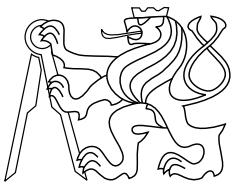
CENTER FOR
MACHINE PERCEPTION



CZECH TECHNICAL
UNIVERSITY IN PRAGUE

MASTER'S THESIS

ISSN 1213-2365

Implementing and Applying Fast Moving Object Detection on Mobile Devices

Aleš Hrabalík

hrabaale@fel.cvut.cz, matas@cmp.felk.cvut.cz

May 26, 2017

Thesis Advisor: prof. Ing. Jiří Matas, Ph.D.

Published by

Center for Machine Perception, Department of Cybernetics
Faculty of Electrical Engineering, Czech Technical University
Technická 2, 166 27 Prague 6, Czech Republic
fax +420 2 2435 7385, phone +420 2 2435 7637, www: http://cmp.felk.cvut.cz

Implementing and Applying Fast Moving Object Detection on Mobile Devices

Aleš Hrabalík

May 26, 2017

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Hrabalík** Jméno: **Aleš** Osobní číslo: **393099**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Studijní obor: **Počítačová grafika a interakce**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Implementing and Applying Fast Moving Object Detection on Mobile Devices

Název diplomové práce anglicky:

Implementing and Applying Fast Moving Object Detection on Mobile Devices

Pokyny pro vypracování:

1. Get familiar with the algorithm for tracking fast moving objects [1].
2. Implement the above-mentioned algorithm in portable C++. Pay attention to efficiency, profile the code and discuss bottlenecks.
3. Design and develop a mobile application that makes use of the algorithm. The application is to estimate the speed of a tennis ball or a similar object. Target the Android operating system.
4. Collect data for evaluation of the application. Assess the performance and usability of the application on the data.
5. Optionally, suggest modification of the algorithm.

Seznam doporučené literatury:

[1] D. Rozumnyi, J. Kotera, F. Sroubek, L. Novotny, J. Matas: The World of Fast Moving Objects. Currently in review. Available online at https://arxiv.org/abs/1611.07889

Jméno a pracoviště vedoucí(ho) diplomové práce:

prof. Ing. Jiří Matas Ph.D., skupina vizuálního rozpoznávání FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **12.01.2017** Termín odevzdání diplomové práce: **26.05.2017**

Platnost zadání diplomové práce: _____

Podpis vedoucí(ho) práce	Podpis vedoucí(ho) ústavu/katedry	Podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. _____ Datum převzetí zadání

_____ Podpis studenta

Prohlášení autora práce

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne ..... Podpis autora práce .....

Abstrakt

Tato práce se zabývá tzv. rychle se pohybujícími objekty (fast-moving objects, FMO), které se jeví rozmazané, protože během expozice urazí značnou vzdálenost. Počítačové vidění se touto problematikou zabývá teprve krátce, ačkoliv takovéto objekty často nalezneme například ve videích se sportovní tématikou. Vyvinuli jsme efektivní algoritmus na detekci FMO ve videích, který se zaměřuje na objekty ve tvaru koule viditelné po tři za sebou jdoucí snímky, a který svou kvalitou detekce překonal nejlepší doposud známé řešení. Algoritmus jsme implementovali přenositelně pomocí jazyka C++ a aplikovali jej na osobních počítačích i na mobilních telefonech. Výkon algoritmu jsme zlepšili na základě podrobné analýzy. Testovali jsme na několika zařízeních s operačním systémem Android a na všech jsme dosáhli detekce FMO v reálném čase.

Dále jsme upravili zmíněný algoritmus tak, abychom mohli odhadnout další vlastnosti detekovaných FMO, jako je poloměr a rychlost. Zjistili jsme, že pokud je video dostatečně kvalitní, odhad poloměru je možný s uspokojivou přesností. Odhad rychlosti jsme uskutečnili v nahrávce tenisového zápasu. Za účelem lepšího porovnání s budoucími algoritmy pro detekci FMO jsme také přidali videa a anotace do datové sady, která je veřejně dostupná.

Abstract

This thesis studies so-called fast-moving objects (FMOs), which appear blurry due to moving a significant distance during camera exposure time. This field of computer vision research is still new and largely unexplored, in spite of FMOs being commonplace in sports footage and elsewhere. We have developed an efficient algorithm for FMO detection in video sequences, which specializes on spherical objects visible for three consecutive frames, and which outperforms the best previously published approach in terms of detection quality. The algorithm has been implemented in portable C++, deployed on desktop and mobile, profiled, and optimized for speed. We demonstrate that real-time FMO detection with a live camera feed as input is feasible on a variety of popular Android devices.

