itself works comparably well on different platforms (i.e. the faster machine should give faster results) and different setups.

Linux

Linux tests have been carried out using a Samsung 900X3A notebook computer with a quad-core Intel Core i5-2537M CPU at 1.40 GHz and 4 GB RAM. Ubuntu 12.10 (32-bit) was used as the operating system. The Kernel version used was 3.5.0-19-generic (Stock Ubuntu Kernel). A PS Eye camera in wide angle view (blue dot) was used for image capture. OpenCV 2.3.1-11ubuntu2 (Stock Ubuntu Package) was used for image processing.

Mac OS X

Mac OS X tests have been carried out on a MacBookPro5,5 with a dual-core Intel Core 2 Duo CPU at 2.53 GHz and 4 GB RAM. Mac OS X 10.8.2 was used as the operating system. The built-in iSight camera was used for image capture. OpenCV 2.4.2 (from Mac OS X Homebrew) was used for image processing.

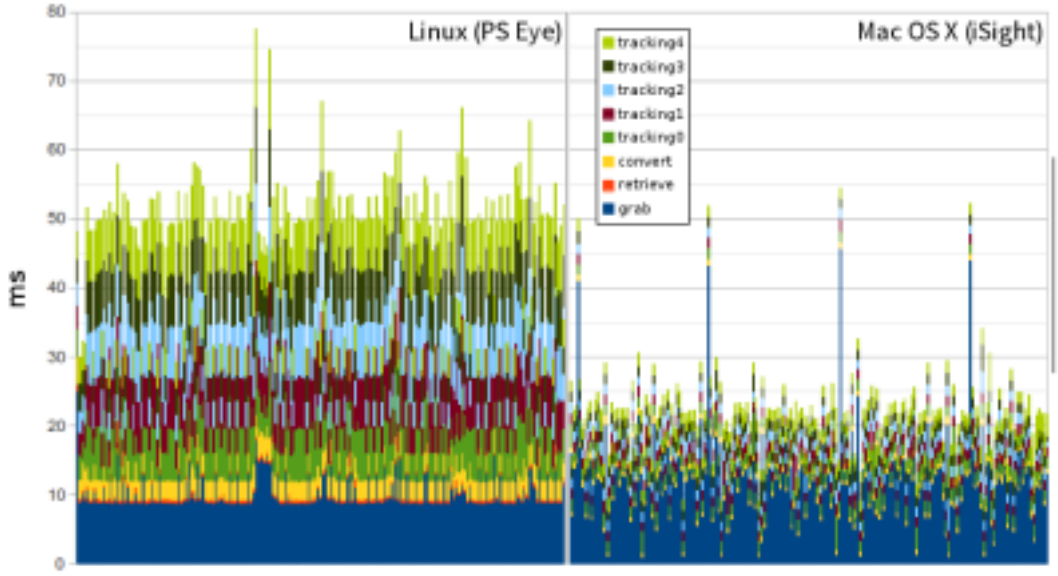


Figure 5.1: This graph shows the cumulative per-frame processing duration of 500 frames when tracking 5 PS Move Motion Controllers with PS Move API Tracker Module in horizontal mirror mode. **grab** is the OpenCV frame grab duration, **retrieve** the OpenCV frame decoding/conversion duration, **convert** is the mirroring step and **tracking0**-**tracking4** are the tracking steps for each controller.

5.2 Capture and Tracking Performance

To measure the duration and frame rate of the Tracker Module, the **test_capture_performance** utility of the PS Move API was used. It tracks all connected controllers for a duration of 500 frames and measures the time each operation takes. The results are written into a CSV file, and each captured frame is also saved for verification.

To achieve a target frame rate of 60 FPS, the whole tracking process (except for grabbing the image, which could be done in a separate thread) must complete in 16.67 ms or less.

Test Setup

For the Linux system, the PS Eye Camera was set on a 12 cm high stand, and the 5 PS Move Motion Controllers were put in front of the PS Eye camera (standing), 2 cm apart from each other and 40 cm away from the camera. The default settings of test_capture_performance were used, i.e. the Tracker uses Low Exposure mode and mirroring (to determine the processing cost of image mirroring).

For the Mac OS X system, the controllers were set placed 40 cm before the MacBook Pro's iSight camera, and the controllers were placed 2 cm apart from each other. Again, the default settings of the test utility were used.

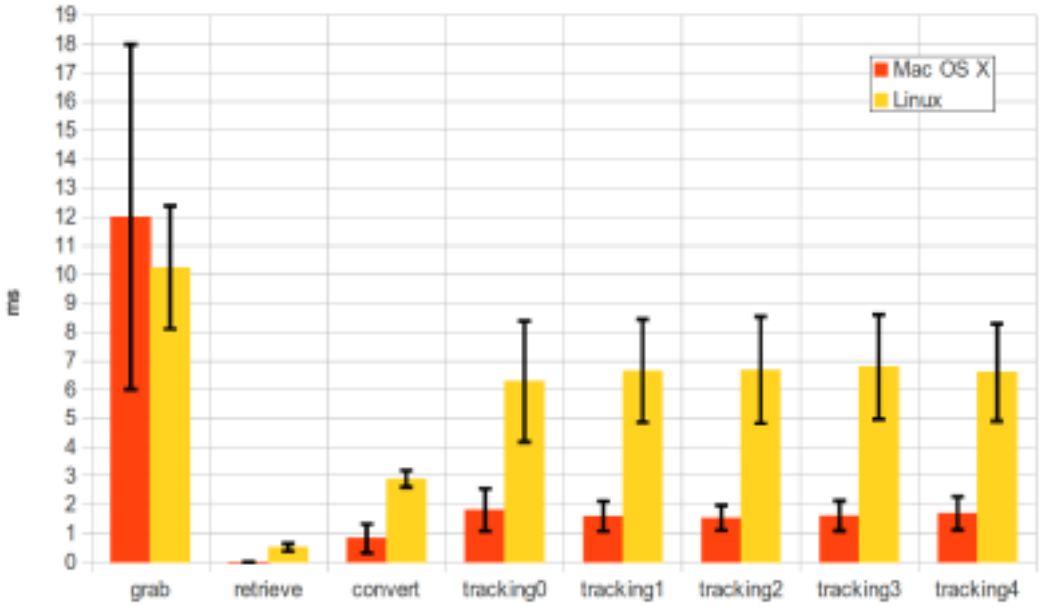


Figure 5.2: This graph shows the average duration of processing for each step of the tracker pipeline (grab, retrieve, convert are camera-related steps, tracking0-tracking4 the per-controller tracking steps). The purpose of each step is described in figure 5.1. The error bars represent the standard deviation for each step based on the 500 sample frames in the experiment.

Captured Frames

Figure 5.1 shows the cumulative frame processing duration for all 500 frames captured in the experiment, split by camera-related operations (grab, retrieve, convert) and by tracking duration for each controller (tracking0 to tracking4). As the tracker module is single-threaded, tracking additional controllers does not yet make use of multiple CPU cores. On the other hand, a single-threaded tracking process also means that all but one CPU cores are available for the user application.

Processing Steps

The average duration of each processing step is shown in figure 5.2. Getting a frame from the camera takes around 12 ms ($\pm$ 6 ms) on Mac OS X and around 10 ms ($\pm$ 2 ms) on Linux. Decoding the image (in OpenCV) less than 1 ms on Linux and 0 ms on Mac OS X - this suggests that decoding is already done in the "grab" phase on Mac OS X or that no decoding is necessary. The "convert" step (1 ms on Mac OS X, 3 ms on Linux) is the time it takes to mirror the image on the X axis - if mirroring is disabled, this step is not executed. The tracking steps are the per-controller steps for finding the controller in the camera image and updating the size and position of the controller, including all image manipulation steps. Tracking one controller takes roughly 1.5 ms ($\pm$ 1 ms) on Mac OS X and 6.5 ms ($\pm$ 2 ms) on Linux, independent of wheter

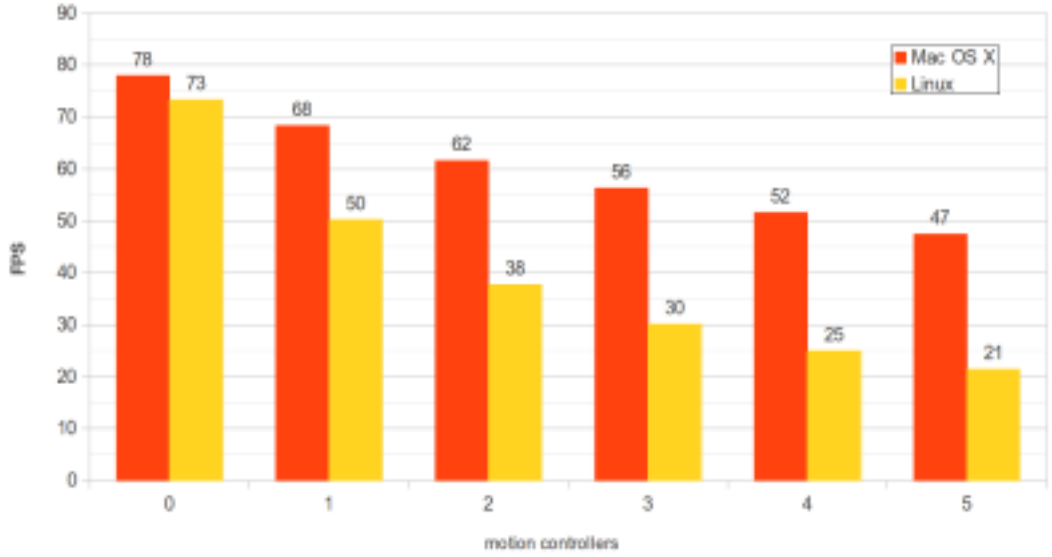


Figure 5.3: This graph shows the average frame rate values for 0-5 motion controllers on both test systems. 0 motion controllers means that only the frame capture and mirroring is taken into account (the "fixed cost"). Because controllers are tracked sequentially in a single thread, tracking more controllers will cause a reduction in frame rate.

it is currently tracked or whether recovery (see section 3.5, "Recovery after Tracking Loss") is running, due to the way the recovery procedure is designed (only one tile per frame will be searched).

Frame Rate

Based on the frame durations, the average frame rate can be calculated. In figure 5.3, the average-case frame rates are shown, based on the average duration of tracking for each of the 500 example frames from the experiment. To compare the impact of the number of tracked controllers, the duration for tracking no controllers (duration only of "grab", "retrieve" and "convert" steps - the "fixed costs" of capture) can be compared with tracking one or more controllers (the fixed costs + tracking the first N controllers).

On average, the current tracking algorithm can track a single controller with a frame rate of 50 (68) FPS, and two controllers (single-threaded) with 38 (62) FPS on Linux (Mac OS X). The differences come from the higher clock rate of the CPU in the Mac OS X machine, the newer version of OpenCV might also contribute to the better result.

The frame rate of 78 (73) FPS for tracking no controllers has to be seen as a theoretical value, as it is calculated from the capture duration of a 5-controller tracking setup. Because the tracking of 5 controllers is done sequentially, a new frame will always be available when it is requested (the new frame becomes available while the 5 controllers are still tracked). Reducing the number of controllers will cap the FPS of the no-controller tracking at the real frame rate

of the camera, because the grab operation will then have to wait for a new frame to become available, and take longer because of this waiting period.

Interpretation

One interesting aspect that can be taken away from these measurements is that when mirroring of the X axis values is desired, but the image is never displayed to the user, it is better to disable mirroring in the tracker (the default), and convert the coordinates manually ($x_{mirror} = image\_width - x$), which will save up to 3 ms per processed frame. 3 ms is a non-trivial saving in the tracking pipeline (see figure 5.2).

Another insight is that the number of controllers is important for the frame rate, especially in the current single-threaded setup with slower CPUs. As the tracking operations for each controller are independent from other controllers, these operations could be carried out in multiple threads, so that each controller is tracked in a separate thread.

The controllers have not been moved during the experiment. Also, the distance of 40 cm used in the setup means that the biggest ROI size (see section 3.5, "Region of Interest Size Calculation") is used. In situations where the distance from the camera to the controller is bigger, a smaller ROI will be used, and the per-controller tracking duration will be smaller in this case.

Comparing the Linux and Mac OS X results, it can be seen that a faster CPU can already help a lot with keeping a good frame rate even for multiple controllers. While image processing is faster on Mac OS X, obtaining the camera image is slighty slower, most likely because of the lower frame rate of the iSight.

Assuming a camera frame rate of 60 FPS, the Mac OS X machine will become CPU-bound at 3 or more controllers (56 FPS) and the Linux machine is already CPU-bound at 1 controller (50 FPS). Using one additional thread on Mac OS X would allow all 5 controllers to be processed (1 ms convert time + 1.5 ms tracking $\times$ 5 controllers = 8.5 ms, which is still below the average capture duration of 12 ms) without becoming CPU-bound (assuming zero additional cost due to multithreading overhead).

