PS Move API: A Cross-Platform 6DoF Tracking Framework

Thomas Perl* Benjamin Venditti† Hannes Kaufmann‡

Interactive Media Systems Group
University of Technology Vienna

Abstract

With the introduction of 6DoF motion controllers for game consoles, low cost hardware for 3D interaction became widely available. However, no fully-featured software solution for 6DoF tracking exists that takes advantage of the PlayStation (PS) Move Motion Controller without additional hardware.

We designed, developed and evaluated a library - the PS Move API - that enables developers to use the PS Move Motion Controller as 6DoF input device in combination with a camera. Initially we solved hardware related problems such as pairing and communication over USB and Bluetooth. In this paper we describe how we perform visual tracking and sensor fusion, combining visual and inertial data. Performance results show that multiple controllers can be tracked simultaneously in real time.

Developers using the library can choose between a low-level C API or higher-level abstractions in Python, Java, C# or the Processing framework. The library is open source, has been developed and tested on Windows, Mac OS X and Linux, and is released under a Simplified BSD License. It also runs on mobile Linux distributions such as MeeGo 1.2 Harmattan and Android. The full source code is available on the PS Move API website at http://thp.io/2010/psmove/.

Index Terms: I.4.8 [Image Processing and Computer Vision]: Scene Analysis—Sensor Fusion; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual Reality; B.4.2 [Hardware]: Input/Output Devices—Channels and controllers

1 Introduction

With the introduction of motion controllers in home computer game consoles in the mid- to late-2000s, motion input devices such as the Wii Remote, Playstation Move or Microsoft Kinect became widely available. Their use of standard interfaces, such as USB and Bluetooth, allowed PC developers to utilize the controllers in their applications.

The PS Move Motion Controller is an affordable USB [5] and Bluetooth [7] input device featuring 8 digital buttons and 1 analog trigger, a 3-axis accelerometer, 3-axis gyroscope and 3-axis magnetometer. A temperature sensor is integrated as well, and for tactile feedback, the controller can be vibrated using a rumble motor. Tactile feedback does not aid in tracking, but can be used in applications to increase immersion. The controller has a white sphere at its top, which contains an RGB LED that can be set from the host computer. The glowing sphere is used to track the controller's 2D position and radius in the camera frame which enables calculation of the 3D position in world coordinates (figure 1). Unfortunately Sony does not provide a free SDK for PC developers to utilize the PS Move controller, and their paid SDK requires the use of a PlayStation 3 system in a network configuration with a PC.

*e-mail: m@thp.io
†e-mail: benjamin.venditti@gmail.com
‡e-mail: kaufmann@ims.tuwien.ac.at

Figure 1: Interacting with existing applications using the PS Move

The PS Move Controller has been designed to work with the PlayStation 3 (PS3) system and does not automatically pair or work with a normal PC. Therefore our library solves the following problems: Bluetooth **pairing** via custom USB HID messages; setting the **RGB LED** and **rumble motor** intensity; reading **inertial sensor** information and **button/trigger** states; calculating the controller's **orientation** from inertial sensors; tracking the **3D position** relative to the camera via OpenCV [8] and real-time **fusion** of position and orientation data.

The developed library supports all camera inputs that are supported by OpenCV. For our experiments, the PS Eye USB 2.0 camera has been used. It has a very low price point, offers full exposure control and provides a frame rate of 60 FPS at a resolution of 640x480 pixels. We also tested the PS Move API with a RED Scarlet camera which delivers full HD (1920x1080) at 60 Hz via an HDMI or HD-SDI framegrabber card. This allows tracking within a bigger tracking volume at higher accuracy.

To give a broad spectrum of developers and researchers access to these features and to allow straight-forward integration of the tracking features into existing applications and frameworks, the core library is written in C, and SWIG [15] bindings are provided for language interoperability. To verify the functionality and demonstrate possible use cases, example applications have been developed in C, Python, Java, Processing [6] and C#. For determining tracking performance, profiling tools have been written that measure Bluetooth I/O throughput (inertial sensor updates), end-to-end system latency - from setting the LED to getting a stable tracking position - and performance when multiple controllers are tracked simultaneously.

Contributions to the field of low-cost tracking include a fast tracking and sphere estimation algorithm that works well in situations of partial occlusion, cross-platform portability of the core framework, as well as cross-language compatibility using a core C library and SWIG-based bindings.

2 Related Work

In this section, we give a brief overview of off-the-shelf motion input devices that act as controllers for current-generation game consoles. We then analyse relevant colored sphere tracking approaches, since fast and accurate sphere tracking is a critical component of the

PS Move visual tracker. Finally, related work in inertial sensors and sensor fusion is analyzed.

2.1 Game Controllers

Motion game controllers introduced in the seventh generation video game consoles are widely available, and usually feature easy connectivity with PCs using USB, Bluetooth or both, eliminating the need for custom connectors and adapter hardware. However, each major game console (Wii, XBox 360 and PS3) features its own motion controller system:

The **Wii Remote** is used by the Wii system as default input device, whereas the PS3 and XBox 360 ship with gamepad controllers by default, and the motion controller is an additional peripheral. For motion input, the Wii Remote has a 3-axis accelerometer and an infrared optical sensor. The accelerometer is used for simple gesture-based input (shaking, steering), while the infrared (IR) optical sensor is used to control an on-screen pointer. Every user needs to have a separate Wii Remote. Projects such as [13] use the Wii Remote for head tracking, interactive whiteboards and other use cases that involve the built-in IR camera and custom-built devices with infrared LEDs. Other Wii Remote-based head tracking solutions include [17], while [24] and [22] provide generic libraries to interface with the Wii Remote in custom applications.