Furthermore, we have modified the algorithm to estimate properties of spherical FMOs, in particular their radius and velocity. We show that radius can be estimated reliably, given source footage of a sufficient quality. We experiment with velocity estimation using a recording of a tennis match. Finally, in order to make future comparisons of FMO detection algorithms more comprehensive, we contribute videos and ground truth into a publicly available data set.

Acknowledgements

I would like to thank prof. Matas for all the patience and guidance regarding the thesis. Further thanks go to my family, who have been immensely supportive, and to my friends. Special thanks go to the members of the now-defunct band Trilobajt, who to this day keep me in good spirits, and I suspect they always will.

Contents

1. Introduction

This thesis deals with high-velocity objects captured on video. Introduced only recently by Rozumnyi et al.[1], this field of computer vision research is still new and unexplored. Intuitively, in a still image, a high-velocity object appears as a long, blurred, semi-transparent streak; the goal is to detect it and track it. To distinguish such an object from regular, low-velocity entities in the image, [1] use the term *fast-moving objects*, and go on to show that traditional object tracking techniques fail when confronted with the new problem. The methods that are known not to work include: ASMS[2], DSST[3], MEEM[4], SRDCF[5], and STRUCK[6]. With the existing techniques inapplicable, there is a need for a completely new approach.

Also pointed out by [1] is the lack of benchmarking data; standard object tracking data sets, such as ALOV[7], VOT[8], and OTB[9] have been found to contain no fast-moving objects whatsoever. This is a curious oversight, as it is quite common to observe such objects when capturing video — see figures 1.1 and 1.2 for examples. In sports videos, a ball or other equipment tends to fly at a high velocity; given enough speed, vehicles such as cars and airplanes appear as a streak in the frame of reference of a close-by still camera; and particles such as sparks, raindrops and hail also tend to appear long and blurred.

We are interested in estimating the velocity of fast-moving objects in video sequences. This is useful in the domain of sports, which our efforts are focused on, but not limited to. A good example of applying such technology would be the game of tennis, where the audience is particularly interested in the speed of the tennis ball during a service. In high-profile matches, a special device usually measures ball speed and reports it to the viewers. Our intention is to enable the viewers to measure the speed themselves, using only their smartphone devices. Another example would be a ball throwing competition, in which players compete to throw the ball the fastest; here we envision a single smartphone functioning as a speed camera.

The main contribution of this thesis is a highly efficient FMO detector written in portable C++, which we use to demonstrate the viability of real-time FMO detection on desktop and mobile. We go on to show that the detector performs better that the method published in [1], and can be used for estimation of object radius and velocity. The structure of the thesis is as follows: in chapter 2, the problem and the goals are described formally; in chapter 3, an efficient algorithm for fast-moving object detection is described in detail; chapter 4 discusses the implementation details of said algorithm; in chapter 5, performance of the algorithm is analyzed; detection quality is evaluated in chapter 6; and chapters 7 and 8 discuss radius and velocity estimation, respectively, followed by conclusion in chapter 9.

Figure 1.1. Real-world examples of fast-moving objects. Top left: a Formula 1 car. Top right: a luger. Bottom left: hail. Bottom right: sparks.