5.3 Inertial Sensor Read Performance

An important part of the quality of inertial sensor measurements is the update rate of the inertial sensors. This describes the rate at which sensor updates can be read from the controller via Bluetooth. The PS Move API distribution includes the utility **test_read_performance** that reads data from the controller with different LED update rates. Sending LED updates to the controller usually decreases the read performance, because the Bluetooth connection has to deal with more data.

Half-Frames for Accelerometer and Gyroscopes

Each update described in this section actually contains two readings for the accelerometer and the gyroscope (only one reading for the magnetometer): The most recent reading and the reading

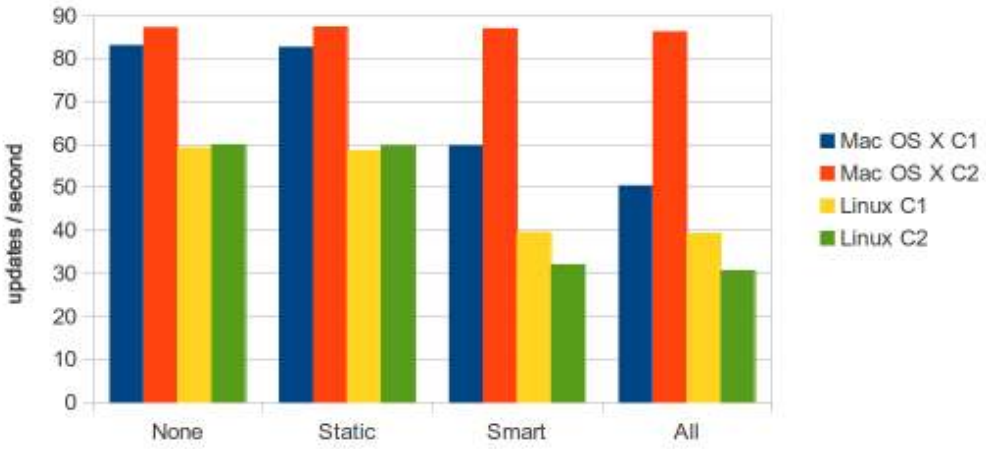


Figure 5.4: Average read performance of inertial sensors. For each of the test systems (Mac OS X and Linux), two controllers have been tested (C1 and C2), where one has been bought in late 2010 and the other one in early 2012. The graph shows the average update rate per seconds (average of 5 samples, where each sample is calculated from the time it takes to read 1000 updates from the controller).

before that. This effectively doubles the update rate measured here for the accelerometer and gyroscope, resulting in a finer-grained update rate for integrating the readings over time.

Definition of Update Methods

In this section, we use different methods for updating the LEDs of the controller. The meaning of these update methods are described here:

- **None**: No LED updates are sent to the controller. This means that the controller's sphere will be dark. This is useful in situations where visual tracking is not needed, and the LEDs should not be used as visual indicator. This gives the maximum update rate, as the Bluetooth connection can be used exclusively for reads.

- **Static:** The controller LEDs have a static color. As the controller will keep the LED lit for 4-5 seconds, the library will only send an update every 4000 milliseconds. This is useful in combination with visual tracking and sensor fusion, where the color of the controller is usually static.

- **Smart:** The controller LEDs will have their color constantly changed after every read. The built-in rate limiting of the library is enabled to avoid unnecessary updates. The current rate limit is set to 120 ms, meaning that when rate limiting is enabled, a LED update will only be sent every 120 ms, and excessive updates will be ignored by the library (not sent over Bluetooth).

- **All:** The controller LEDs will have their color constantly changed after every read. The rate limiting is disabled, which means that every LED update is sent to the controller. This might be desirable in certain situations where users want to send LED updates at a higher rate than every 120 ms, but in general will cause worse read performance results for inertial sensors.

Test Setup

For testing the performance measurements, both the Linux and the Mac OS X system were used. For each system, two runs were carried out: One with an older PS Move Motion controller bought in late 2010 and one with a newer controller bought in early 2012. Earlier experiments found that the update rate differs for different controllers on the same machine, which means that the read performance might vary between different controllers, so we used two controller per system here to have comparable results.

In each run, one controller was connected via Bluetooth to the test machine.

The **test_read_performance** was used with the default settings: Each update method (None, Static, Smart and All) is tested in 5 rounds, each round reads 1000 input reports from the controller. The update rate is then calculated from the number of read reports (1000) and the total time it took to read all input reports:

$$update\_rate = \frac{reads}{duration}$$

The 5 update rates are then averaged to a single update rate value. The standard deviation was not significant between runs - it is not shown here.

Measured Update Rate

Figure 5.4 shows the average update rates for each LED update method and OS/Controller combination. The maximum update rate is achieved by not sending any LED updates (Method "None"): Mac OS X achieves 83 and 87 updates per second, depending on the controller used. On Linux, 59 and 60 updates per second can be read from each controller.

Static LED update rates are basically the same, and do not differ significantly from not sending no LED updates. This suggests that we can use static LED updates for tracking and still achieve the best sensor update rate.

Updating the LEDs constantly, but with rate-limiting enabled shows a reduction in update rate: The Mac OS X update rate for controller 1 drops to 60 updates per second (from 83) and the Linux update rates drop to 40 and 31 updates per second (from 59 and 60).

Disabing rate-limiting and updating the LEDs constantly shows a reduction in sensor update rate for controller 1 on Mac OS X: The update rate drops to 50 updates per second (from 60 updates in the rate-limited case). On Linux, the rate-limiting does not cause significant differences, because of the way the LED updating is implemented: The LED updater thread will not buffer any updates and only send out the latest one.

Interestingly, controller 2 under Mac OS X is not affected at all by the LED updates, suggesting that for some controller on Mac OS X, the read performance is independent of LED updates.

Packet Loss

The average update rate tells us the amount of updates we get for each second, but it does not tell us the "real" update rate of the controller. With each update, the controller sends a 4-bit sequence number, which we use here to detect dropped frames. We consider a packet lost if the sequence number jumps between two reads (i.e. is not incremented by 1, modulo 16 to account for 4-bit counter wrapping). While this might not be a problem in practice, having no or little packet loss means that we read at the maximum rate that the controller sends.

In figure 5.5 the packet loss in percent is given. As an example, a packet loss of 20 percent means that for 200 packets out of the 1000 packets received, the sequence number was not an increment of 1 from the sequence number of the previous packet.

In the no-LED-updates case ("None"), Mac OS X has a packet loss of 3 percent and less than 1 percent, while Linux results show 45 and 46 percent packet loss.

As with the update rates, the static LED update case, where LEDs are only updated every 4 seconds, the results are not significantly different from the no-LED-updates case.

Comparing the continous LED updates (both rate-limited and not rate-limited) shows an increase in packet loss for Mac OS X - interestingly, the packet loss in Linux seems to improve here for the rate-limited situation. One explanation for this unexpected result is that the method of detection gives incorrect results when the distance between two updates reaches 16 frames (the 4-bit counter would then wrap around, and give the expected incremented sequence number). The results of controller 2 on Linux shows that the packet loss rate does indeed become worse with additional LED updates.

Again, controller 2 under Mac OS X seems not to be affected by the LED updates, and also exhibits a packet loss of less than 1 percent (less than 10 of 1000 frames have a non-sequential sequence number).

Interpretation

The most interesting outcome of this measurement is the behavior of controller 2 on Mac OS X. The packet loss rate of less than 1 percent suggests that for this controller, we have reached the hardware limit of updates, and this limit seems to be around 87 updates / second. Also, for this controller, the update rate seems to be unaffected by the LED updates.

For all other controllers, the update rate is indeed affected by the LED update rate, so rate-limiting the updates usually helps to keep a good sensor update rate. In general, sending a LED update every 4 seconds (the default for vision tracking in PS Move API) does not affect the sensor update rate.

Application developers wanting to make use of the LEDs for output or for effects should be aware of these technical limitations if they depend on a good update rate of the inertial sensors. For generic use cases, the built-in rate-limiting of the PS Move API will avoid problems with the update rate, even if the application developer does not take care of limiting their updates.

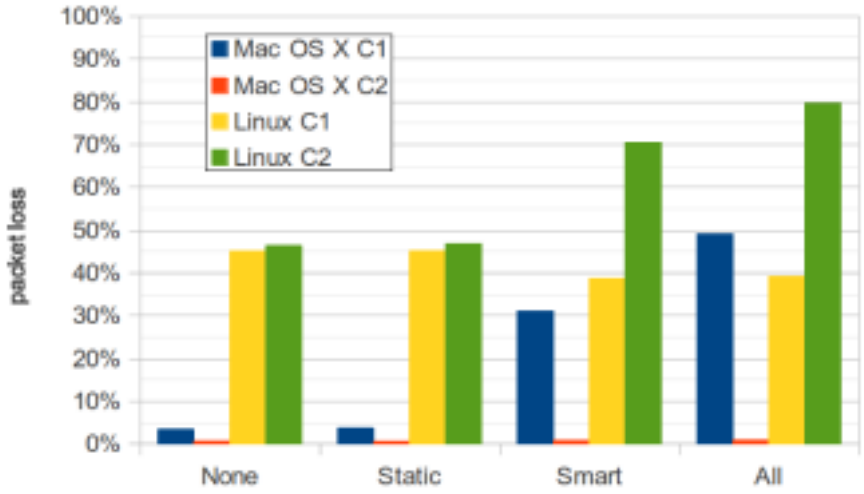


Figure 5.5: Packet loss in percent of total reads (1000) for the inertial sensor read performance tests. The setup is the same as in figure 5.4. A "packet loss" here means that the internal sequence number sent by the controller between two subsequent reads differs by a value greater than 1 (this might not be a real packet loss, depending on how the sequence number is calculated in the controller).

5.4 End-to-End System Latency

Without additional (and costly) equipment, testing the latency of the camera frame grabbing is very hard to do reliably. Instead of testing the latency of the camera, the approach chosen here is to measure the end-to-end latency of the whole tracking system, i.e. the time from switching on the LEDs of the controller to getting a tracking result. These measurements give us an idea how big the latency of the whole system is. Making assumptions about the relation of latencies of the system parts, we can come up with an idea on how big the latency of the camera and algorithm is.

This test uses the **test_end2end_latency** to measure the latency in tracking a controller after switching on the LEDs.

Workflow

This experiment measures time and accuracy. The exact flow is implemented in the test utility, see there for details.

1. Switch the controller LEDs on
2. Wait until the controller is tracked, save the timestamp as initial timestamp
3. Wait until the tracking becomes stable, save the timestamp and the position/radius of the controller

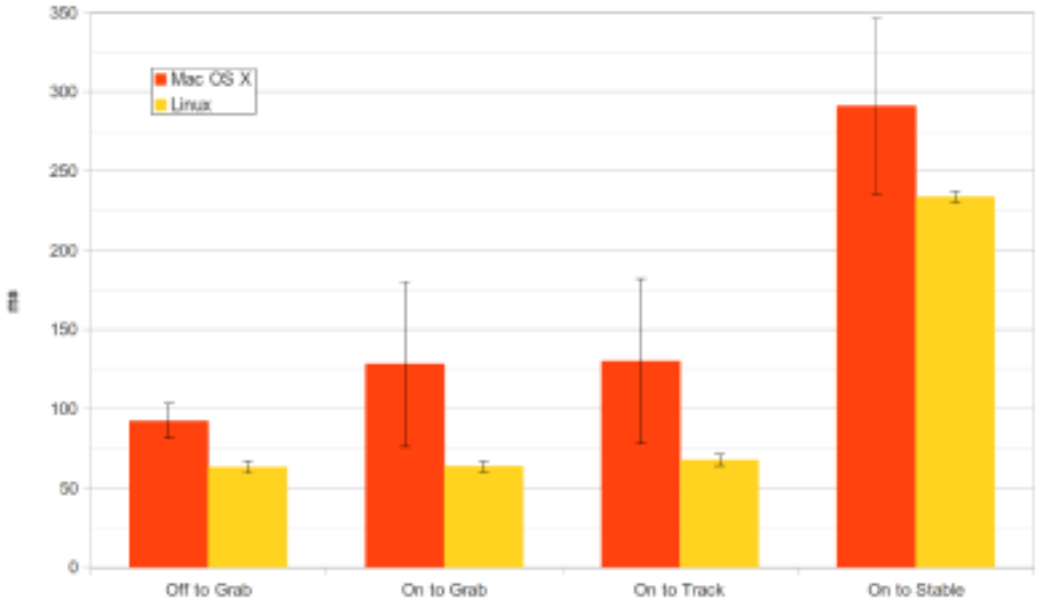


Figure 5.6: End-to-End System Latency: In this experiment, the latency from switching the LEDs off and on to grabbing the first frame, tracking the first frame and tracking the stable frame has been measured and compared for the Mac OS X (iSight) and Linux (PS Eye) machines.

4. Switch the controller LEDs off, save the timestamp

5. Wait until the tracking is lost, save the capture timestamps (including grab and tracking time)

6. Wait until the tracking loss becomes stable (tracking is lost for several frames in a row), save the timestamp

7. Switch the controller LEDs on, save the timestamp

8. Wait until the tracking is recovered, save the capture timestamps (including grab and tracking time)

9. Wait until the tracking becomes stable (tracking is kept for several frames in a row), save the timestamp and position/radius of the controller

10. Calculate the difference of position and radius obtained in step 3 and step 9

The position/radius difference calculation has been put in place to verify that the tracking recovery becomes stable, and that the pixel error is not significant (see figure 5.7). From these measurements, we can calculate different time periods.