In order to track the Wii Remote it must point towards the "Sensor Bar", which uses infrared LEDs on the bar and an IR camera at the front of the Wii Remote for tracking. Compared to the Wii Remote, the Playstation Move system can track the 3D position and orientation in 3D space in all poses of the controller.

The Xbox 360 system uses the **Kinect** controller for motion input. The Kinect uses structured light, projecting an infrared dot pattern into the scene which is captured by an infrared camera, to generate a depth image. An object recognition approach [21] has been chosen to detect skeletons of players standing in front of the sensor. A single Kinect sensor is capable of tracking two to four users concurrently. Direct input is not possible with the Kinect due to the lack of buttons. Actions are carried out by using gestures and pointing, which introduces some delay. In the open source community, the OpenKinect project has created libfreenect [19], a library for accessing the Kinect camera, upon which projects such as [12] build and provide an easier-to-use API for using the Kinect in a cross-platform way. Microsoft provides a closed-source free Kinect for Windows SDK.

Compared to the Microsoft Kinect, the PS Move controller can be tracked at much higher speed, higher update rate and with high accuracy. Kinect skeletal tracking latency tests showed latency of about 100-150 milliseconds. In addition the PS Move provides digital buttons and an analog trigger which support interaction without having to rely on gestures. The RGB LED and the built-in rumble motor can be used to give feedback to the user.

On the Playstation 3, the **Playstation Move** is used, consisting of a single PS Eye camera and one or more PS Move Motion Controllers. The controller is tracked visually by the PS Eye camera and 3D position data is fused with orientation data gained via inertial sensors. Multiple controllers can be tracked in one camera image. They are distinguished by different sphere colors. A detailed hardware description is given in section 3.

While Sony provides **move.me** [23] as an official solution for using the PS Move with a PC, this solution is expensive: As of March 2013, the cheapest PS3 model available in Austria is the "Super Slim 12GB" for 189 Euro, the Move.me software is only available for purchase in the American Playstation Store for 99 US Dollars. In Europe the Move.me software is only available upon request from an official university e-mail account [1]. The PS3 also needs some kind of display (e.g. TV set or HDMI monitor), which increases the costs and power consumption and reduces the portability. Our system can work on a battery-powered laptop in scenarios where power outlets are not available, and it can also work with mobile and embedded Linux devices where space is limited. For some use cases, the usage of Move.me could still be an option, especially to off-load the tracking workload to the PS3 system. Move.me only works in combination with a PS3 system, and does not provide access to the tracking algorithm or raw data. Other community projects supporting the PS Move on PCs exist, such as the Move Framework [2]. Compared to our solution, the Move Framework is not maintained any longer, only supports C++ and C# development on Windows, while the PS Move API supports all major platforms and several programming languages in addition to C and is actively developed.

[1] http://uk.playstation.com/moveme/, retrieved 2013-03-10

2.2 Colored Sphere Tracking

Circle and sphere tracking has been a topic in computer vision for decades. Some approaches use edge-based sphere tracking [9] which uses edge detection with Hough Circle Transform to find the boundary of a sphere in an image. Edge detection works well, even if parts of the border are occluded, but since color information is ignored by this approach, spheres with different colors cannot be distinguished.

Another approach for tracking spheres in images uses color filters, such as [1]. Instead of relying on edge/circle detection alone, color is taken into account to filter elements of different color in the camera image. From all detected blobs, the most suitably-shaped blob is used as the detected blob. In addition the authors place differently colored dots on the sphere to enable tracking of the orientation of the sphere.

In [16], colored spheres are used for real-time visualization of gestures in 3D. They use similarly colored RGB LEDs to the ones on the PS Move, but do not utilize inertial sensors.

With respect to tracking of multiple blobs in a single image, solutions such as [11] exist. They ignore color information, but rather create a model of the scene based on the visible image. This model is then validated and refined by comparing it with the captured frame. Assuming the model is valid, movements and positions of all tracked objects can be obtained from it. [11] notes that in some cases it is hard to distinguish single blobs, especially if they are close to each other and appear as a single blob in the camera image.

In the PS Move API we must distinguish between multiple controllers. Therefore color filters are used. We choose an approach similar to [1] where a color filter and blob search are combined to find the sphere in the image. Blob detection is validated by estimating the sphere's center and radius (see section 5.2) and comparing how many pixels of the blob lie inside to how many of them lie outside the estimated circle - if too many pixels of the blob lie outside the circle, we assume that the estimated circle size and position are wrong. As the PS Move Motion Controller features inertial sensors that can be used to track the rotation of the controller, it is not necessary to use visual tracking of the orientation as it is done in [1].

2.3 Inertial Sensors and Sensor Fusion

We use sensor fusion in two ways: (1) To combine readings of accelerometers, gyroscopes and magnetometers to compute the controller's orientation, and (2) to combine 3D position and orientation data for 6DoF tracking.

The PS Move Motion Controller has accelerometers, gyroscopes and magnetometers. Accelerometers measure acceleration relative to the earth's center. The gravity vector can be used to determine the controller's orientation. During movement, the accelerometer measures the combination (sum) of the gravity vector and the acceleration direction vector. However, rotation about any axis parallel to the gravity vector cannot be measured. Gyroscopes measure

angular velocity around a specific axis. In idle state an ideal gyroscope outputs zero, but in general, gyroscopes have a measurement bias. This bias introduces errors when integrating readings over time to calculate the angular rotation. Therefore angular rotation readings drift over time. Magnetometers measure the direction of the magnetic field. This allows to calculate the rotation about an axis parallel to the gravity vector. Magnetometers are susceptible to interference from local magnetic fields, i.e. electronic devices, ferromagnetic objects such as metallic furniture, and others.