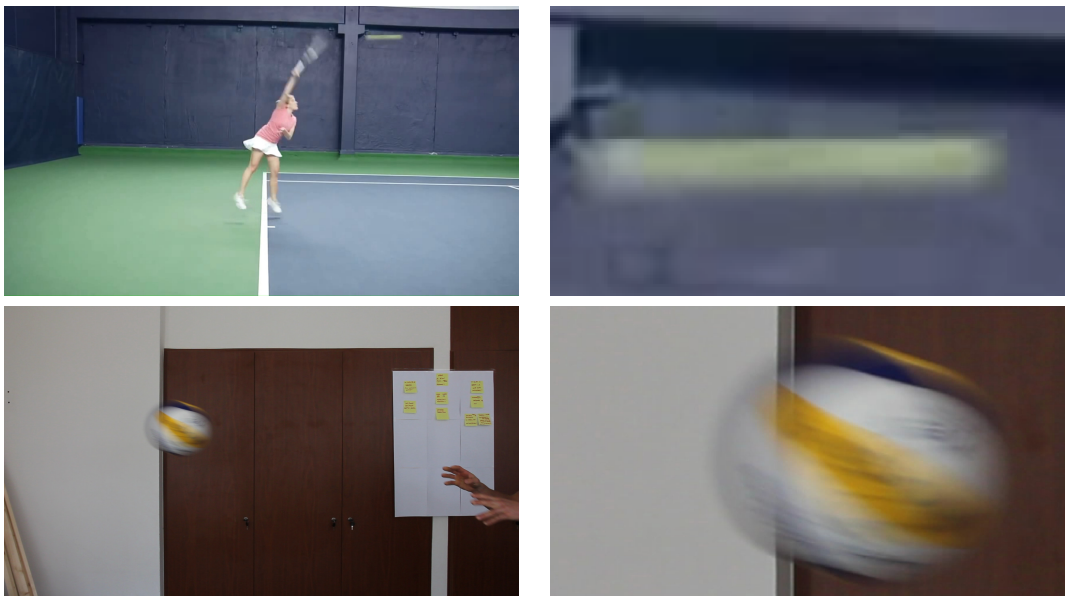


Figure 1.2. Examples of fast-moving objects in the FMO data set (see section 6.2 for more information). First row: a tennis ball. Second row: a volleyball.

2. Fast-moving objects

A digital video camera functions by periodically exposing its sensor array to visible light. Each of the exposures produces an image, commonly referred to as a *frame*, and lasts for a set time known as *exposure time*. Although cameras may automatically adjust exposure time to accomodate changes in lighting conditions, for short sequences it is reasonable to assume that exposure time is constant. The frequency at which frames are produced, also known as *frame rate*, is typically constant as well.

To produce high-resolution images, the sensor array is composed of a large number of small, independent light sensors. In the resulting image, we refer to the smallest independent measurements as *pixels*, although a pixel might itself be produced by multiple sensors. During exposure, each of the sensors accumulates charge as it is being excited by photons of a selected wavelength. In radiometry, *irradiance* is the power received by a surface per unit area; one may consider sensor functionality as integration of irradiance (that the sensor is exposed to) over time. Assuming constant irradiance coming into the camera, the value reported by the sensor is directly proportional to exposure time, implying that exposure must be non-zero in order to capture any footage.

Let us operate in the camera frame of reference, i.e. we say that an object moves if it moves relative to the camera. Given the integrating nature of the sensors, objects that move during exposure appear different to the objects that stay still. A particular spot on a moving object may be captured by different pixels at different times, making the surface details indistinct. A particular pixel might receive irradiance from the moving object for only a fraction of exposure time; for the rest of the time, irradiance is received from other objects, rendering the moving object semi-transparent. Collectively, the effects due to object motion during exposure are known as *motion blur*.

Rozumnyi et al.[1] define a *fast-moving object* (FMO) as an object that during exposure moves a distance exceeding its size, where distance and size are measured in image plane, and size is measured in the apparent direction of the motion. In other words, a fast-moving object is one that would not be transparent at all if it was still (and opaque), but is subject to so much motion blur that it appears semi-transparent in its entirety.

Assume a linear color space $\mathcal{L}$, and denote $I_t : \mathcal{P} \to \mathcal{L}$ the frame at time t, where $\mathcal{P}$ is the set of valid pixel coordinates. Let us define *foreground* $[\mathcal{H}_t F] : \mathcal{P} \to \mathcal{L}$ as the image produced by integrating only the light directly bounced off or emitted from FMOs in I_t. Likewise, let us define *background* $B_t : \mathcal{P} \to \mathcal{L}$ as the hypothetical image that would be captured if none of the FMOs in I_t were directly visible, although the environment would still be subject to indirect lighting and shadows produced by the objects. Finally, let us define *mask* $[\mathcal{H}_t M] : \mathcal{P} \to \mathbb{R}$ as the indicator function of FMOs in I_t. Image formation of I_t is modelled using the following equation [1]:

$$\forall x \in \mathcal{P} : I_t(x) = (1 - [\mathcal{H}_t M](x))B_t(x) + [\mathcal{H}_t F](x)$$

In graphics terms, I_t is the result of alpha blending FMOs on top of B_t, with $[\mathcal{H}_t M]$ being the opacity (alpha) of the FMOs, and $[\mathcal{H}_t F]$ being the color of the FMOs with pre-multiplied alpha.