Definition of Time Periods

In this experiment, we save the timestamp at different points in time. For the results presented in figure 5.6, we measure the following periods:

- **Off to Grab:** The time from switching the LEDs off to the time the first frame is grabbed where the tracking is lost.
- **On to Grab:** The time from switching the LEDs on to the time the first frame is grabbed where the tracking is recovered.
- **On to Track:** The time from switching the LEDs on to the time the first frame is processed (tracked) where the tracking is recovered.
- **On to Stable:** The time from switching the LEDs on to the time the frame is processed (tracked) where the tracking is stable (position and radius have converged to their old values).

The old value of the position and radius is known because we measure and save it before turning the LEDs off (step 3 in the workflow).

Test Setup

This test is carried out both on the Linux system with a PS Eye camera and the Mac OS X system with the built-in iSight camera. In both cases, a single controller is connected via Bluetooth and placed in front of the camera (centered) and 40 cm away from the camera.

The test utility **test_end2end_latency** is used with default settings (100 iterations, low exposure mode).

Measurements

Figure 5.6 shows the measurements taken. In general, Mac OS X has a higher latency here, even though the machine has a faster CPU. The system latency seems to be more bound to the camera than to the CPU performance in this case (the PS Eye must have a shorter latency than the iSight used on the MacBook Pro used in the test).

Switching the controller off and grabbing the image where this is detected takes 92 ms ($\pm$ 11 ms) on Mac OS X and 63 ms ($\pm$ 4 ms) on Linux. Because losing tracking is a zero-cost operation (compared to doing tracking recovery), switching the controller on and grabbing the first frame where it is detected again takes slightly longer: 128 ms ($\pm$ 52 ms) on Mac OS X and 64 ms ($\pm$ 3 ms) on Linux. Tracking this new frame (which is required to measure and obtain the position and radius) takes an additional 2 ms on Mac OS X and an additional 4 ms on Linux - this additional latency is introduced by the tracking algorithm.

Finally, getting a stable position (sometimes when tracking is found, the tracker takes some additional frames to correct and filter the position and radius information) takes 290 ms ($\pm$ 55 ms) on Mac OS X and 233 ms ($\pm$ 3 ms) on Linux. This latency is caused mostly by the tracking algorithm - it gives an idea on how long it takes to get a reliable position information after tracking is lost and then recovered.

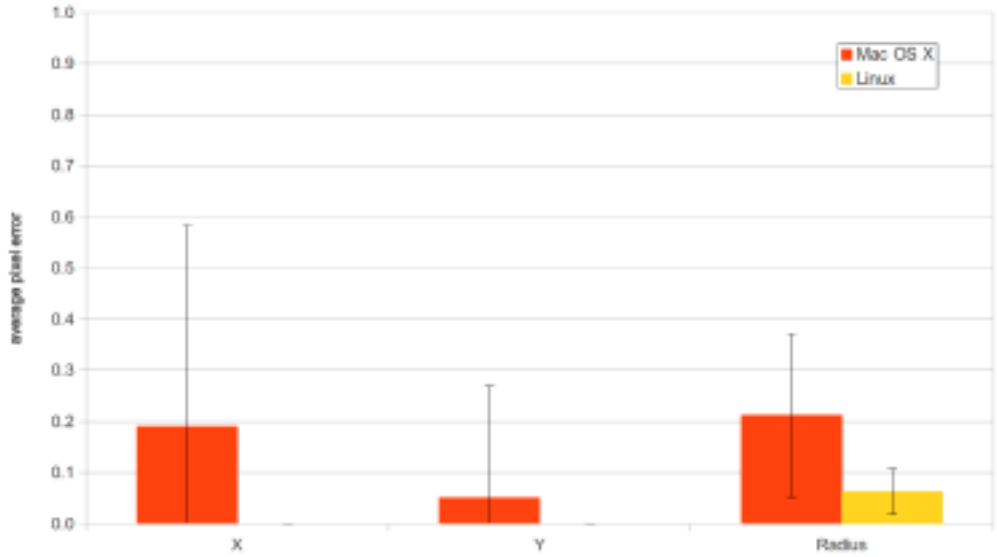


Figure 5.7: Average Pixel Error in the End-to-End System Latency test. These measurements show the average pixel error when the measurements become stable ("On to Stable"). The error is defined as the absolute difference between the initial tracking position and size and the position and size after tracking recovery. These values are well below 1 pixel, so the "On to Stable" measure is correct for tracking.

Interpretation

We can assume that the "Off to Grab" measure in figure 5.6 gives a good estimate of the combined LED update and camera latency. This latency is mostly hardware-related, and cannot be overcome without using different hardware for the controller or the camera (it is assumed that sending the LED update and reading from the camera takes up a small amount of this latency).

Comparing the "On to Grab" results with the "Off to Grab" results describes the additional latency we get through the recovery algorithm. Adding to this the cost of tracking the controller in the camera gives the "On to Track" measure.

Finally, a measure for explaining how long it takes for the controller to report a stable position and radius after tracking loss can be obtained from the "On to Stable" measurements. On Linux with the PS Eye camera, tracking can be recovered from tracking loss in about 233 ms (in real-world situations, the delay from switching on the LEDs would be eliminated, because the LEDs are always on during tracking, even when tracking is lost).

5.5 Performance Impact of ROI Size

For tracking the controller, image manipulations have to be carried out on the camera image to reduce noise, convert the image from RGB/BGR colorspace to HSV colorspace and to threshold the image to desired color hue values. A naive implementation would carry out these operations

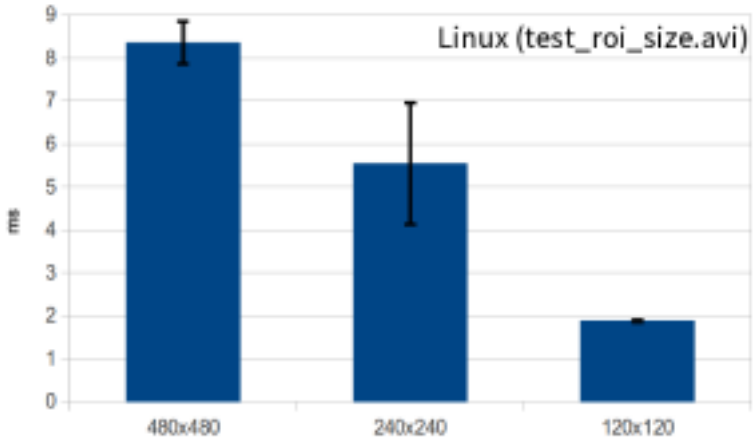


Figure 5.8: Average tracking duration in ms for tracking a single controller using different fixed ROI sizes in **test_roi_sizes.avi** (first 500 frames) on the Linux machine

on the whole image, which would be very processing intensive. To avoid this, we use a region of interest (ROI) to carry out these image manipulations, which reduces the size of the image data that needs to be processed (see section 3.5, "Region of Interest Size Calculation" for details on how this is implemented).

Tradeoffs

There is a tradeoff between fast sphere tracking and tracking stability - when the region of interest is too small, tracking is fast, but when the controller moves, it will be outside of the region of interest, causing a tracking loss and usage of the recovery procedure. If the region is too big, tracking will be slower, but even small movement might not move the controller outside of the region of interest, thereby avoiding unnecessary usage of the recovery procedure. The PS Move API implementation deals with this by using different ROI sizes depending on how big the sphere appears in the camera image (and as the sphere size corresponds to the distance, the ROI will become smaller when the controller moves farther away).

Test Setup

To test the tracking performance with different ROI sizes, a video of a single controller was captured (using the **test_record_video** utility included in the PS Move API source distribution). This video has been used for all tests to make them comparable. The video is available for download on the MoveOnPC downloads page[1].

The test utility used is **test_roi_size**, which again is included in the PS Move API distribution. This utility expects one controller to be connected (although it won't be used for tracking,

[1] http://code.google.com/p/moveonpc/downloads/detail?name=test_roi_size.avi, retrieved 2012-12-03

it's just used as a dummy controller for the library). It also expects the .avi file to be present in the current working directory.

The ROI sizes that were tested were 480x480, 240x240 and 120x120 (the camera image was 640x480). The Linux system was used to carry out the tests, and the video has been recorded using a PS Eye camera. 500 frames were processed and the measurements (tracking duration and position for each ROI size) stored.

Captured Data

The test utility will write a CSV file "roi_size.csv" to the current directory, as well as screenshots for every 50th frame for every ROI size. The screenshots can be used to verify and compare the results.

Measurements

The average tracking duration in milliseconds for each ROI size is shown in figure 5.8. The error bars represent the standard deviation based on the 500 sample frames analyzed. The ROI size of 480x480 pixels has an average tracking duration of 8.3 ms ($\pm$ 0.5 ms), the ROI size of 240x240 pixels an average duration of 5.5 ms ($\pm$ 1.4 ms) and the ROI size of 120x120 pixels an average duration of 1.9 ms ($\pm$ 0 ms).

Interpretation

A reduction of the ROI size by half reduces the processing overhead by 34 percent when going from 480x480 to 240x240 and by 65 percent when going from 240x240 to 120x120. This saving in processing time can be used to track additional controllers or allow more CPU time for the user application.

Using dynamic ROI sizes (as is implemented in the PS Move API Tracker) will help gain both the benefits of big ROI sizes when required, as well as not having an impact on performance when the controllers are far away.

5.6 Sphere Detection in Motion Blur Situations

With the PS Eye camera, we experienced visual tracking problems caused by motion blur with green colors (figure 5.9). While other colors such as magenta (red and blue channels at full intensity) were tracked without problems even in situations with fast movements, green does not work so well with this camera. For this reason, green is to be avoided with the current hue-based tracking algorithm when using the PS Eye camera.

Eliminating Motion Blur Errors

Improvements to the tracking algorithm could improve the situation. As the motion blur usually happens while the controller is moved quickly, and this only happens for short amounts of time, using the One Euro Filter [4] for the radius values could mitigate the tracking errors.

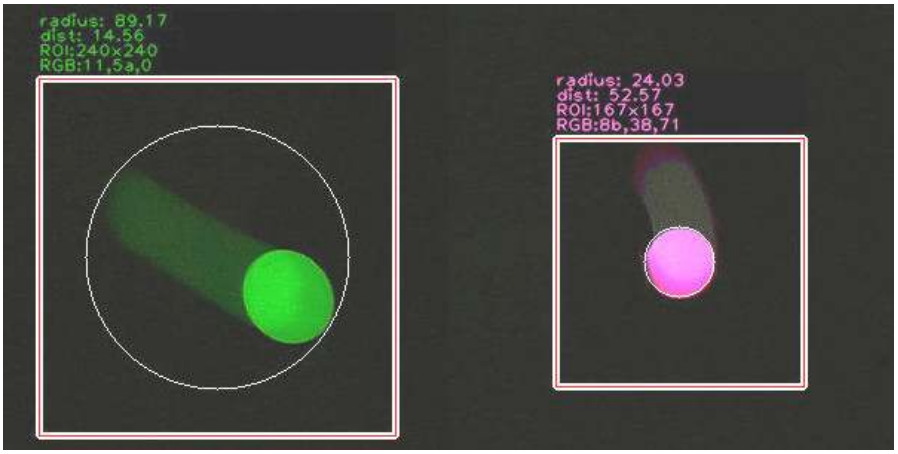


Figure 5.9: Motion blur from the PS Eye camera: Even in low exposure mode, green (left) produces motion blur causing wrong size detection, while magenta (right) can be tracked well. Also note the deforming of the green sphere, while the magenta sphere is still a circle.

Also, additional checks could be built into the algorithm to detect motion blur situations such as the one in figure 5.9.

5.7 Example Applications

In this section, some of the example applications that have been developed for testing and demonstrating the PS Move API libraries are presented.

Sensor Filter / Accelerometer Visualization

- **Input data:** Accelerometer
- **Used modules:** Core, Calibration
- **Dependencies:** Qt 4
- **Usage scenario:** Testing calibration, filter response

The sensor filter application (figure 5.10) uses one motion controller connected via Bluetooth, and reads the calibrated accelerometer values from it. The accelometer readings are displayed visually, and color-coded for each axis.

This example is a very barebones application, but helps explaining the functions of an accelerometer in a visual way. It also shows how to integrate the PS Move API in a C++ Qt application without the need for the Qt bindings.