Since accelerometer and magnetometer provide a stable reference frame, drift in orientation can be reduced by fusing data from all three sensors. Frequently Kalman filtering is used in sensor fusion algorithms with inertial measurement units, e.g. [4], [20] or [25].

In [14] two new orientation filter algorithms are presented, depending on the availability of sensors and the desired output accuracy:

Inertial Measurement Unit (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.

Attitude and Heading Reference System (AHRS) – This algorithm builds on top of the IMU algorithm, but includes 3-axis magnetometer data. The output is a quaternion describing the attitude relative to both gravity and the magnetic north pole. AHRS systems are also known as MARG (Magnetic, Angular Rate and Gravity) systems.

The 3D orientation is represented using quaternions in [14] to avoid problems with gimbal lock that can happen when using Euler angles.

In the AHRS algorithm, magnetometer and accelerometer input is used for gyroscope bias drift compensation. The orientation is derived from the angular rate measured by the gyroscope. The filter fusion algorithm works by obtaining an estimated orientation from orientation calculations based on the gyroscope readings and accelerometer readings. The filter can be tuned via a single parameter β (the filter gain) for responsiveness, by giving more weight to gyroscope readings, or to long-term stability, by giving more weight to accelerometer and magnetometer readings. The algorithm also takes magnetic distortion into account when magnetometer readings are used. If no magnetometer readings are available, the AHRS algorithm falls back to using the IMU algorithm with the accelerometer and gyroscope only.

In our work we use Madgwick's AHRS algorithm for sensor fusion. Compared to Kalman filtering, Madgwick's AHRS algorithm is easy to use, an efficient implementation in C is provided, and evaluations in [14] have shown that the results are similar or even better than comparable Kalman filter-based sensor fusion algorithms.

3 Hardware Description

Figure 2: Playstation Move Motion Controller

The **PS Move Motion Controller** is a cylindric game controller with a soft white sphere on top of it. The sphere is lit from the inside by an RGB LED. The controller features six buttons on the front, two buttons on the sides, one analog trigger at the back, a Mini USB socket used for pairing and charging, dock charging connectors and an expansion port for connecting accessories.

The controller has a 3-axis accelerometer, a 3-axis gyroscope and a 3-axis magnetometer. The PS Move Motion Controller is often bundled with the **PS Eye Camera** - a USB 2.0 camera providing 640x480 pixels at 60 FPS. A high update rate is a prerequisite for good tracking quality and low latency. Image exposure of the PS Eye can be controlled via software, which aids sphere color detection by avoiding over-exposure of a bright LED in the camera image. Low exposure can also help to avoid motion blur during quick movements. The PS Move API is optimized for the PS Eye USB camera, but is compatible and has been tested with cameras ranging from laptop webcams to professional progressive full HD cameras.

4 Tracking Workflow

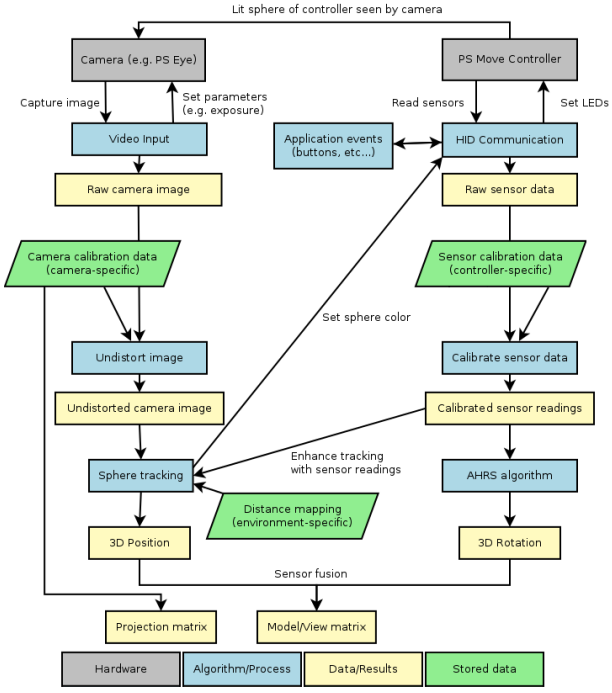

Figure 3: PS Move API Sensor Fusion Workflow.

In figure 3, the sensor fusion workflow is shown, describing the architecture and data flow of a full sensor fusion session using the PS Move API. The left column in figure 3 describes the computer vision component, the right column the inertial sensor fusion component and communication with the PS Move using Bluetooth.

The computer vision component starts by obtaining the image from the camera. Lens undistortion is performed by OpenCV using calibration data, then the sphere is tracked in the undistorted image, the output is the 3D position of the controller.

Inertial sensor fusion obtains the raw inertial sensor data via Bluetooth HID. Raw sensor data is converted into calibrated sensor readings using calibration data from the controller. The calibrated readings are fed into the AHRS algorithm, the output is the 3D rotation of the controller.

In the final sensor fusion step (bottom center of figure 3), the 3D position and 3D rotation are combined into a Model/View matrix for use in 3D rendering of augmented reality content. The Projection matrix is independent of the current position, but is dependent on the intrinsic camera parameters, which are saved in the calibration data.

5 VISUAL TRACKING

Before we can track and estimate position and distance of the controller in space, an initial color calibration step is required. This section describes the challenges we faced during implementation of the visual tracking algorithm, how tracking can be recovered quickly if a controller disappears and reappears in the camera image and how distances can be computed more precisely based on the sphere's radius in the image.

The PS Move API supports image undistortion caused by the camera's lens using the **tracker_camera_calibration** utility for carrying out the calibration. OpenCV is used to estimate focal length, offset and distortion parameters. For details on camera calibration in OpenCV, see *Camera Models and Calibration* in [8].

To get accurate measurements, the calibration of intrinsic camera parameters is required and advised.