We use the operator $\mathcal{H}_t$ to denote the complex transformation of moving 3-D objects

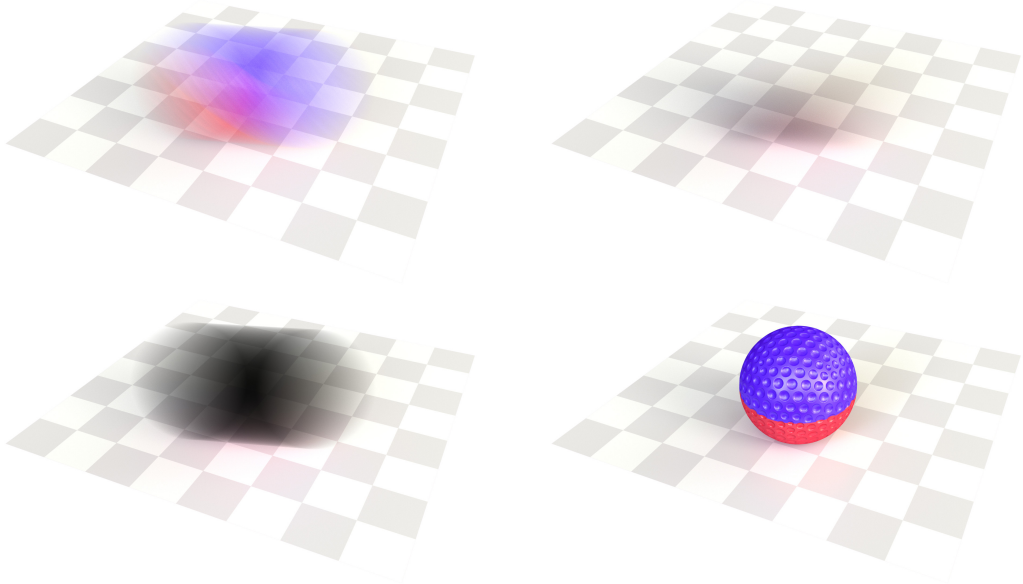


Figure 2.1. A synthetic example of our image formation model. Top left: $I_t(x)$. Top right: $B_t(x)$. Bottom left: $(1 - [\mathcal{H}_t M](x))B_t(x)$. Bottom right: a hypothetical appearance of $I_t(x)$ when exposure time approaches zero.

into still 2-D objects in the image plane, including perspective transformation and motion blur. For each FMO in the frame, the following parameters influence $\mathcal{H}_t$:

- The shape and appearance of the FMO.
- The exact trajectory of the FMO throughout exposure, including orientation.
- The lighting conditions along the trajectory.

The task at hand is to recover all the parameters of $\mathcal{H}_t$, given the frame I_t. Unfortunately, such a problem is impossible to solve in the general case; there is too much ambiguity introduced by the parameters. Additional assumptions need to be put in place in order for the problem to be feasible, and one can select the assumptions in a variety of ways. This choice is critical, as the algorithm that can tackle the problem is very much determined by the assumptions that alleviate its complexity. Following are the assumptions that we use:

1. The shape of a FMO is that of a sphere.
2. A FMO is subject to no perceptible deformation.
3. In terms of appearance, the surface of a FMO is homogeneous.
4. Along the trajectory of a particular FMO, the lighting conditions are constant.

Note that points 1., 2. and 3. imply that the orientation of a FMO does not influence its appearance. With the above assumptions in place, the unknowns regarding each FMO are limited to: radius, trajectory, optical properties of its surface, and lighting conditions along its trajectory. The limitations are significant, but still allow us to explore a broad range of situations in the domain of sports and elsewhere. In addition to

the general assumptions stated above, the algorithm described in chapter 3 establishes additional requirements on I_t that need to be satisfied.

Our goal was to develop a method that, given the assumptions above, uncovers a subset of parameters of $\mathcal{H}_t$, specifically the radius, trajectory, and velocity of each FMO in I_t. Unfortunately, even this simplified task is problematic due to the ambiguity of camera projection. We go around that fact by introducing additonal prior knowledge about the FMOs: either the radius of the FMOs must be known, or the distance of the FMOs to the camera plane must be known. Apart from the properties of tracked objects, background B_t is also recovered as a side product of our method.

3. An efficient algorithm for fast-moving object detection

The goal was to adapt the algorithm described in [1] for real-time operation on mobile. Although the initial intention was to mimic the original method with no significant changes, it quickly became apparent that this would be impossible due to performance constraints. The result is that our approach is entirely different from the one described in [1], however they do share most of the assumptions about input images. It is important to note that while [1] describes a three-stage procedure consisting of detection, re-detection, and tracking, our algorithm only has a single stage corresponding to the original detection phase. Algorithm 1 outlines the steps that constitute the method.