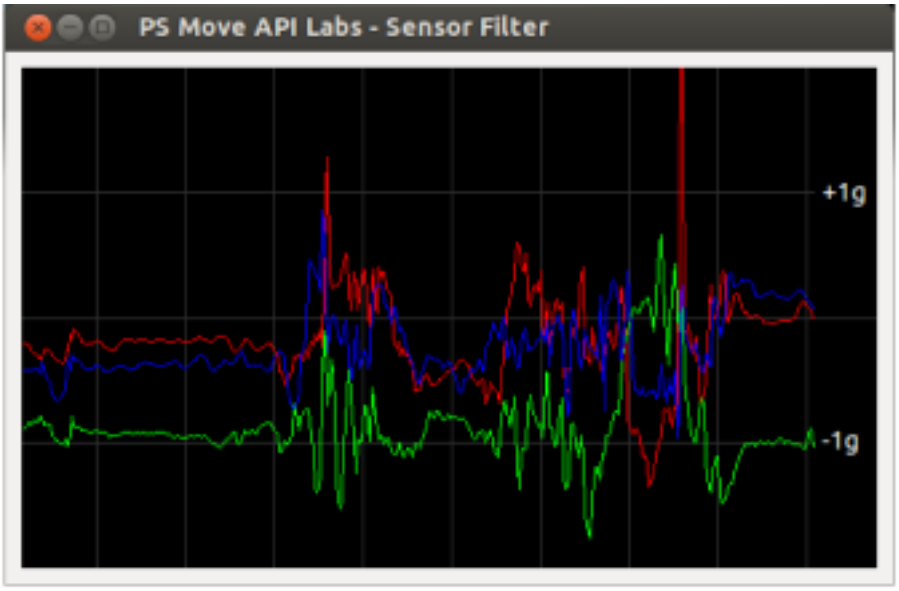


Figure 5.10: Sensor Filter Example

Paint

- **Input data:** Buttons, Vision Position
- **Used modules:** Core, Tracker
- **Dependencies:** Qt 4
- **Usage scenario:** Interactive drawing

Drawing on the screen is one obvious use case for the vision tracker application. The Paint example application (figure 5.11) can use one or more controllers connected via Bluetooth, and tracks them using the PS Eye camera. Pressing the trigger button draws lines.

Additional features such as choosing the color for drawing are also provided, as well as saving the current drawing and restoring it at some point in the future. This application also shows how to display the camera image inside a Qt-based application.

3D Orientation Visualizer

- **Input data:** Accelerometer, Gyroscope, Magnetometer and Buttons
- **Used modules:** Core, Calibration, Orientation
- **Dependencies:** Qt 4, OpenGL
- **Usage scenario:** Testing orientation algorithm, OpenGL integration

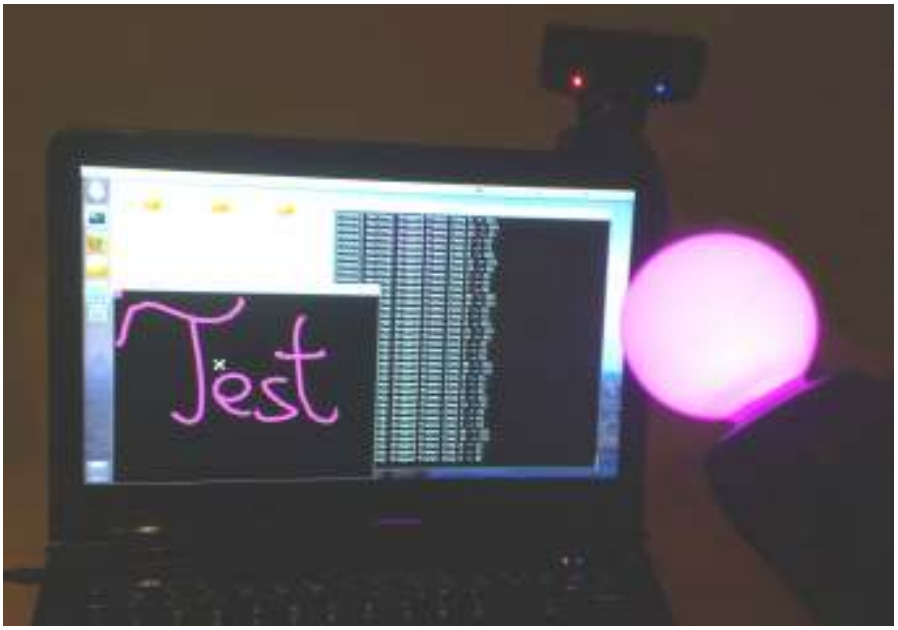


Figure 5.11: Paint Application

To test the orientation filter and show the obtained rotation values, the 3D Orientation Visualizer (figure 5.12) can be used. It displays the current orientation in a 3D environment, and optionally shows a simple 3D model of a controllers that reacts to button and trigger events.

Advanced features of this application include the option to let the OpenGL camera fly around the world origin and to use the orientation information to control the camera viewing direction.

X11 Mouse Emulator

- **Input data:** Gyroscope and Buttons
- **Used modules:** Core, Calibration
- **Dependencies:** xdo
- **Usage scenario:** Mouse / input device replacement

The X11 Mouse Emulator uses the gyroscope of a single motion controller as well as the buttons on the controller to emulate the mouse on Linux and Mac OS X systems. The source code is kept as generic as possible, so the buttons can be re-configured to emulate keyboard presses or mouse buttons (or a combination thereof).

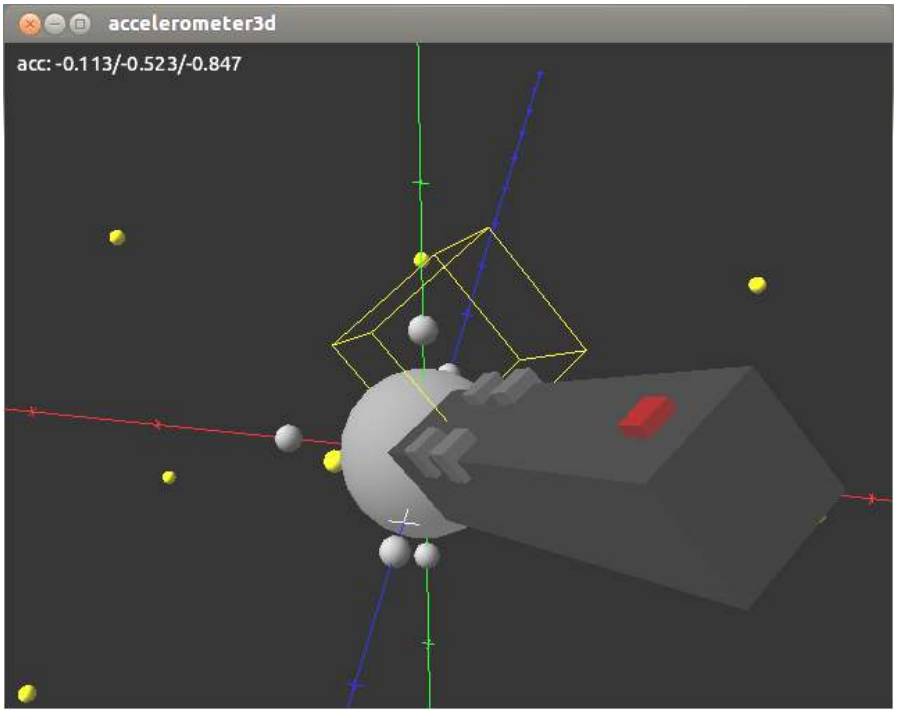


Figure 5.12: 3D Orientation Visualizer Application

Interactive Whiteboard

- **Input data:** Vision Position and Buttons
- **Used modules:** Core, Tracker
- **Dependencies:** Qt 4
- **Usage scenario:** Low-cost interactive whiteboard solution

Another interesting use of the vision module is to map camera image coordinates to screen coordinates (with the camera pointed towards the screen). A 4-point calibration (each corner of the screen) has to be done before the application can be used. After the setup, the user can draw on the screen as if it was a real interactive whiteboard.

An example photo of the application in use is shown in figure 5.13.

Multi-Touch Rotate/Zoom Example

- **Input data:** Vision Position and Buttons
- **Used modules:** Core, Tracker
- **Dependencies:** Qt 4

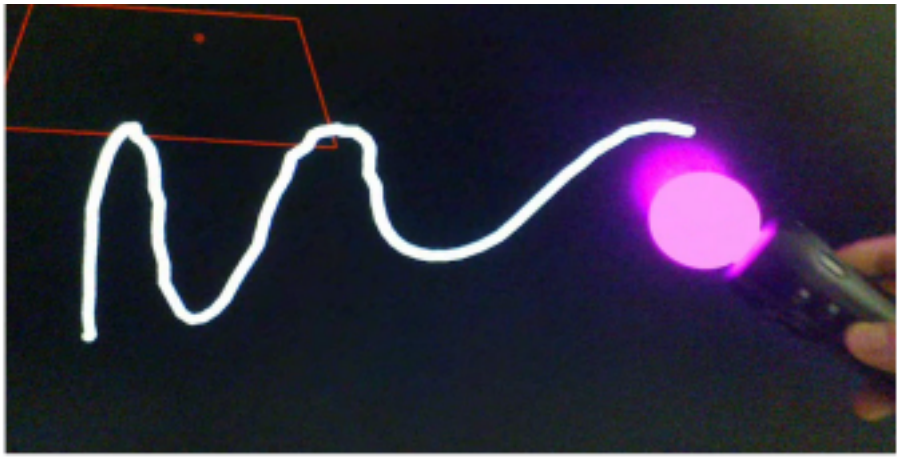


Figure 5.13: Interactive Whiteboard Application

- **Usage scenario:** Multi-cursor interaction

The Multi-Touch Rotate/Zoom example application emulates the function of an image viewer as seen in multi-touch table applications. Instead of touching the windows, colored dots representing the controller images are shown on screen, the trigger button can be used to "grab" the on-screen objects at the position of the dot.

Sensor Fusion: Augmented 3D Paint

- **Input data:** Vision Position, Orientation (Accelerometer, Gyroscope and Magnetometer) and Buttons
- **Used modules:** Core, Tracker, Fusion
- **Dependencies:** OpenGL, GLUT, SDL
- **Usage scenario:** Augmented reality application, sensor fusion

As an example for augmented reality 3D rendering, this application implements a simple 3D painting application using one or more motion controllers. Figure 5.14 shows the application in action while drawing 3D boxes in space.

The drawn objects can be translated in 3D space with the buttons, as well as rotated on the Y axis (rotation based on the controller orientation could also be implemented, but is not available in this example).

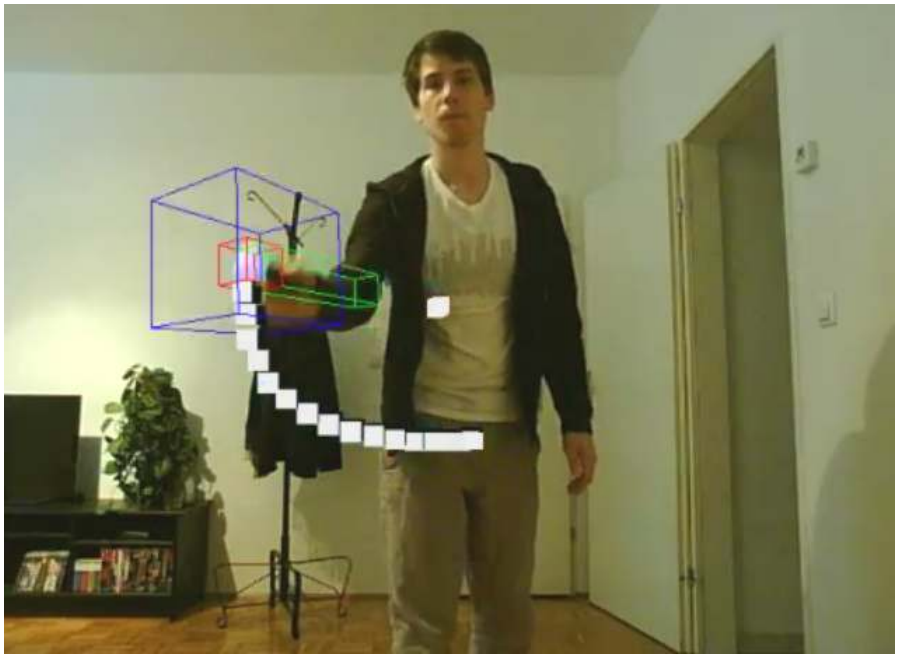


Figure 5.14: 3D Paint Sensor Fusion Application

Sensor Fusion: Augmented 3D Particles

- **Input data:** Vision Position, Orientation (Accelerometer, Gyroscope and Magnetometer) and Buttons
- **Used modules:** Core, Tracker, Fusion
- **Dependencies:** OpenGL, GLUT, SDL
- **Usage scenario:** Augmented reality application, sensor fusion

Building on top of the 3D Paint Sensor Fusion example, this application implements physical particles that can be emitted from and attracted by a (possibly different) controller (depending on which button is pressed). Figure 5.15 shows an example of this in action - the controller on the left attracts particles, while the controller on the right emits new particles.

5.8 Integration with Other Frameworks

While the PS Move API itself provides a very easy way to interface with existing code, some frameworks and protocols exist that are used by existing applications for augmented reality (AR) or virtual reality (VR) use cases. Interfacing with these frameworks allows users to use the PS Move API without having to write special integration code or linking the application directly with the library.