5.1 Initial Color Calibration

Having knowledge of and full control over the color of the lit sphere makes tracking much easier. Differently colored parts of the image can be discarded immediately. Unfortunately only the assigned RGB color value of the LED is known and the effective color in the camera image is influenced by intrinsic parameters like color sensitivity, exposure and white balance, and extrinsic parameters like artificial light from flourescent lamps and natural sun light. To overcome this initial lack of information, the PS Move API uses a simple color calibration process whenever a controller is registered for visual tracking. We take two grayscale pictures within a short time, one in which the sphere is switched off and one in which it is lit. From this image pair a difference image is computed in order to find the sphere in the image and extract its effective color by averaging multiple pixels. Unwanted visual changes within two subsequent frames can be diminished by calculating n difference images that are coalesced. Figure 4 illustrates the blinking calibration progress.

For the initial calibration the following steps are executed:

1. Repeat for n times:
   (a) Switch the LED ON and capture a frame (A-Frame)
   (b) Switch the LED OFF and capture a frame (B-Frame)
   (c) Calculate the difference image (C-Frame)
   (d) Remove low energy noise by thresholding
   (e) Remove small structures with eroding and dilating
   (f) Coalesce with previous C-Frame (if existing)
2. Find biggest blob in coalesced C-Frames
3. Estimate average color of the blob in the first A-Frame
4. Use the estimated color to find a blob in each A-Frame so that the blob has a **suitable size** (minimum threshold), and that it has a **similar position and size** compared to the blobs found in the other A-Frames.

Finally for each assigned RGB value, the corresponding color value of the lit sphere in the camera image is known.

5.2 Sphere Size and Center Calculation

To estimate the 2D position of the sphere's midpoint and the sphere's radius in each frame, we tried an approach combining a color filter followed by edge detection and a Hough Circle Transform [10]. However, the Hough Circle Transform turned out to be computationally expensive and quite instable in cases of strong occlusions. It did not provide reliable robust measurements for the sphere's radius. Therefore we estimate the blob's maximum diameter in a robust way by using the maximum distance between points on the contour and applying adaptive smoothing (figure 5):

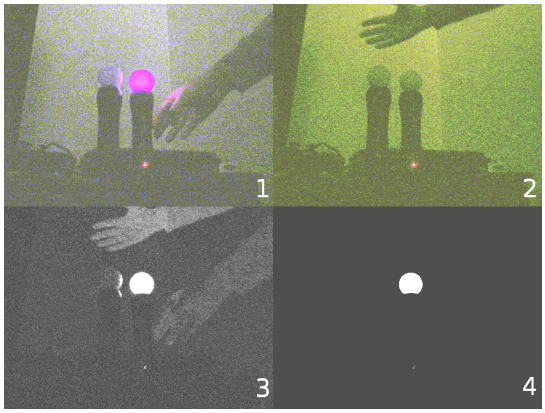

Figure 4: The hand position changes from (1) to (2), resulting in shadows in the difference image (3). Thresholding, eroding and dilating removes these errors (4).

1. Perform a color filter in HSV-space for the estimated color in the current frame t.
2. Find the biggest blob in the resulting binary image.
3. Approximate the spheres center point c_t and radius r_t.
   (a) Take 20 uniformly spaced points of the blob's contour.
   (b) Find the points p_1, p_2 with the biggest distance d_{max}.
   (c) Approximate $c_t = (p_1 + p_2)/2$ and $r_t = d_{max}/2$.
4. Apply adaptive smoothing for the radius with factor f_r.
   (a) Calculate $f_r = MIN\left(\frac{ABS(r_t - r_{t-1})}{4} + 0.15, 1\right)$
   (b) Apply smoothing with $r_t = r_{t-1} * (1 - f_r) + r_t * f_r$.
5. Apply adaptive smoothing for the center with factor f_c.
   (a) Calculate $f_c = MIN\left(\frac{DIST(c_t, c_{t-1})}{7} + 0.15, 1\right)$
   (b) Apply smoothing with $c_t = c_{t-1} * (1 - f_c) + c_t * f_c$.

Diameter estimation can be performed as long as there are at least two opposing pixels from the sphere's contour visible. Adaptive smoothing is only applied for movements smaller than $7px$ in a 640x480 image, which proved to be a good value in our experiments. For smaller movements, stronger smoothing is applied. The same holds for the estimation of the radius but for a lower bound of $4px$. This reduces jitter greatly when the controller is held steadily but remains responsive and introduces no perceived lag for big or spontaneous movements. The parameters f_r and f_c were determined experimentally and are optimized for the 640x480 image of the PS Eye.

5.3 Region of Interest Panning

To speed up the tracking process, we use regions of interest (ROIs) to search for blobs only in relevant subportions of the whole image. The current implementation uses $n = 4$ levels of ROIs decreasing in size by 30% for each level. Whenever a blob has been detected, we adjust the size and position of the ROI used for tracking for the next iteration. In case of fast movements it might happen, that the sphere is already leaving the current ROI, thus only a part of the sphere is detected. Therefore we pan the center of the ROI in each iteration to the center of mass of the detected blob and reevaluate the panned ROI again. The ROI panning enables the algorithm to keep the ROI relatively small even for fast movements and therefore greatly improves the algorithm's performance and stability.

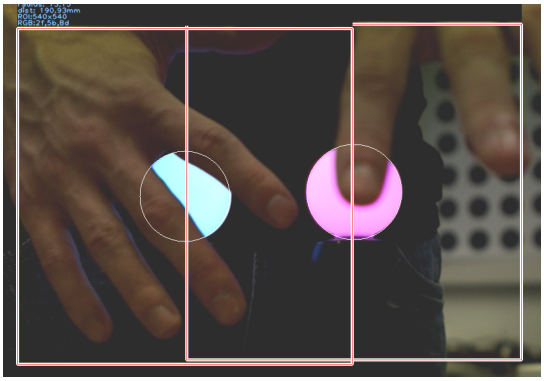

Figure 5: The sphere sizes and centers are properly calculated even though they are only partially visible.