Algorithm 1 An efficient algorithm for fast-moving object detection.

1: **for each** t **do**
2: obtain image I_t
3: calculate image $\hat{I}_t$ by reducing the resolution of I_t
4: calculate background B_t from $\hat{I}_t$, $\hat{I}_{t-1}$, $\hat{I}_{t-2}$
5: calculate binary difference image D_t from $\hat{I}_t$, B_t
6: generate the set of strips $\mathcal{S}_t$ from D_t
7: generate the set of components $\mathcal{C}_t$ from $\mathcal{S}_t$
8: generate the set of objects $\mathcal{O}_t$ from $\mathcal{C}_t$
9: find matches $\mathcal{M}_t$ by matching objects in $\mathcal{O}_t$ to objects in $\mathcal{O}_{t-1}$
10: detect fast-moving objects by analyzing $\mathcal{M}_t$, $\mathcal{M}_{t-1}$, $\mathcal{O}_t$, $\mathcal{O}_{t-1}$, $\mathcal{O}_{t-2}$.
11: **end for**

An implementation of algorithm 1 is publicly available in the C++ code repository [10]. Naturally, the work on the project was an iterative process, and the approach changed significantly over time. The following is a description of the latest version of the algorithm; although the previous versions are being maintained in the code base, they shall not be discussed here.

3.1. Assumptions

In addition to the assumptions stated in chapter 2, algorithm 1 establishes extra requirements on the input image I_t:

1. I_t is a part of a video sequence, and the preceding frames I_{t-1} and I_{t-2} are available.
2. There is a significant image-plane distance between all FMOs in I_t and any other moving objects. Consequently, there is no overlap among FMOs in I_t.
3. The direction of motion of FMOs in I_t is roughly horizontal.
4. Motion of FMOs in I_t is roughly linear.
5. The relative motion between the camera and the environment is small.

By small relative motion we mean that the scene in the image appears predominantly still. A large amount of real-world video sequences have this property, because in order to preserve detail, it is desirable to keep the amount of motion blur to a minimum. This is achievable with either a hardware approach, such as mounting a camera on a tripod, or via software stabilization of the captured video stream. Algorithm 1 does not stabilize the input video, however it should be noted that real-time software stabilization on mobile devices is possible.

3.2. Input

Algorithm 1 receives a stream of images from a camera or a video file. Multiple color formats are accepted, including grayscale, RGB, and YUV. Using grayscale inputs improves performance, while other color formats allow for better detection. Before any further processing, the input images are downscaled to a suitable resolution; to achieve that, a subsampling operation is used which reduces each dimension of the image by a factor of 2. Decimation reduces groups of four neighboring pixels into one by averaging.

Let us denote $h(I)$ the height of image I. The image $\hat{I}_t$ is the result of the fewest repeated subsampling steps of I_t, so that the condition $h(\hat{I}_t) < h_{\max}$ holds true, where $h_{\max}$ is a parameter. The advantage of reducing the image size is that it is significantly faster to do further processing; the steps that immediately follow have an asymptotic computational complexity of $O(n)$, where n is the number of pixels. Therefore, subsampling twice (which is what we usually do) will speed up the following steps approximately by a factor of 16. On the other hand, the reduction in size makes it more difficult to detect small objects.

3.3. Background subtraction

In this step, the background is isolated and used to detect areas of the image $\hat{I}_t$ that contain fast motion. The background $B_t(x)$ is found by calculating the per-pixel median of the three latest consecutive frames:

$$\forall x \in \mathcal{P} : B_t(x) \coloneqq \text{median}\{\hat{I}_t(x), \hat{I}_{t-1}(x), \hat{I}_{t-2}(x)\} \tag{3.1}$$

If the inputs have multiple color channels, the median operation is performed for each channel separately. Given that assumptions 2 and 5 are correct, the median operation will effectively erase fast-moving objects from the image, except when exposure time approaches frame duration. In that case, artifacts may appear where a fast-moving object was present for two of the three frames, but the fact does not impede the subsequent steps of algorithm 1.