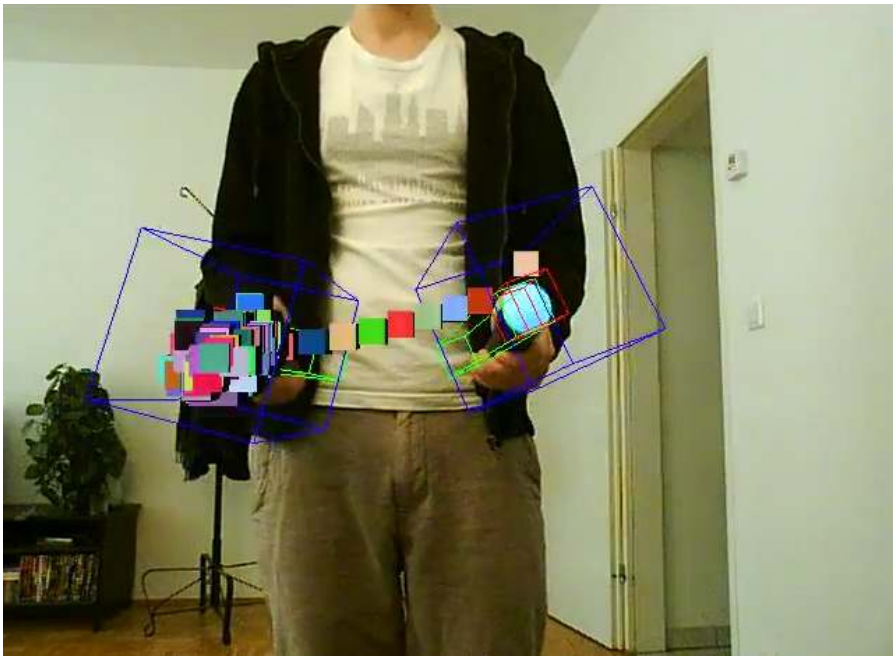


Figure 5.15: 3D Particles Sensor Fusion Application

TUIO Input Bridge

TUIO [17] is a simple protocol initially used for multi-touch interfaces. The "Interactive Whiteboard" application described in section 5.7 already works similar to a multi-touch surface, but instead of detecting finger presses directly using one of the common multi-touch-table technologies, the PS Eye camera is used to create a mapping from the camera frame to the surface, and the sphere on the PS Move Motion Controller is used as "cursor".

The TUIO Input Bridge is available as **examples/c/tuio_server.cpp** in the PS Move API source distribution, and depends on the TUIO reference implementation in external/TUIO_CPP/, which has to be built as static library.

Once built and started, the TUIO Input Bridge will enumerate all PS Move Motion Controllers available to the library, and try to calibrate them using the tracker. By default, the application is configured so that TUIO touch points are only sent when the T (trigger) button is pressed on the controller.

As the generic TUIO "2Dcur" profile doesn't allow for button events, the TUIO Input Bridge also contains support for a simple line-based TCP protocol that can be used to send additional button events and information about tracking (when a controller's tracking position gets lost, etc..) and the 3D position (distance from the camera) - in this case, the developers using the library will still have to add support for these additional events, but can reuse existing support for the TUIO protocol.

OpenTracker Module

OpenTracker [27] is an open source framework for integrating different input sources used for tracking the position, orientation and button states of input controllers into a single library. Input modules provide additional data sources that can be configured in an XML file. Output modules provide sinks into which data from input modules can be written. Between input and output modules, OpenTracker can convert and modify the data to make input data suitable for a given application without having to modify either the input or output module.

As the PS Move API library provides access to the Motion Controller's state, and the PS Move Tracker library can determine the 3D position of one or more Move controllers in space, it makes sense to add support for the PS Move Controller to OpenTracker. To this end, a patch for OpenTracker has been written that adds support for using the PS Move as an input device. The module and the patch, including an example OpenTracker XML configuration can be found in the directory **bindings/opentracker/** in the PS Move API source distribution.

PSMoveModule

The PS Move Module (**PSMoveModule.cxx** and **PSMoveModule.h**) works as a central point for connecting to PS Move Motion Controllers. It will create new instances of **PSMoveSource** (see below) and it will also manage the single instance of the PS Move tracker module, as only one tracker must be used even in the case of multiple controllers. A controller is obtained by calling **PSMoveModule::createNode()** and specifying the controller ID as zero-based index in the attributes. The other important function in this module is **PSMoveModule::pushEvent()** which will update the camera image and tracking information for each controller, and then update the state of every single controller (buttons / inertial sensors) by passing the request to **PSMoveSource::getEvent()**.

PSMoveSource

The PS Move Source (**PSMoveSource.cxx** and **PSMoveModule.h**) represents the event source of a single controller. It gets instantiated by the **PSMoveModule** (see above) and gets a reference to the PS Move Tracker instance passed in its constructor. The most important function in this class is **PSMoveSource::getEvent()**, which carries out the following tasks:

1. Update the LEDs of the controller to keep it lit
2. Read the latest input reports from the controller using **psmove_poll()**
3. Save the trigger and button states of the controller in the event object
4. Get the 3D position from the tracker
5. Save the 3D position in the event object
6. Get the orientation (3D rotation) as quaternion from the library
7. Save the orientation in the event object

The PS Move Source explicitly doesn't update the camera image or per-controller tracking, as this is taken care of centrally in the PS Move Module.

Changes required in OpenTracker

The necessary changes to OpenTracker (as shipped in the VRUE2011 package) are available as source-code patch in the PS Move API source distribution as **bindings/opentracker/opentracker_vrue2011.patch**.

- **CMakeLists.txt** - An option "OT_USE_PSMOVE" has to be added so that users can enable and/or disable the PS Move feature.

- **src/misc/OpenTracker.cxx** - The **PSMoveModule** has to be registered to the OpenTracker core, so that it can be found at runtime, using "OT_REGISTER_MODULE()".

- **opentracker/src/CMakeLists.txt** - Include directories and the linking information has to be set, so that header files and the shared library of PS Move API can be found by the compiler and linker. We are using the pkgconfig facility here, and assume a system-wide installation of the PS Move API. Also, the "USE_PSMOVE" define must be set here, so that the PS Move-specific code is compiled.

Example OpenTracker configuration

A very simple configuration that connects to a single PS Move Motion controller and prints the tracking information on the console looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE OpenTracker SYSTEM "opentracker.dtd">
<OpenTracker>
  <configuration>
    <ConsoleConfig headerline="My Output" interval="1"/>
    <PSMoveConfig />
  </configuration>