5.4 Tracking Recovery

Whenever the sphere is not found in the current ROI, the algorithm reports a tracking loss and increases the ROI size to the next higher level so that in the next iteration a bigger area is evaluated at its last known position. It may happen that a controller's sphere is not visible in the camera image at all and tracking fails even at the highest ROI level. At that point, the whole image would have to be evaluated to recover tracking of the lost sphere, as it is unknown where it will reenter the image. However, evaluating the whole image introduces a high computational load and would decrease the system's responsiveness severely. Therefore the system doesn't process the camera image in one iteration entirely, but divides it into p tiles and evaluates one tile each iteration. First, all even and then all odd indexed tiles are processed to increase the probability of recovery. Figure 6 ilustrates the processing procedure if tracking is lost:

First, the tile numbered "0" is searched for blobs having the controller's color. Next, the tile numbered "2" is searched (then tiles 4, 6, 1, 3 and 5). When a colored blob is found in the current tile, the tile's position is moved so that the colored blob is at the center of the tile. This moved tile is used as the new ROI for tracking.

Our current implementation uses a simple chess pattern with tiles numbered row-first from top left to bottom right (figure 6).

For further improvement of tracking recovery speed other search patterns could be evaluated in the future e.g. searching a widening region around the last known path of the controller, or searching circularly from image borders to the center of the image.

5.5 Distance Mapping

For true 6DoF tracking, the radius of the sphere in the camera image must be mapped to a real-world distance measurement. The PS Move API returns global 3D coordinates with reference to a coordinate system centered in the optical center of the camera. An empirical mapping is used in the PS Move API for distance mapping. A calculation of the distance using the camera's focal length, its sensor's pixel size and the sphere's size in real world yielded no satisfying results, probably because of irregularities introduced by the tracking algorithm. To create a suitable empirical mapping for desktop sized interaction volumes, we measured the sphere's size in pixels for a range of 2 meters every 20 cm, hence providing a subsampled version of the underlying function. Using the freely available curve-fitting tool *Fityk* [26], the "Pearson7" distribution in equation 1 was found the be the best fit, providing the four configurable parameters *height*, *center*, *hwhm* and *shape*.

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

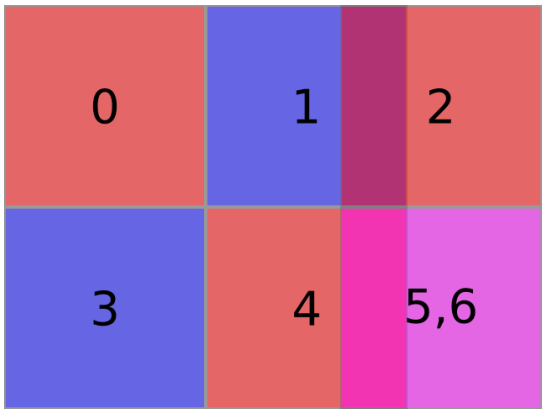

Figure 6: Tiling of a 640x480 camera image $p = 7$. The red (0,2,4,6) and blue(1,3,5) tiles form alternating groups, the last tile is indexed twice to simplify the calculation for the next tile to evaluate with $index_{t+1} = (index_t + 2) mod(p)$.

The PS Move API is already configured to use appropriate values for the PS Eye, so that no additional distance calibration steps are neccessary for users. For other cameras the calibration tool **distance_calibration**, included in the PS Move API, is provided to create a *.csv*-file for successive measurements that can be imported into *Fityk* to obtain custom parameters for the distance mapping.

We chose this empirical way of determining the distance mapping so that custom setups with cameras other than the PS Eye can be easily bootstrapped by simple measurements instead of calculating the distance mapping function from the camera and lens parameters.

5.6 Sensor Fusion

The final sensor fusion step combines the results of computer vision and inertial sensor processing. Our tracking algorithm returns the 3D position of the controller's sphere relative to the camera position. Inertial sensors utilizing the AHRS algorithm provide 3D orientation relative to the earth's frame (center of gravity and magnetic north pole).

The projection matrix is determined by the intrinsic camera parameters, and does not change during runtime, even if multiple controllers are tracked.

The fusion algorithm takes the projection matrix into account, and fits the 3D position of the controller into the scene.

6 Implementation

The core of the PS Move API is implemented as a C library with three different modules (see figure 7): The **Core Module** takes care of HID communication, pairing, processing of inertial sensor data and communication with remote PS Move Controllers via our Move Daemon protocol.

The **Tracker Module** uses OpenCV to obtain images from the camera, processes captured frames and calculates the position and size of controllers as seen from the camera position.

To combine the data of the Core and Tracker modules, the **Fusion Module** is used to provide the 3D position and orientation of each controller relative to the camera position. This module also handles matrix calculations and sets up the projection matrix for a given camera (by default it is configured for the PS Eye camera).