As the next step, absolute difference of the input image and the background is calculated per pixel. If there are multiple color channels, the differences are summed to obtain a single value:

$$\forall x \in \mathcal{P} : \tilde{D}_t(x) \coloneqq \begin{cases} |\hat{I}_t(x) - B_t(x)| & \text{in grayscale} \\ \sum_{C \in \{R,G,B\}} |\hat{I}_t^C(x) - B_t^C(x)| & \text{in RGB color space} \\ \sum_{C \in \{Y,U,V\}} |\hat{I}_t^C(x) - B_t^C(x)| & \text{in YUV color space} \end{cases} \tag{3.2}$$

In this manner, an image is obtained with values corresponding to the amount of color change in each pixel, which we interpret as motion. Of course, it is possible to

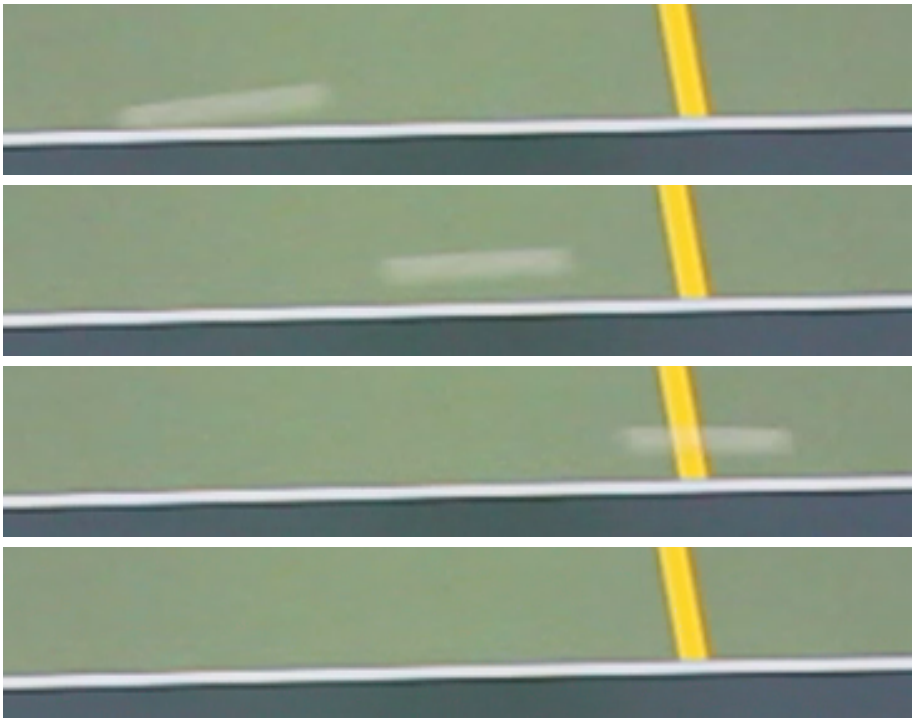


Figure 3.1. An example of background isolation using the formula 3.1 in the RGB color space. From top to bottom: $\hat{I}_{t-2}$, $\hat{I}_{t-1}$, $\hat{I}_t$, B_t.

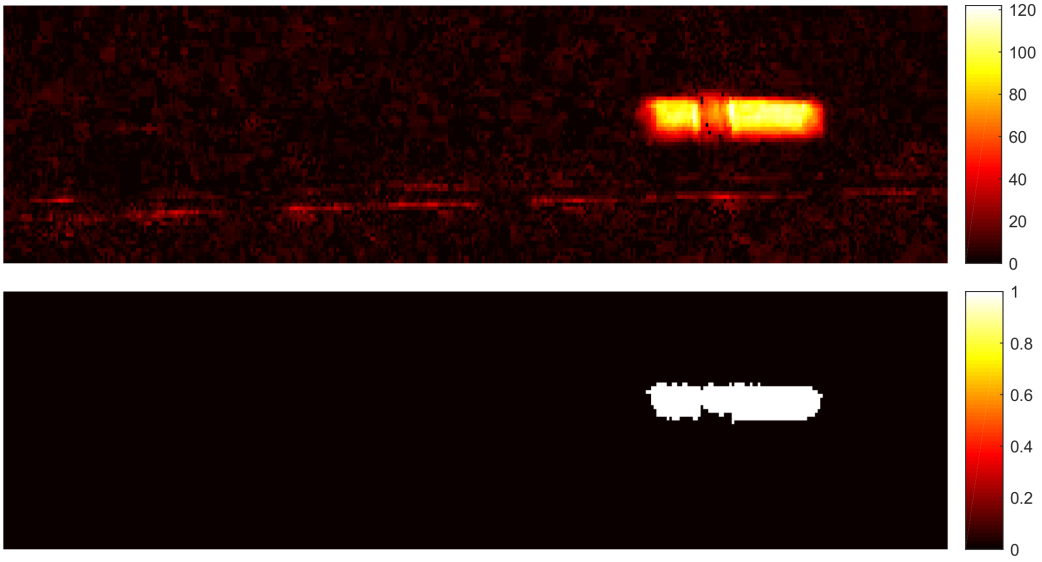


Figure 3.2. An example of motion detection using formulas 3.2 and 3.3. Top: $\tilde{D}_t$. Bottom: D_t.