  <ConsoleSink comment="PSMove Controller" active="on">
    <PSMoveSource controller="0"/>
  </ConsoleSink>
</OpenTracker>
```

To read from more than one controller, add another **PSMoveSource** element to the XML file, and supply a different (zero-based) controller index in the "controller" XML attribute.

5.9 Performance and Limits Summary

This section gives an overview of the performance measured in the previous sections, as well as built-in limits of the current implementation. The hardware and software setups used for these measurements are described in section 5.1.

- **Vision Tracker Frame Rate** (tracking 1 controller)
  - **Linux (PS Eye):** 50 FPS
  - **Mac OS X (iSight):** 68 FPS
- **End-to-End System Latency** (LED setting to initial tracking)
  - **Linux (PS Eye):** 68 ms ($\pm$ 3 ms)
  - **Mac OS X (iSight):** 130 ms ($\pm$ 52 ms)
- **Maximum HID Update Rate** (inertial sensors, trigger and buttons)
  - **Linux:** 60 updates/second
  - **Mac OS X:** 87 updates/second
  - **Hardware Limit:** ~87 updates/second
- **Tracker Module Controller Limit:** 5 controllers
- **Per-Host Bluetooth Controller Limit:** up to 9 (depending on adapter)

The following limits have been tested in working conditions, depending on the environment and hardware used it might be possible to use even more controllers simultaneously:

- **Maximum Simultaneous Controllers in a Single Room:** 21 (3 laptops, Move Daemon)
- **Maximum Bluetooth Distance:** up to 15 m (depending on adapter and environment)

CHAPTER 6

Summary and Future Work

This chapter gives a quick overview of the implemented features, as well as a list of open issues that have not been resolved. In closing, an overview of future work is given, which extends upon the scope of this thesis.

6.1 Implemented Features

This section gives an overview of the features that are implemented in the PS Move API 3.0 release:

PS Move Core Library

The core library deals with controller communication and reading and writing data from/to the controller. The LEDs and inertial sensors are handled by the core library, including further processing of the data.

- **USB pairing** - Pairing works reliably on Mac OS X and Linux, the Windows implementation works unreliably due to Windows issues
- **LED setting** - Setting the controller LEDs works reliably and fast on Mac OS X an Windows when paired, and also works reliably on Linux using the workarounds (multithreading) implemented for Linux
- **Calibration reading** - Reading the calibration data and storing it on disk is implemented as part of the USB pairing process
- **Button and sensor reading** - Buttons and sensors (raw and calibrated) can be read by the library and reported to the user app
- **Orientation tracking** - The orientation of the controller can be obtained as quaternion describing rotation in 3D space

PS Move Tracker Library

The tracker library deals with OpenCV integration, vision tracking of the lit sphere in the camera frame, including further processing of the data.

- **PS Eye support** - PS Eye support works natively in Linux, and on Windows with the CLEye driver. On Mac OS X, PS Eye support is not yet available due to missing 64-bit camera drivers.
- **Exposure setting** - The exposure of the PS Eye can be reliably controlled on Linux and Windows. On Mac OS X, a workaround for the built-in iSight camera exists.
- **Multi-controller tracking** - This has been tested with up to 5 controllers with a 1080i HD camera. In general, the PS Eye camera can also track up to 5 controllers without problems, the limitation being the number of different colors that can be detected in a reliable way.
- **Camera image mirroring** - The camera image can be mirrored. All tracking data will be mirrored accordingly (including the part in the sensor fusion algorithm), which is useful for situations where the camera has the same direction as the screen.
- **HD camera support** - The library works with any camera resolution, and has been tested with 1080i and 1080p input. For 1080i input, the library includes runtime support for deinterlacing the image using line doubling.
- **Distance calibration/reporting** - The distance of the sphere from the camera can be calculated using the radius. Calibration can be carried out, upon which the distance calibration is based. For the PS Eye, the distance calibration is built in.
- **Sensor fusion** - The orientation (from the core library) and position (from the tracker library) can be combined to render 3D data on top of the controller in the camera frame.

Language and Framework Bindings

The PS Move API can be used from different target languages. The library itself is implemented in C, and the default language is also C. Bindings and examples for more languages exist for both the core library and the tracker library.

- **Python** - A CPython extension module. Tested with Python 2.7.
- **Java** - A JNI module and .jar library. Tested with Java SE 1.6.0.
- **Processing** - Based on the Java bindings. Tested with Processing 1.5.1 in Mac OS X 10.8.
- **C#** - A C# library using P/Invoke. Tested with Mono 2.10.8.1 on Ubuntu 12.10.
- **OpenTracker** - An input module for the OpenTracker Framework.
- **TUIO** - A server broadcasting TUIO 1.1 2Dcur messages.

6.2 Discussion of Open Issues

This section gives an overview of issues that have not been solved, and cannot be solved without modifications in the target platforms, operating system kernels, middleware or dependency libraries. Where applicable, ideas for solutions to the problems are given.

HID Write Performance on Linux

Compared to Mac OS X and Linux, the Bluetooth HID stack on Linux has some delay[1] when writing LED and rumble information out to the controller. In practice, this means that the program (and consequently tracking) would be stopped for a few milliseconds every time the LEDs are set.

We currently work around this problem by making intelligent decisions about when to send LED updates and when to avoid them (e.g. if the LED color is not changed, it does not make sense to send an update to the controller - the controller will keep the LED on for a few seconds after the last update). Similarly, a rate-limiting feature has been implemented so that excessive updates that would not be visible to the user are not sent to the controller.

In addition to making sure not to send unnecessary updates, the Linux implementation of PS Move API runs the LED setting in a separate thread, thereby avoiding blocking the main application.

Ideally, the Linux Bluetooth stack would be fixed, so that the updates are sent as quickly as on Mac OS X and Windows. The performance tests have been carried out on the same hardware on Mac OS X and Linux, so the problem is software-related and has nothing to do with the hardware used.

Motion Blur with Different Channels when Using the PS Eye Camera

In the tests, the green color channel of the PS Eye camera image caused more motion blur for fast movements than the red and blue channels (see section 5.6). In practice, this means that a green-only controller color is to be avoided when using the PS Eye camera, as it will lead to worse tracking results.

This issue is mostly a hardware limitation of the PS Eye camera sensor, and can probably not be solved in software easily. When using higher-quality cameras, this problem usually does not appear at all, so it can also be solved by using different hardware.

Unused / Unknown Fields in the Calibration Data

We currently do not parse all available information from the calibration blob that is retrieved from the controller via USB during the pairing process. The reason for this is that the format and contents of the calibration data is not documented, and the documentation on the MoveOnPC wiki pages is not complete.

Being able to utilize more or all of the calibration data might improve the sensor-based tracking performance, because more calibration data can be used to interpret the raw sensor

[1] http://moveonpc.blogspot.com/2012/06/multi-threaded-led-writing.html, retrieved 2012-12-08

readings. Still, with the calibration data that we currently use, the accelerometer and gyroscope calibration already works quite well for the common use cases.

Fixing this issue would require some analysis of existing calibration data from multiple PS Move Motion Controllers, or documentation of the format from the hardware vendor (Sony) and usage of the additional calibration data in the library.

Bluetooth Pairing on Windows

Pairing a PS Move Motion Controller to a Windows (7/8) computer does not work reliably. There are several guides online on how to do the pairing with some tricks (pressing the PS button at the right times, timing the clicks in the UI "just right", etc...), but in general it's a trial-and-error procedure, and it might not work at all in some cases.

Research into existing solutions (e.g. for the Sixaxis controller, which has similar connection problems via Bluetooth) suggests that it is not possible to gain full control over the Windows Bluetooth stack without writing a complete replacement driver (which is out of scope for this project, and might bring other problems - e.g. when users want to use Bluetooth audio devices together with the Motion Controller).

This problem might not be solvable without support from Microsoft. One possibility for making it a bit easier to do the pairing without help from Microsoft would be to analyze the changes in the Windows Registry before and after a controller has been successfully paired. Such data already exists, and there has been a discussion[2] on the mailing list in June 2012 about the registry keys that get updated. Further work on this might reveal a solution in which the Windows registry is updated manually, making pairing work reliably. Similar "hacks" are already employed for Mac OS X (10.7 and later) and Linux, where the controller's Bluetooth address is injected into the system's Bluetooth stack configuration during pairing.

Exposure Setting on Mac OS X

As described in section 3.7, the blinking calibration on Mac OS X does not work as cleanly as on other systems, because the exposure cannot be set manually. The only possible way to avoid having continous auto-exposure on Mac OS X is to "lock" the exposure, which means that exposure adjustments happen only once when the camera device is opened.

This in turn means that the user has to move the glowing sphere in front of the iSight camera whenever an application starts in order for the exposure to be at a very low level suitable for tracking a brightly-lit controller.

Fixing this issue would require interfacing with the camera on a low level (again, the OS X-provided libraries do not provide this level of control), or interfacing with the PS Eye camera in userspace, and doing the frame grabbing manually.

Alternatively, Apple could provide advanced APIs in a future OS X update that allows setting the exposure of the iSight camera (as the hardware seems to be capable of setting the exposure, because the auto-exposure is something that is very likely done in software).

[2] http://lists.ims.tuwien.ac.at/pipermail/psmove/2012-June/000029.html, retrieved 2012-12-08

6.3 Future Work

This section presents some ideas which have not been fully implemented as part of this thesis, and which could be implemented in a future follow-up project.

Tracking with Low-FPS Cameras

The library has been designed to work together with the PS Eye camera, as it provides a known-good hardware platform for which the parameters of the lens can be known and the exposure settings are adjustable from Linux and Windows.

In general, however, the library does not limit input to the PS Eye. All cameras supported by OpenCV can be used for vision tracking, and the library has also been used in situations where frame grabbing was done in user code, so any camera that provides an RGB framebuffer (possibly converted from the camera's native colorspace) to the library could be used by the PS Move API.

Some consumer cameras do have a very low FPS (frames per second) count, so tracking fast movements is harder and becomes more error-prone. Another issue with normal consumer cameras is that they usually exhibit large amounts of motion blur, even with relatively slow movements, so tracking quality is not very good, and tracking losses can occur.

To fix these issues, the region of interest size (see section 3.5) can be increased, so that the frame-after-frame tracking works for greater movement distances. Also, the intertial sensors of the controller could be used to predict the new position of the controller, so that for fast movements, only the inertial sensor data is used instead of relying on the (blurry) camera image data.

Methods similar to the ones proposed in [18] could be used to mitigate the effects of motion blur for these low-FPS consumer cameras, too.

Color Selection

Right now, the color selection for each controller in the tracker module is static. The color selection could be dynamic by analyzing the camera picture (at the beginning only or even during tracking) for existing colors, and picking colors that are not seen in the camera frame. As the environment might change (e.g. a person with colored clothes walks into the camera frame), this might need automatic switching of the sphere color during tracking.

Fixing these issues should make the blinking calibration more robust and should allow users to use the PS Move API with more cameras and in more difficult environments without having to do manual configuration steps.

Multi-Threaded Controller Tracking

As described in section 5.2, tracking performance could be improved by moving per-controller tracking into separate threads. This makes sense on multi-core CPUs. In addition to making better use of the available CPU resources, this would reduce the end-to-end system latency when tracking multiple controllers, as tracking results will be available sooner.

Improved Sensor Fusion

Right now, sensor fusion is done by combining the orientation of the controller (3D rotation, from inertial sensors) and the position of the controller (3D position, from computer vision). One idea how this could be improved is to use the inertial sensors to refine the position information of the controller. For example, when the inertial sensors register no movement, and the vision tracking "sees" the controller moving, this can be an indication that the vision part is probably not tracking the right object.

Dead Reckoning using Inertial Sensors

When visual tracking is lost, the inertial sensor data (especially the accelerometer data) can be used for short amounts of time for dead reckoning of the controller's position.

Movement Prediction using Inertial Sensors

In cases where the camera frame comes in with a certain delay (introduced by grabber cards and external cameras) or for fast motion movement, the inertial sensor data from the accelerometer can be used to predict future movement of the controller in the camera image, and partially mitigate the delay introduced by the capture pipeline.

Custom Hardware

Right now, the PS Move API only works with off-the-shelf hardware from Sony. Building custom hardware could give us more control over the communication protocol, and would avoid the pairing issues on Windows. A disadvantage would be that the hardware must be designed and produced, and that large parts of the software might need to be rewritten.

Performance Improvements

Performance is already good on most devices, but especially on mobile devices like phones, the tracking performance could be improved. A different CV library like FastCV could bring some performance improvements, but might require rewriting of parts of the tracker codebase.

For the Processing bindings, the way we convert the captured camera image into a PImage for display in processing is not very efficient. While it works fine for 640x480 images, higher resolutions will become a bottleneck, especially on older hardware. A better solution using shared memory or overlays could improve the situation here.

Connection Monitoring for the Move Daemon on Mac OS X

The Move Daemon "moved" currently only monitors connect/disconnect events on Linux. On Mac OS X, newly-connected controllers are not automatically detected. In situations where long-running processes run using the PS Move API, it might be useful to allow disconnects and reconnects during application runtime. The OS X implementation could be improved in a way

that disconnected devices and newly-connected devices are connected and picked up by moved automatically.

Camera Calibration and Improved OpenGL Projection

For augmented reality using the sensor fusion parts of the library, the OpenGL projection matrix is just an approximation of the real projection of the camera. This could be improved by using a more realistic model of projecting the 3D scene on top of the camera image. This would probably involve re-using data from the camera calibration step and modifying the sensor fusion algorithm accordingly. Having a better projection matrix would also result in better results for the 3D rendering of objects on top of the camera image.

Unity3D Integration using ARTiFICe

ARTiFICe [24] is a framework for quickly creating virtual and augmented reality applications using Unity3D and various types of input devices. ARTiFICe itself supports OpenTracker as input framework, so the OpenTracker integration of the PS Move API can be used to integrate PS Move Motion Controller input with Unity3D. Using the C# bindings of the PS Move API in addition to (or instead of) the OpenTracker framework could make it easier for developers to integrate PS Move input into Unity3D applications.

6.4 Resources on the Internet

This section lists important web pages as a starting point for further reading.

PS Move API

The PS Move API is the open source implementation of the C library, language bindings and example applications described in this thesis.

- `http://thp.io/2010/psmove/`
- `http://github.com/thp/psmoveapi/`

MoveOnPC

The MoveOnPC project is the umbrella project for PS Move on personal and mobile computers, including the Git mirror for the library, file downloads, Wiki and the mailing list.

- `http://code.google.com/p/moveonpc/`
- `https://www.ims.tuwien.ac.at/projects/moveonpc`

PS Move Mailing List

The PS Move Mailing List should be used for discussing the PS Move API and MoveOnPC projects (thesis feedback should be sent to the author, see below).

- `https://lists.ims.tuwien.ac.at/mailman/listinfo/psmove`
- `http://lists.ims.tuwien.ac.at/pipermail/psmove/`

Thesis Errata and Feedback

The webpage contains up to date information and errata. Feedback and questions can be sent via e-mail to the address listed on the about page. For generic (not thesis-related) questions about the PS Move or the PS Move API, please use the PS Move Mailing List.

- `http://thp.io/2012/thesis/`
- `http://thp.io/about`

APPENDIX A

Library API Documentation

This section gives an overview of the API functions, declared types and enumerations in the PS Move API library. This section is intended to be a quick reference for looking up a certain function, and not a detailed explanation of the API functions. Details such as parameter specifications and meanings of return values should be obtained from the header files or the Doxygen documentation (see below).

Generating the API Documentation from Source

The API documentation can be generated using the Doxygen[1] utility. Generating the documentation from source has the advantage of always having the latest up-to-date documentation corresponding to the source release - the documentation is this chapter was valid at the time it has been generated from the source code in the PS Move API Git repository (2012-12-06).

On Debian-based systems such as Ubuntu, Doxygen can be installed with:

```
1 sudo apt-get install doxygen
```

After installing Doxygen and checking out the PS Move API source code, the documentation can be built by issuing the following command in the top-level source directory:

```
1 doxygen
```

The resulting documentation will be built in the "html/" subdirectory, the starting page of the documentation can be found in "html/index.html".

Structure of the Library

The public API of the library is split into several header files, which are found in "include/" in the PS Move API source distribution. Depending on which features of the PS Move API you

[1] http://www.doxygen.org/, retrieved 2012-12-06

use, you might not need the Tracker or Fusion modules. The “psmove.h” file is always needed. Some functions of the API are exposed both as API functions as well as command-line utilities, such as the “psmovepair” utility (which wraps the psmove_pair() API function).

To build your application, you can link against the shared or static library versions of the PS Move API. For the Core module, the shared library is called “libpsmoveapi”, and the static library is called “libpsmoveapi_static”. The Tracker and Fusion modules are both included in the tracker library. The shared tracker library is called “libpsmoveapi_tracker”, and the static tracker library is called “libpsmoveapi_tracker_static”.

A.1 Core Module (psmove.h)

The Core Module handles connections to the PS Move Motion Controller, as well as pairing and reading of button and sensor values.

Types

- **PSMove:** Handle to a PS Move Controller.

Enumerations

PSMove_Connection_Type

Connection type for controllers.

- **Conn_Bluetooth:** The controller is connected via Bluetooth.
- **Conn_USB:** The controller is connected via USB.
- **Conn_Unknown:** Unknown connection type / other error.

PSMove_Button

Button flags.

- **Btn_TRIANGLE:** Green triangle.
- **Btn_CIRCLE:** Red circle.
- **Btn_CROSS:** Blue cross.
- **Btn_SQUARE:** Pink square.
- **Btn_SELECT:** Select button, left side.
- **Btn_START:** Start button, right side.
- **Btn_MOVE:** Move button, big front button.

- **Btn_T:** Trigger, on the back.
- **Btn_PS:** PS button, front center.

PSMove_Frame

Frame of an input report.

- **Frame_FirstHalf:** The older frame.
- **Frame_SecondHalf:** The most recent frame.

PSMove_Battery_Level

Battery charge level.

- **Batt_MIN:** Battery is almost empty ($< 20\%$)
- **Batt_20Percent:** Battery has at least 20% remaining.
- **Batt_40Percent:** Battery has at least 40% remaining.
- **Batt_60Percent:** Battery has at least 60% remaining.
- **Batt_80Percent:** Battery has at least 80% remaining.
- **Batt_MAX:** Battery is fully charged (not on charger)
- **Batt_CHARGING:** Battery is currently being charged.
- **Batt_CHARGING_DONE:** Battery is fully charged (on charger)

PSMove_Update_Result

LED update result, returned by psmove_update_leds()

- **Update_Failed:** Could not update LEDs.
- **Update_Success:** LEDs successfully updated.
- **Update_Ignored:** LEDs don't need updating, see psmove_set_rate_limiting()

PSMove_Bool

Boolean type.

- **PSMove_False:** False, Failure, Disabled (depending on context)
- **PSMove_True:** True, Success, Enabled (depending on context)

PSMove_RemoteConfig

Remote configuration options, for psmove_set_remote_config()

- **PSMove_LocalAndRemote:** Use both local (hidapi) and remote (moved) devices.
- **PSMove_OnlyLocal:** Use only local (hidapi) devices, ignore remote devices.
- **PSMove_OnlyRemote:** Use only remote (moved) devices, ignore local devices.

Functions

psmove_set_remote_config

Enable or disable the usage of local or remote devices.

```
void
psmove_set_remote_config(enum PSMove_RemoteConfig config);
```

psmove_count_connected

Get the number of available controllers.

```
int
psmove_count_connected();
```

psmove_connect

Connect to the default PS Move controller.

```
PSMove *
psmove_connect();
```

psmove_connect_by_id

Connect to a specific PS Move controller.

```
PSMove *
psmove_connect_by_id(int id);
```

psmove_connection_type

Get the connection type of a PS Move controller.