The C library is used by core utilities of PS Move API such as **psmovepair**, a pairing utility that pairs a PS Move Motion Controller via USB to the host computer's Bluetooth address. Example applications are provided in C that demonstrate how to use the

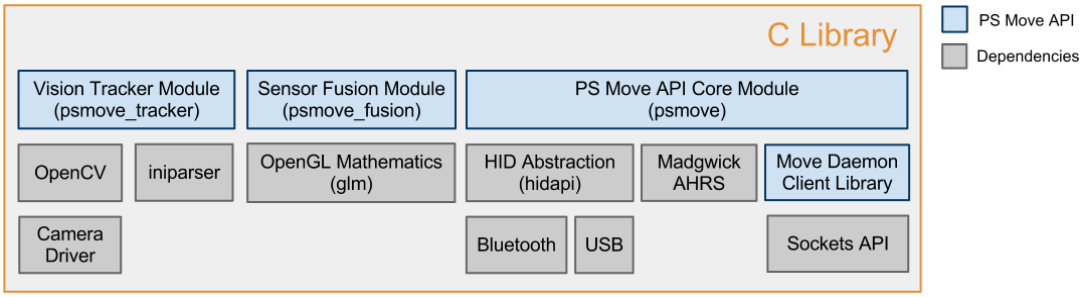

Figure 7: PS Move API C Library Architecture.

Tracker and Fusion modules, as well as very basic examples that show how to use the Core module directly.

6.1 Language Bindings

While C is an efficient and portable language for the core module, in order to make the PS Move API available to a wide range of possible users, we have developed additional language bindings using SWIG [15]. By using the Simplified Wrapper and Interface Generator, we enable developers to use all the features of the C library from different languages. Right now, supported languages are **Java**, **Python** and **C#**. The Java bindings also enable users of the **Processing** framework to use the PS Move API directly.

Because of language differences, the Java, Python and C# bindings do not map all functions in the exact same way. To demonstrate the differences in APIs, the most important tasks such as reading sensor values, performing vision tracking and sensor fusion have all been implemented for all supported languages. This gives potential developers a good starting point for doing custom development in their favorite language.

7 Results

To test our system, and to measure accuracy, latency and performance, several test cases have been designed and implemented in test tools. This ensures repeatability and allows reevaluation whenever the code changes, to easily detect regressions and to measure performance on different computer systems.

Two systems were used for performance tests: A Linux-based notebook with a 1.40 Ghz quad-core CPU and a MacBook Pro running Mac OS X on a 2.50 Ghz dual-core CPU.

We do not intend to compare Linux vs. Mac OS X performance, but rather want to study performance on different types of configurations, especially CPU speed.

Additional test results and a more in-depth analysis of the performance can be found in [18].

7.1 Inertial Sensor Updates

Reading the controller's sensor updates as fast as possible is cricital for calculating orientation and for acquiring button states to react to button presses. In our initial tests, we found that updating the sphere's LED - which involves sending a data packet to the controller - has a negative impact on the effective sensor update rate.

To test the sensor update rate in relation to LED updates, we defined four update methods:

None: No LED updates are sent to the controller - the controller's sphere will be dark. Visual tracking cannot be used in this mode, but we can measure the maximum update rate.

Static: The controller's LED has a static color. The PS Move Motion Controller will automatically keep the LED lit for 4-5 seconds after an update. Therefore the library can be optimized so that it only sends an update every 4000 milliseconds unless the color changes. This is the usual case that we implemented for visual tracking situations.

Smart: The LED color is changed regularly, but the updates are filtered in the library to not exceed a rate of one update every 120 ms. Excessive updates will be ignored.

All: Same as "Smart", but with filtering disabled.

For the tests, we measured the time it takes to read 1000 input reports from the controller, repeated each measurement 5 times and generated the mean over all rounds. The standard deviation was not significant between 5 repeated measurements, so it is not shown in figure 8. The update rate is then calculated as follows:

update_rate = reads / duration

As can be seen in figure 8, the maximum update rate was achieved with the "None" method (no LED updates), and is 83 and 87 updates per second on Mac OS X for the two tested controllers. On Linux, 59 and 60 updates per second can be read from the same two controllers. The differences between Mac OS X and Linux are because of differences in the Bluetooth stack implementation, and the difference between the controllers on the same system is because of different firmware versions in the PS Move controllers.

Updating the LED statically did not result in significant differences in the sensor update rate. This suggests that updating the LED for use in vision tracking does not decrease the quality of inertial sensor calculations.

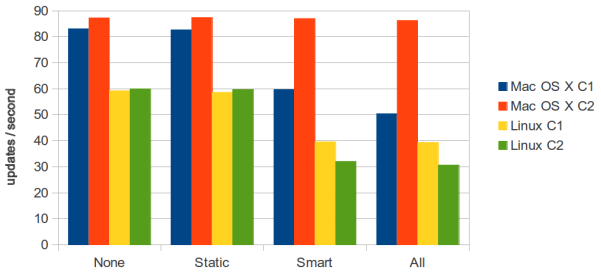

Figure 8: Measured inertial sensor update rate.

Interestingly, controller 2 under Mac OS X is not affected by LED updates, which could suggest that for some controllers on Mac OS X, the read performance is independent of the LED updates, but this depends on the controller and the firmware version in the controller. The reason why controller 2 performs differently under Linux is a bug in the Linux Bluetooth stack (see section 7.5).

7.2 Tracking Performance

For measuring visual tracking performance, 5 PS Move Motion Controllers were tracked simultaneously. As a test application we used **test_capture_performance** from the PS Move API source distribution.

Comparing the slower Linux machine (PS Eye) with the Mac OS X machine (built-in iSight camera) yields data about the relative time used for different processing steps (see figure 9):

Grab describes the time it takes to grab an image from the OS camera driver buffer, **retrieve** the time it takes to convert the image to a colorspace usable by OpenCV. **Convert** is the time it takes to mirror the image horizontally. **Tracking0-4** are the processing steps of tracking each controller in the camera image and obtaining size and position information from it (see section 5.2).