observe color change without motion, due to sensor noise, camera movement, or viewing objects that change color, e.g. a computer screen. We rely on the fact that none of these become a significant factor, and thus motion is the main reason for color change. Moreover, we rely on the color of the fast-moving objects to be distinct from the color of the background; should they have a similar color, the motion will not be detected. Next, we apply a threshold to classify pixels as containing motion or otherwise:

$$D_t(x) \coloneqq \begin{cases} 1 & \text{if } \tilde{D}_t(x) > \Delta \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

The threshold Δ is adaptive, being modified in order to maintain a certain ratio of noise pixels. For details on how noise is detected, see the next section. See appendix A for related parameters and their defaults.

3.4. Strip generation

In this step, we find image regions in D_t with a width of 1 pixel; we call these *strips*. We are looking for strips that contain motion, i.e. the values in the region are mostly equal to 1. Algorithm 2 describes our strip generation method, with $w(D_t)$ and $h(D_t)$ denoting width and height of the difference image, respectively, and ν_h and κ_y being parameters. What the algorithm effectively does is it processes each column of D_t separately; for each column c, a run-length encoding e is calculated. The following invariants are established: e always begins with a 0-valued run, thus the values of runs need not be stored; after e is constructed, e also ends with a 0-valued run.

Runs shorter than the minimum height ν_h are disregarded (see line 11) in order to ignore noise. The loop starting at line 23 inspects e, reporting 1-valued runs as strips; only strips that maintain a minimum gap κ_y from neighboring strips are reported (see line 24). This is where we rely on assumption 2, which states that there should be a significant gap between a FMO and any other moving entities. By removing objects that are not isolated in this manner, we hope to reduce the amount of slow motion that is being processed by the following steps.

Aside from reporting strips, algorithm 2 also provides the noise level ν based on the total number of times a short run has been ignored (see line 13). This value is used to adapt the differentiation threshold Δ (see equation 3.3) to changing lighting conditions in the following manner. Every ν_N frames, the median of last ν_N reported values of ν is calculated to obtain a robust noise level estimate. If the median is outside a set range $(\nu_{\min}, \nu_{\max})$, we update the threshold Δ to adjust the sensitivity of motion detection. The quantities ν_N, $\nu_{\min}$, $\nu_{\max}$ are parameters.

Let us define the following properties describing a strip s found in frame t: the point $(c_x(s), c_y(s))$ is the location of the strip center in I_t, while $h_x(s)$, $h_y(s)$ denote half of the width and height in I_t, respectively. Note that from this point on, we operate in source image coordinates and dimensions, as opposed to the downscaled ones. All strips found in frame t form the set $\mathcal{S}_t$.

3.5. Component generation

In this step, components are formed by joining strips that have been found in D_t. A greedy algorithm is used to construct the mapping $\sigma : \mathcal{S}_t \to \mathcal{S}_t$; we call the strip $\sigma(s)$ the *successor* of s. The greedy aspect is that a strip may become a successor only

Algorithm 2 Strip generation.

let $n \coloneqq 0$
let $p \coloneqq \max\{\nu_h, \kappa_y\}$
for each column c **in** D_t **do**
  let e be an empty vector
  $\texttt{push\_back}(e, -p)$
  if $\texttt{at}(c, 0) \neq 0$ **then**
    $\texttt{push\_back}(e, 0)$
  end if
  for i **in** 1 **to** $h(D_t) - 1$ **do**
    if $\texttt{at}(c, i) \neq \texttt{at}(c, i-1)$ **then**
      if $i - \texttt{back}(e) < \nu_h$ **then**
        $\texttt{pop\_back}(e)$
        $n \coloneqq n + 1$
      else
        $\texttt{push\_back}(e, i)$
      end if
    end if
  end for
  if $\texttt{size}(e)\%2 = 0$ **then**
    $\texttt{push\_back}(e, h(D_t))$
  end if
  $\texttt{push\_back}(e, h(D_t) + p)$
  for i **in** 0 **to** $\texttt{size}(e) - 2$ **advance by** 2 **do**
    if $(\texttt{at}(e, i+1) - \texttt{at}(e, i) \geq \kappa_y) \wedge (\texttt{at}(e, i+3) - \texttt{at}(e, i+2) \geq \kappa_y)$ **then**
      report strip in column c spanning rows $\texttt{at}(e, i+1)$ to $\texttt{at}(e, i+2)$
    end if
  end for
  report $n/(w(D_t)h(D_t))$ as the noise level ν
end for