```
enum PSMove_Connection_Type
psmove_connection_type(PSMove *move);
```

psmove_is_remote

Check if the controller is remote (moved) or local.

```
enum PSMove_Bool
psmove_is_remote(PSMove *move);
```

psmove_get_serial

Get the serial number (Bluetooth MAC address) of a controller.

```
char *
psmove_get_serial(PSMove *move);
```

psmove_pair

Pair a controller connected via USB with the computer.

```
enum PSMove_Bool
psmove_pair(PSMove *move);
```

psmove_pair_custom

Pair a controller connected via USB to a specific address.

```
enum PSMove_Bool
psmove_pair_custom(PSMove *move, const char *btaddr_string);
```

psmove_set_rate_limiting

Enable or disable LED update rate limiting.

```
void
psmove_set_rate_limiting(PSMove *move, enum PSMove_Bool enabled);
```

psmove_set_leds

Set the RGB LEDs on the PS Move controller.

```
void
psmove_set_leds(PSMove *move, unsigned char r, unsigned char g, unsigned char
    b);
```

psmove_set_rumble

Set the rumble intensity of the PS Move controller.

```
void
psmove_set_rumble(PSMove *move, unsigned char rumble);
```

psmove_update_leds

Send LED and rumble values to the controller.

```
enum PSMove_Update_Result
psmove_update_leds(PSMove *move);
```

psmove_poll

Read new sensor/button data from the controller.

```
int
psmove_poll(PSMove *move);
```

psmove_get_buttons

Get the current button states from the controller.

```
unsigned int
psmove_get_buttons(PSMove *move);
```

psmove_get_button_events

Get new button events since the last call to this fuction.

```
void
psmove_get_button_events(PSMove *move, unsigned int *pressed, unsigned int *
    released);
```

psmove_get_battery

Get the battery charge level of the controller.

```
enum PSMove_Battery_Level
psmove_get_battery(PSMove *move);
```

psmove_get_temperature

Get the current raw temperature reading of the controller.

```
int
psmove_get_temperature(PSMove *move);
```

psmove_get_trigger

Get the value of the PS Move analog trigger.

```
unsigned char
psmove_get_trigger(PSMove *move);
```

psmove_get_accelerometer

Get the raw accelerometer reading from the PS Move.

```
void
psmove_get_accelerometer(PSMove *move, int *ax, int *ay, int *az);
```

psmove_get_gyroscope

Get the raw gyroscope reading from the PS Move.

```
void
psmove_get_gyroscope(PSMove *move, int *gx, int *gy, int *gz);
```

psmove_get_magnetometer

Get the raw magnetometer reading from the PS Move.

```
void
psmove_get_magnetometer(PSMove *move, int *mx, int *my, int *mz);
```

psmove_get_accelerometer_frame

Get the calibrated accelerometer values (in g) from the controller.

```
void
psmove_get_accelerometer_frame(PSMove *move, enum PSMove_Frame frame, float *
    ax, float *ay, float *az);
```

psmove_get_gyroscope_frame

Get the calibrated gyroscope values (in rad/s) from the controller.

```
void
psmove_get_gyroscope_frame(PSMove *move, enum PSMove_Frame frame, float *gx,
    float *gy, float *gz);
```

psmove_get_magnetometer_vector

Get the normalized magnetometer vector from the controller.

```
void
psmove_get_magnetometer_vector(PSMove *move, float *mx, float *my, float *mz)
    ;
```

psmove_has_calibration

Check if calibration is available on this controller.

```
enum PSMove_Bool
psmove_has_calibration(PSMove *move);
```

psmove_dump_calibration

Dump the calibration information to stdout.

```
void
psmove_dump_calibration(PSMove *move);
```

psmove_enable_orientation

Enable or disable orientation tracking.

```
void
psmove_enable_orientation(PSMove *move, enum PSMove_Bool enabled);
```

psmove_has_orientation

Check if orientation tracking is available for this controller.

```
enum PSMove_Bool
psmove_has_orientation(PSMove *move);
```

psmove_get_orientation

Get the current orientation as quaternion.

```
void
psmove_get_orientation(PSMove *move, float *w, float *x, float *y, float *z);
```

psmove_reset_orientation

Reset the current orientation quaternion.

```
void
psmove_reset_orientation(PSMove *move);
```

psmove_reset_magnetometer_calibration

Reset the magnetometer calibration state.

```
void
psmove_reset_magnetometer_calibration(PSMove *move);
```

psmove_save_magnetometer_calibration

Save the magnetometer calibration values.

```
void
psmove_save_magnetometer_calibration(PSMove *move);
```

psmove_get_magnetometer_calibration_range

Return the raw magnetometer calibration range.

```
int
psmove_get_magnetometer_calibration_range(PSMove *move);
```

psmove_disconnect

Disconnect from the PS Move and release resources.

```
void
psmove_disconnect(PSMove *move);
```

psmove_reinit

Reinitialize the library.

```
void
psmove_reinit();
```

psmove_util_get_ticks

Get milliseconds since first library use.

```
long
psmove_util_get_ticks();
```

psmove_util_get_data_dir

Get local save directory for settings.

```
const char *
psmove_util_get_data_dir();
```

psmove_util_get_file_path

Get a filename path in the local save directory.

```
char *
psmove_util_get_file_path(const char *filename);
```

psmove_util_get_env_int

Get an integer from an environment variable.

```
int
psmove_util_get_env_int(const char *name);
```

psmove_util_get_env_string

Get a string from an environment variable.

```
char *
psmove_util_get_env_string(const char *name);
```

A.2 Tracker Module (psmove_tracker.h)

The Tracker Module interfaces with the OpenCV library and the Core Module to provide vision-based tracking of the controller in the camera image.

Types

- **PSMoveTracker:** Handle to a Tracker object.

Enumerations

PSMoveTracker_Status

Status of the tracker.

- **Tracker_NOT_CALIBRATED:** Controller not registered with tracker.
- **Tracker_CALIBRATION_ERROR:** Calibration failed (check lighting, visibility)
- **Tracker_CALIBRATED:** Color calibration successful, not currently tracking.
- **Tracker_TRACKING:** Calibrated and successfully tracked in the camera.

PSMoveTracker_Exposure

Exposure modes.

- **Exposure_LOW:** Very low exposure: Good tracking, no environment visible.
- **Exposure_MEDIUM:** Middle ground: Good tracking, environment visibile.
- **Exposure_HIGH:** High exposure: Fair tracking, but good environment.
- **Exposure_INVALID:** Invalid exposure value (for returning failures)

Functions

psmove_tracker_new

Create a new PS Move Tracker instance and open the camera.

```
1 PSMoveTracker *
2 psmove_tracker_new();
```

psmove_tracker_new_with_camera

Create a new PS Move Tracker instance with a specific camera.

```
1 PSMoveTracker *
2 psmove_tracker_new_with_camera(int camera);
```

psmove_tracker_set_auto_update_leds

Configure if the LEDs of a controller should be auto-updated.

```
void
psmove_tracker_set_auto_update_leds(PSMoveTracker *tracker, PSMove *move,
    enum PSMove_Bool auto_update_leds);
```

psmove_tracker_get_auto_update_leds

Check if the LEDs of a controller are updated automatically.

```
enum PSMove_Bool
psmove_tracker_get_auto_update_leds(PSMoveTracker *tracker, PSMove *move);
```

psmove_tracker_set_dimming

Set the LED dimming value for all controller.

```
void
psmove_tracker_set_dimming(PSMoveTracker *tracker, float dimming);
```

psmove_tracker_get_dimming

Get the LED dimming value for all controllers.

```
float
psmove_tracker_get_dimming(PSMoveTracker *tracker);
```

psmove_tracker_set_exposure

Set the desired camera exposure mode.

```
void
psmove_tracker_set_exposure(PSMoveTracker *tracker, enum
    PSMoveTracker_Exposure exposure);
```

psmove_tracker_get_exposure

Get the desired camera exposure mode.

```
enum PSMoveTracker_Exposure
psmove_tracker_get_exposure(PSMoveTracker *tracker);
```

psmove_tracker_enable_deinterlace

Enable or disable camera image deinterlacing (line doubling)

```
void
psmove_tracker_enable_deinterlace(PSMoveTracker *tracker, enum PSMove_Bool
    enabled);
```

psmove_tracker_set_mirror

Enable or disable horizontal camera image mirroring.

```
void
psmove_tracker_set_mirror(PSMoveTracker *tracker, enum PSMove_Bool enabled);
```

psmove_tracker_get_mirror

Query the current camera image mirroring state.

```
enum PSMove_Bool
psmove_tracker_get_mirror(PSMoveTracker *tracker);
```

psmove_tracker_enable

Enable tracking of a motion controller.

```
enum PSMoveTracker_Status
psmove_tracker_enable(PSMoveTracker *tracker, PSMove *move);
```

psmove_tracker_enable_with_color

Enable tracking with a custom sphere color.

```
enum PSMoveTracker_Status
psmove_tracker_enable_with_color(PSMoveTracker *tracker, PSMove *move,
    unsigned char r, unsigned char g, unsigned char b);
```

psmove_tracker_disable

Disable tracking of a motion controller.

```
void
psmove_tracker_disable(PSMoveTracker *tracker, PSMove *move);
```

psmove_tracker_get_color

Get the desired sphere color of a motion controller.

```
int
psmove_tracker_get_color(PSMoveTracker *tracker, PSMove *move, unsigned char
    *r, unsigned char *g, unsigned char *b);
```

psmove_tracker_get_camera_color

Get the sphere color of a controller in the camera image.

```
int
psmove_tracker_get_camera_color(PSMoveTracker *tracker, PSMove *move,
    unsigned char *r, unsigned char *g, unsigned char *b);
```

psmove_tracker_set_camera_color

Set the sphere color of a controller in the camera image.

```
int
psmove_tracker_set_camera_color(PSMoveTracker *tracker, PSMove *move,
    unsigned char r, unsigned char g, unsigned char b);
```

psmove_tracker_get_status

Query the tracking status of a motion controller.

```
enum PSMoveTracker_Status
psmove_tracker_get_status(PSMoveTracker *tracker, PSMove *move);
```

psmove_tracker_update_image

Retrieve the next image from the camera.

```
void
psmove_tracker_update_image(PSMoveTracker *tracker);
```

psmove_tracker_update

Process incoming data and update tracking information.

```
int
psmove_tracker_update(PSMoveTracker *tracker, PSMove *move);
```

psmove_tracker_annotate

Draw debugging information onto the current camera image.

```
void
psmove_tracker_annotate(PSMoveTracker *tracker);
```

psmove_tracker_get_frame

Get the current camera image as backend-specific pointer.

```
void *
psmove_tracker_get_frame(PSMoveTracker *tracker);
```

psmove_tracker_get_image

Get the current camera image as 24-bit RGB data blob.

```
PSMoveTrackerRGBImage
psmove_tracker_get_image(PSMoveTracker *tracker);
```

psmove_tracker_get_position

Get the current position and radius of a tracked controller.

```
int
psmove_tracker_get_position(PSMoveTracker *tracker, PSMove *move, float *x,
    float *y, float *radius);
```

psmove_tracker_get_size

Get the camera image size for the tracker.

```
void
psmove_tracker_get_size(PSMoveTracker *tracker, int *width, int *height);
```

psmove_tracker_distance_from_radius

Calculate the physical distance (in cm) of the controller.

```
float
psmove_tracker_distance_from_radius(PSMoveTracker *tracker, float radius);
```

psmove_tracker_set_distance_parameters

Set the parameters for the distance mapping function.

```
void
psmove_tracker_set_distance_parameters(PSMoveTracker *tracker, float height,
    float center, float hwhm, float shape);
```

psmove_tracker_free

Destroy an existing tracker instance and free allocated resources.

```
void
psmove_tracker_free(PSMoveTracker *tracker);
```

A.3 Sensor Fusion Module (psmove_fusion.h)

The Sensor Fusion Module uses both the Core Module and the Tracker Module to provide position and orientation information for use in OpenGL-based augmented reality applications.

Types

- **PSMoveFusion:** Handle to a PS Move Fusion object.

Functions

psmove_fusion_new

Create a new PS Move Fusion object.

```
PSMoveFusion *
psmove_fusion_new(PSMoveTracker *tracker, float z_near, float z_far);
```

psmove_fusion_get_projection_matrix

Get a pointer to the 4x4 projection matrix.

```
float *
psmove_fusion_get_projection_matrix(PSMoveFusion *fusion);
```

psmove_fusion_get_modelview_matrix

Get a pointer to the 4x4 model-view matrix for a controller.

```
float *
psmove_fusion_get_modelview_matrix(PSMoveFusion *fusion, PSMove *move);
```

psmove_fusion_get_position

Get the 3D position of a controller.

```
void
psmove_fusion_get_position(PSMoveFusion *fusion, PSMove *move, float *x,
    float *y, float *z);
```

psmove_fusion_free

Destroy an existing fusion instance and free allocated resources.