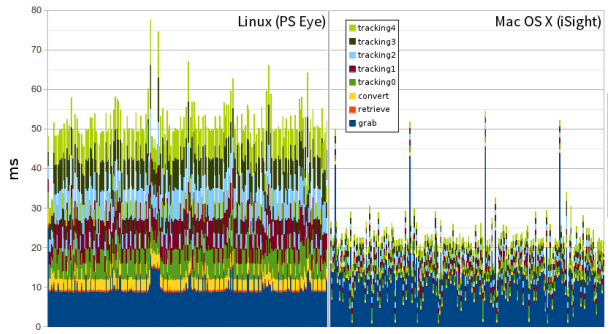

Figure 9: Cumulative per-frame processing duration for 500 frames when tracking 5 PS Move Motion Controllers (horizontal mirror mode).

As can be seen in figure 9, tracking 5 controllers takes about 50 ms on Linux and about 25 ms on Mac OS X, the differences are due to longer per-controller tracking computations.

These results show that with the current single-threaded approach, the number of controllers directly influences the achieveable frame rate of the tracker. Since the clock rate of the Mac OS X-based machine is higher, we can see that tracking is CPU-bound. Using faster processors or multi-threading can lead to higher frame rates without modifications to the tracking algorithm.

Tracking of a single controller takes about 15 ms on the Mac OS X machine (Intel Core 2 Duo, 2.53 GHz). Therefore our tracker implementation can achieve a theoretical frame rate of up to 68 FPS for one controller on that machine.

7.3 System Latency

Without precise measurement hardware, it is difficult to determine the real latency of all processing and tracking steps. Instead of trying to calculate estimations for each step in the pipeline, we chose to implement an experiment that measures the total end-to-end system latency for the PS Move API, which should give a good indication about performance.

Several scenarios were defined to measure the end-to-end system latency, as seen in figure 10:

Off to Grab: The time from switching the LED off to the time the first frame is grabbed where the tracking is lost.

On to Grab: The time from switching the LED on to the time the first frame is grabbed where tracking is recovered.

On to Track: The time from switching the LED on to the time the first frame is processed (tracked) where the tracking is recovered.

On to Stable: The time from switching the LED on to the time the frame is processed (tracked) when 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 LED off.

We can assume that the "Off to Grab" measurement in figure 10 gives a good estimate of the combined LED update and camera latency. Comparing the "On to Grab" results with the "Off to Grab"

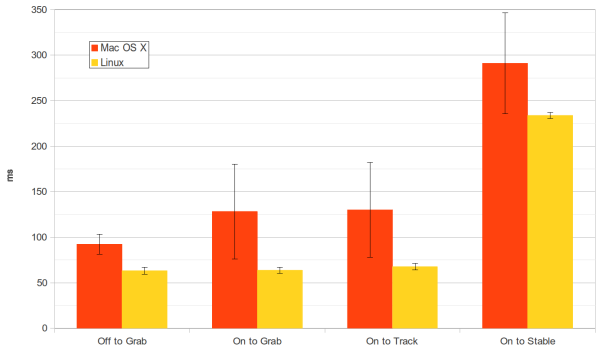

Figure 10: End-to-End System Latency.

results describes the additional latency that we get through the recovery algorithm (ROI panning). "On to Track" is comparable to "On to Grab", so the time it takes to track a single controller in the frame is insignificant compared to the time it takes to switch on the LED and capture frames with the camera. Finally, "On to Stable" describes how long it takes to get a reliable position from the algorithm due to smoothing and dynamic adaption of the ROI position.

The combined LED update and camera latency is about 90 ms for the Mac OS X system (using a built-in iSight camera) and about 60 ms for the Linux system (using the PS Eye camera). Given that the Mac OS X system has a faster CPU, this suggests that the PS Eye camera has less latency compared to normal cameras built into laptops. The total latency from switching the LED on to getting the first position result is about 60 ms on Linux due to a stable frame rate from the PS Eye, and about 160 ms on Mac OS X, where the frame rate is unstable.

The system's latency can be reduced by using faster CPUs, cameras with a higher frame rate or multi-threading. This latency measurement includes the initialization time for the tracking system. In practice, the LEDs are already on, and the tracking latency is less (the combination of the camera image retrieval time and the image processing time). This means that we get a latency of about 15 ms when tracking is already in progress. In AR applications, the camera image will always be in sync with the tracking, so any latency from the camera will not be noticeable in such videos and to users.

7.4 Example Applications

To demonstrate and verify the capabilities of the PS Move API, we have developed several example applications.

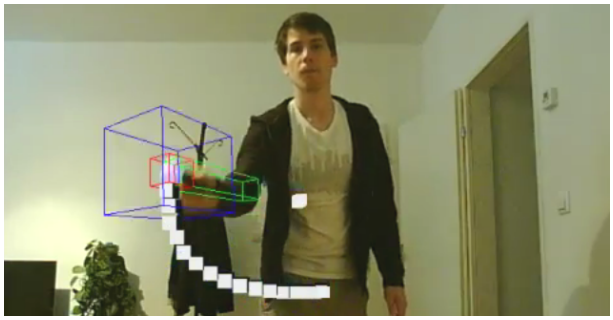

Figure 11: OpenGL-based augmented reality example application

One of these examples is a 3D drawing application where the user can directly interact with the drawn objects (figure 11). In addition to using a single controller, the application can also deal

with multiple controllers, which makes it possible to create multi-user augmented reality applications.

Other examples include a 2D painting application, a two-controller pinch-and-zoom "multi-touch" application, a mouse emulator using the gyroscopes and an interactive whiteboard, where the camera is pointed towards the screen. All applications are available in the PS Move repository.

7.5 Open Issues

While the PS Move API is already used in several academic and non-academic applications, there are still some open issues in the current implementation that have not been resolved yet:

HID write performance on Linux: On Linux, there is some delay when sending LED and rumble information to the controller.

Motion Blur with the PS Eye: In our experiments, the green color channel of the PS Eye camera is very sensitive and causes more motion blur for fast movements than red and blue.

Bluetooth pairing on Windows: Pairing a PS Move Motion Controller on Windows does not work reliably. This is a limitation of the Windows Bluetooth stack.

7.6 Future Work

Color Selection: Right now, controller colors are hard-coded in the library. This could be improved by doing image analysis on the current scene and avoiding colors already present in the scene.

Multi-Threaded Controller Tracking: To improve the frame rate when tracking more than one controller, each controller could be tracked in a separate thread, utilizing each core in today's multi-core CPU setups. This would lower the time it takes from image acquisition to tracking.

Dead Reckoning using Inertial Sensors: When visual tracking is lost, 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: The ROI of the vision tracking algorithm can be moved in the direction of acceleration as reported by the controller to avoid tracking loss in fast movement situations.

Position Filtering using the 1 Euro Filter: We currently use a custom filter for smoothing the computer vision position information. We plan to replace our custom code with the 1 Euro Filter [3] and expect to achieve higher accuracy for position tracking.

8 Summary

We introduce the **PS Move API**, an open source, cross-platform and cross-language framework for doing real time 6DoF tracking with one or more PS Move Motion Controllers and a PS Eye USB 2.0 camera, or any other suitable camera source supported by OpenCV.

The library solves the problem of pairing and communicating with the controller, reading and interpreting the calibration data, and combining computer vision and inertial sensor data to a 6DoF position and rotation estimation for virtual- and augmented reality applications.

Up to five controllers can be tracked simultaneously. In our experiments, we achieved a sensor update frequency of up to 87 updates per second. The end-to-end system latency (from switching the LED on to tracking the controller in the camera) is 68 ms ($\pm$ 3 ms) on Linux using the PS Eye camera, and 130 ms ($\pm$ 52 ms) on Mac OS X with the iSight camera. The tracker implementation can achieve a frame rate of up to 68 FPS for one controller using a dual-core Intel Core 2 Duo at 2.53 GHz.

The PS Move API provides developers with the necessary tools to build useful applications, taking advantage of the accuracy, high update rate and affordability of the PS Move Motion Controller. The full source code can be downloaded from http://thp.io/2010/psmove/.

Acknowledgements

This work was supported by Google Summer of Code 2012.

References

[1] D. Bradley and G. Roth. Natural interaction with virtual objects using vision-based six dof sphere tracking. In *Proceedings of the SIGCHI International Conference on Advances in computer entertainment technology*, ACE '05, pages 19–26, NY, USA, 2005. ACM.

[2] V. Budaházi. Move framework, code.google.com/p/moveframework, 2011.

[3] G. Casiez, N. Roussel, and D. Vogel. 1 euro 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, NY, USA, 2012. ACM.

[4] T. Digaňa. Kalman filtering in multi-sensor fusion. *Control Engineering*, (September), 2004.

[5] U. I. Forum. Device class definition for human interface devices (hid), firmware specification. *Specification, HID*, 1:27, 2001.

[6] B. Fry and C. Reas. Processing, http://www.processing.org/, 2001.

[7] H. W. Group. Human interface device hid profile 1.0. *Bluetooth SIG*, 1:123, 2003.

[8] W. G. Inc. Opencv, http://opencv.willowgarage.com/, 2009.

[9] D. Ioannou. Circle recognition through a 2d hough transform and radius histogramming. *Image and Vision Computing*, 17(1):15–26, 1999.

[10] D. Ioannou, W. Huda, and A. Laine. Circle recognition through a 2d hough transform and radius histogramming. *Image and vision computing*, 17(1):15–26, 1999.

[11] M. Isard and J. MacCormick. Bramble: A bayesian multiple-blob tracker. In *ICCV*, pages 34–41, 2001.

[12] R. Joaquim and L. Eduardo. Gfreenect, https://github.com/elima/gfreenect, 2011.

[13] J. C. Lee. Hacking the nintendo wii remote. *IEEE Pervasive Computing*, 7(3):39–45, 2008.

[14] S. Madgwick, A. 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.

[15] S. maintainers. Swig, http://www.swig.org/, 2010.

[16] 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.

[17] R. Pavlik. Wiimote head-tracking, http://rpavlik.github.com/wiimote-head-tracker-gui/, 2011.

[18] T. Perl. Cross-platform tracking of a 6dof motion controller. Master's thesis, Institut für Softwaretechnik und Interaktive Systeme, 2013.

[19] T. O. Project. libfreenect, https://github.com/openkinect/libfreenect/, 2010.

[20] J. Z. Sasiadek and J. Khe. Sensor fusion based on fuzzy kalman filter. *Proceedings of the Second International Workshop on Robot Motion and Control RoMoCo01 IEEE Cat No01EX535*, (7):275–283, 2001.

[21] J. Shotton, A. Fitzgibbon, M. Cook, T. Sharp, M. Finocchio, R. Moore, A. Kipman, and A. Blake. Real-time human pose recognition in parts from single depth images. In *CVPR*, 2011.

[22] L. D. Smith. cwiid - library to interface with the wiimote, http://abstrakraft.org/cwiid/, 2007.

[23] Sony. Move.me, http://us.playstation.com/ps3/playstation-move/move-me/, 2011.

[24] The WiiYourself author. Wiiyourself! - native c++ wiimote library, http://wiiyourself.gl.tter.org/, 2007.

[25] R. Van Der Merwe, E. A. Wan, and S. Julier. Sigma-point kalman filters for nonlinear estimation and sensor-fusion—applications to integrated navigation. In *Proceedings of the AIAA Guidance, Navigation & Control Conference*, pages 16–19, 2004.

[26] M. Wojdyr. Fityk: a general-purpose peak fitting program. *Journal of Applied Crystallography*, 43(5 Part 1):1126–1128, Oct 2010.