```
void
psmove_fusion_free(PSMoveFusion *fusion);
```

APPENDIX B

Low-Level HID Protocol

This section describes the byte-level layout of the HID messages sent to and from the controller. The first two messages are sent by reading or writing directly from/to the controller. All other messages are read/sent using feature reports.

- 0x01 - Get Input (Move $\rightarrow$ Host)
- 0x02 - Set LEDs (Host $\rightarrow$ Move)
- 0x04 - Get Bluetooth Address (Get Feature Report)
- 0x05 - Set Bluetooth Address (Send Feature Report)
- 0x10 - Get Calibration Data (Get Feature Report)

0x01 - Get Input (Move → Host)

Offset	Bytes	Description	Example
0x00	1	Message ID	always 0x01
0x01	4	Buttons bit field	
0x05	1	Trigger (first frame)	0x00 (not pressed)
0x06	1	Trigger (second frame)	0xff (fully presssed)
0x07	4	Unknown	
0x0b	1	Timestamp (high byte)	
0x0c	1	Battery level	0x05 (fully charged)
0x0d	2	Accelerometer X (first frame)	
0x0f	2	Accelerometer Y (first frame)	
0x11	2	Accelerometer Z (first frame)	
0x13	2	Accelerometer X (second frame)	
0x15	2	Accelerometer Y (second frame)	
0x17	2	Accelerometer Z (second frame)	
0x19	2	Gyroscope X (first frame)	
0x1b	2	Gyroscope Y (first frame)	
0x1d	2	Gyroscope Z (first frame)	
0x1f	2	Gyroscope X (second frame)	
0x21	2	Gyroscope Y (second frame)	
0x23	2	Gyroscope Z (second frame)	
0x25	1	Temperature (bits 12-5)	
0x26	1	Temperature (bits 4-1), Magnetometer X (bits 12-9)	
0x27	1	Magnetometer X (bits 8-1)	
0x28	1	Magnetometer Y (bits 12-5)	
0x29	1	Magnetometer Y (bits 4-1), Magnetometer Z (bits 12-9)	
0x2a	1	Magnetometer Z (bits 8-1)	
0x2b	1	Timestamp (low byte)	
	44	Total size (padded to 49 bytes)	

When reading from the HID device, it returns a 49-byte input report containing the information about the buttons, trigger and inertial sensors. Right now, we can only get this input report via Bluetooth, although it should be technically possible to get the input report via USB as well.

Some values such as the temperature and magnetometer are saved as signed 12-bit values. Accelerometer and gyroscope values are saved as signed 16-bit values. The trigger value is an unsigned 8-bit value.

The buttons bitfield contains a bit for each pressed button. It also contains a sequence number in the lower 4 bits of the fourth byte - this sequence number can be used to detect dropped frames.

Right now, it's not known what the four bytes starting at 0x07 represent.

For the accelerometer and gyroscope values, two frames are contained in each report. The first frame represents the "older" frame, and the second frame represents the most recent frame.

0x02 - Set LEDs (Host → Move)

Offset	Bytes	Description	Example
0x00	1	Message ID	always 0x02
0x01	1	Zero	always 0x00
0x02	1	Red component of LED	0xff (full intensity)
0x03	1	Green component of LED	0x00 (green LED off)
0x04	1	Blue component of LED	0x80 (half intensity)
0x05	1	Unknown	set to 0x00
0x06	1	Rumble	0xff (full intensity), 0x00 (rumble off)
	7	Total size (padded to 49 bytes)	

Setting the LEDs is accomplished by writing a 49-byte message to the HID device. In Linux, it is also possible to skip the padding at the end, which improves the write performance a bit. Both the LED color and the rumble intensity are sent in the same message. The LED color is supplied as three unsigned 8-bit values (red, green and blue component) and the rumble intensity is supplied as single unsigned 8-bit value.

After sending the message, the controller keeps the color and the rumble intensity for about 4-5 seconds. After that, the LEDs and the rumble motor will turn off. For keeping the LEDs lit, it is therefore important to send a message every two seconds to keep the LEDs turned on.

0x04 - Get Bluetooth Address (Get Feature Report)

Offset	Bytes	Description	Example
0x00	1	Message ID	always 0x04
0x01	6	Controller Bluetooth address	
0x0a	6	Current Host Bluetooth address	
	16	Total size	

For the pairing process it is important to get the Bluetooth address of the Motion Controller. This can be accomplished with this message. In addition to getting the Controller address, the currently-set host address is also returned (it could be saved for restoring later).

This message only makes sense over USB during the pairing process. In general, it is useful to use it before the message 0x05 (Set Bluetooth Address) to check if the currently-set host address is different from the desired one. If the host address is already correct, updating it can be avoided.

0x05 - Set Bluetooth Address (Send Feature Report)

Offset	Bytes	Description	Example
0x00	1	Message ID	always 0x05
0x01	6	New Host Bluetooth address	
	23	Total size	

This is the core "pairing" message - it writes the host Bluetooth address into the controller. The controller will connect to this Bluetooth address when switched on.

0x10 - Get Calibration Data (Get Feature Report)

Offset	Bytes	Description	Example
0x00	1	Message ID	always 0x10
0x01	1	Block ID	0x00 (first), 0x01 (second) or 0x82 (third)
0x02	47	Payload data	
	49	Total size	

For reading the factory-set calibration data, this feature report has to be requested three times in a row. Depending on the first byte of the result, the payload block position can be determined. The controller keeps track of which payload block has been sent internally (e.g. if an application requests only one block, the next application will get block two, three and one – in that order – when reading the calibration data). It is therefore important to verify the block ID (at offset 0x01) to determine which block has been read.

The blocks wrap at the third block, so after reading the third block, the next request will read the first block. Independent of the current internal state of the controller, all three blocks can be read by getting this feature report three times in a row, and making sure to place the payload at the correct offset.

The assembled calibration blob consists of the Message ID, the first Block ID (0x00) followed by three payload blocks (each 47 bytes long).

APPENDIX C

Move Daemon UDP Protocol

This chapter describes the UDP-based Move Daemon Protocol used by the Move Daemon and its client implementation in the PS Move API. This is mostly useful for debugging purposes of network-related bugs, and for interoperability: Applications implementing the Move Daemon UDP Protocol can interface with both the PS Move API Library (as server) and the Move Daemon (as client). However, only the bare minimum of functionality is exposed at a very low level (i.e. only raw data reports are sent over the wire protocol) - this is required to simplify the implementation, avoid network traffic overhead and make sure that all calculations (like the orientation algorithm) happen on the client, avoiding unnecessary load on the machine running the Move Daemon.

Generic Information

The following basic information is true for the moved implementation and the current default client implementation in the PS Move API:

- Protocol: UDP
- Well-known port: 17777
- Request size: 9 bytes
- Response size: 50 bytes
- Request timeout: 10 ms
- Request retries: 5

Request Packet Structure

Requests are sent from the client to the server, and have the following structure:

- Packet size: 9 bytes
- Byte 0: Request ID
- Byte 1: Controller ID
- Bytes 2-8: Optional payload

Response Packet Structure

Responses are sent from the server to the client in response to a request, and have the following structure:

- Packet size: 50 bytes
- Bytes 0-49: Response data (message-specific)

Messages

This section lists the types of messages that the client can request. Right now, all messages are client-initiated, which simplifies the protocol and makes sure that the moved server does not have to keep state information around between restarts.

Request 0x01: Count Connected

- Request payload: None
- Response: Byte 0 = Controller count

Get the number of connected controllers from the remote host. If no controllers are connected, the remote host returns 0. The number of connected controllers is returnd as a single unsigned byte value as the first byte of the response.

Request 0x03: Write Data

- Request payload: LED and rumble data
- Response: None

Write data to the controller. The data layout is the same as used in the PS Move HID Protocol for sending LED and rumble updates. No response is sent for this request, as it is expected that the client will send many updates, and does not have time to wait for a response (lost packets will be accounted for by simply sending another update).

Request 0x04: Read Data

- Request payload: None
- Response: Input report data

Read data from the controller. The data layout is the same as used in the PS Move HID Protocol for reading input reports. The client must retry the request if no response is sent after the timeout.

Request 0x05: Get Serial Number

- Request payload: None
- Response: Serial number as ASCII string

Get the serial number as ASCII string from the controller. The response will contain the zero-terminated serial number of the controller as string starting from byte 0.

Bibliography

[1] The BlueZ Authors. Bluez, http://www.bluez.org/, 2010.

[2] Gabriele Bleser and Didier Stricker. Advanced tracking through efficient image processing and visual–inertial sensor fusion. *Computers & Graphics*, 33(1):59 – 72, 2009.

[3] Derek Bradley and Gerhard Roth. Natural interaction with virtual objects using vision-based six dof sphere tracking. In *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, ACE '05, pages 19–26, New York, NY, USA, 2005. ACM.

[4] Géry Casiez, Nicolas Roussel, and Daniel Vogel. 1 € filter: a simple speed-based low-pass filter for noisy input in interactive systems. In *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems*, CHI '12, pages 2527–2530, New York, NY, USA, 2012. ACM.

[5] The MoveOnPC contributors. Moveonpc, http://code.google.com/p/moveonpc/, 2010.

[6] Oracle Corporation. Java native interface 6.0 specification. *Specification, JNI*, 2012.

[7] G-Truc Creation. Opengl mathematics library, http://glm.g-truc.net/, 2005.

[8] Leo Dorst, Daniel Fontijne, and Stephen Mann. *Geometric Algebra for Computer Science: An Object-Oriented Approach to Geometry (The Morgan Kaufmann Series in Computer Graphics)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[9] USB Implementers' Forum. Device class definition for human interface devices (hid), firmware specification. *Specification, HID*, 1:27, 2001.

[10] Ben Fry and Casey Reas. Processing, http://www.processing.org/, 2001.

[11] HID Working Group. Human interface device hid profile 1.0. *Bluetooth SIG*, 1:123, 2003.

[12] Janne Heikkilä. Geometric camera calibration using circular control points. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(10):1066–1077, October 2000.

[13] Kitware Inc. Cmake, http://www.kitware.com/opensource/cmake.html, 1999.

[14] Willow Garage Inc. Opencv, http://opencv.willowgarage.com/, 2009.

[15] D Ioannou. Circle recognition through a 2d hough transform and radius histogramming. *Image and Vision Computing*, 17(1):15–26, 1999.

[16] Michael Isard and John MacCormick. Bramble: A bayesian multiple-blob tracker. In *ICCV*, pages 34–41, 2001.

[17] Martin Kaltenbrunner, Till Bovermann, Ross Bencina, and Enrico Costanza. Tuio - a protocol for table based tangible user interfaces. In *Proceedings of the 6th International Workshop on Gesture in Human-Computer Interaction and Simulation (GW 2005)*, Vannes, France, 2005.

[18] G.S.W. Klein and T.W. Drummond. Tightly integrated sensor fusion for robust visual tracking. *Image and Vision Computing*, 22(10):769 – 776, 2004. <ce:title>British Machine Vision Computing 2002</ce:title>.

[19] Dawei Liang, Qingming Huang, Shuqiang Jiang, Hongxun Yao, and Wen Gao. Mean-shift blob tracking with adaptive feature selection and scale adaptation. In *ICIP (3)*, pages 369–372. IEEE, 2007.

[20] Jorge Lobo and Jorge Dias. Fusing of image and inertial sensing for camera calibration. 2001.

[21] S.O.H. Madgwick, A.J.L. Harrison, and R. Vaidyanathan. Estimation of imu and marg orientation using a gradient descent algorithm. In *IEEE International Conference on Rehabilitation Robotics (ICORR)*, ICORR '11, pages 1–7, 2011.

[22] SWIG maintainers. Swig, http://www.swig.org/, 2010.

[23] R. Miletitch, R. de Courville, M. Rébulard, C. Danet, P. Doan, and D. Boutet. Real-time 3d gesture visualisation for the study of sign language. 2012.

[24] Annette Mossel, Christian Schönauer, Georg Gerstweiler, and Hannes Kaufmann. Artifice - augmented reality framework for distributed collaboration. *The International Journal of Virtual Reality*, 2012.

[25] OmniVision. Ov7725 vga product brief. *Spec Sheet*, 2008.

[26] Alan Ott. hidapi, http://www.signal11.us/oss/hidapi/, 2010.

[27] G. Reitmayr and D. Schmalstieg. Opentracker: A flexible software design for three-dimensional interaction. *Virtual reality*, 9(1):79–92, 2005.

[28] F. Remondino and C. Fraser. Digital camera calibration methods: considerations and comparisons. In Isprs, editor, *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences,*, volume Vol. XXXVI, Dresden, Germany, 2006.

[29] Kenn Sebesta. Repurposing the ps3 move, http://www.eissq.com/ps3_move/, 2011.

[30] Sony. Move.me, http://us.playstation.com/ps3/playstation-move/move-me/, 2011.

[31] Marcin Wojdyr. Fityk: a general-purpose peak fitting program. *Journal of Applied Crystallography*, 43(5 Part 1):1126–1128, Oct 2010.

[32] Suya You, Ulrich Neumann, and Ronald Azuma. Hybrid inertial and vision tracking for augmented reality registration. In *VR*, pages 260–, 1999.

[33] Zhengyou Zhang. A flexible new technique for camera calibration. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22:1330–1334, 1998.