Algorithm 3 Greedy strip merging.

let S be a list of all strips in $\mathcal{S}_t$ ordered by $f(s)$
let $R \coloneqq \emptyset$
for each s **in** S **do**
  let $T \coloneqq \emptyset$
  for each s' **in** S **do**
    if $s' \notin R \wedge p(s, s')$ **then**
      $T \coloneqq T \cup \{s'\}$
    end if
  end for
  if $T = \emptyset$ **then**
    $\sigma(s) \coloneqq s$
  else
    $\sigma(s) \coloneqq \arg\min_{s' \in T} f(s')$
    $R \coloneqq R \cup \{\sigma(s)\}$
  end if
end for

once, after that it is disregarded. Algorithm 3 shows the general outline. Note that the function σ is determined by an ordering function f and a predicate p, which we define as follows: the lexicographic ordering $(c_x(s), c_y(s))$ is used as f; and $p(s, s')$ is considered true if the following two conditions are true:

- $0 < c_x(s') - c_x(s) < \kappa_x h(I_t)$, where κ_x is a parameter, i.e. the successor must be to the right from the strip, and some distance between s and s' is allowed.
- $|c_y(s') - c_y(s)| < h_y(s') + h_y(s)$, i.e. if the strips were in the same column, they would overlap.

From the set of potential successors that satisfy the conditions above, the strip that comes first in the lexicographic ordering $(c_x(s), c_y(s))$ is selected. If and only if there is no valid successor of s, $\sigma(s)$ is set to s. Next, we use σ and the set R to form the set $\mathcal{C}_t$ as follows:

$$\Omega(s) = \{s\} \cup \Omega(\sigma(s)) \tag{3.4}$$

$$\mathcal{C}_t = \{\Omega(s) | s \in \mathcal{S}_t \wedge s \notin R\} \tag{3.5}$$

Let us call the elements of $\mathcal{C}_t$ *components* due to their similarity to connected components, particularly for low values of κ_x. A component is a set of strips, and each strip belongs to exactly one component. It is trivial to find all components once algorithm 3 is complete, simply by following the successor function σ from each strip that is not itself a successor ($s \notin R$).

3.6. Component description

In this step, useful properties of components are calculated, which are later used in object matching (see section 3.7). We obtain these for each component K in $\mathcal{C}_t$:

- $A(K)$ — area of strips in component K, $A(K) = 4 \sum_{s \in K} h_x(s) h_y(s)$.
- $\mathcal{H}(K)$ — convex hull of K, i.e. the smallest convex set of points in I_t that contains all strips in K. Based on assumption 4 and the assumption that FMOs are spherical, the shape of a FMO is convex. By analyzing the convex hull instead of the component itself, the algorithm becomes more robust; gaps in the component have less influence on the calculated quantities.
- $A(\mathcal{H}(K))$ — area of the convex hull.
- $\vec{c}(K)$ — centroid of the convex hull of K.
- $\vec{v}_1(K)$, $\vec{v}_2(K)$, $\lambda_1(K)$, $\lambda_2(K)$ — unit-length eigenvectors and associated eigenvalues of the covariance matrix of $\mathcal{H}(K)$, $\vec{v}_1(K)$ being the primary principal direction. The covariance matrix is estimated by uniformly sampling points that lie in $\mathcal{H}(K)$.
- $d_1(K)$, $d_2(K)$ — estimated width and height of $\mathcal{H}(K)$. Let us assume that the shape of a component is roughly rectangular, therefore $\lambda_i(K)$ are variances of uniform distributions. For a uniform distribution, variance is equal to $\frac{1}{12}w^2$, where w is the length of the interval between its boundaries. The estimated dimensions are thus easily obtained using the following formula:

$$i \in \{1, 2\} : d_i(K) = \sqrt{12 \lambda_i(K)}$$

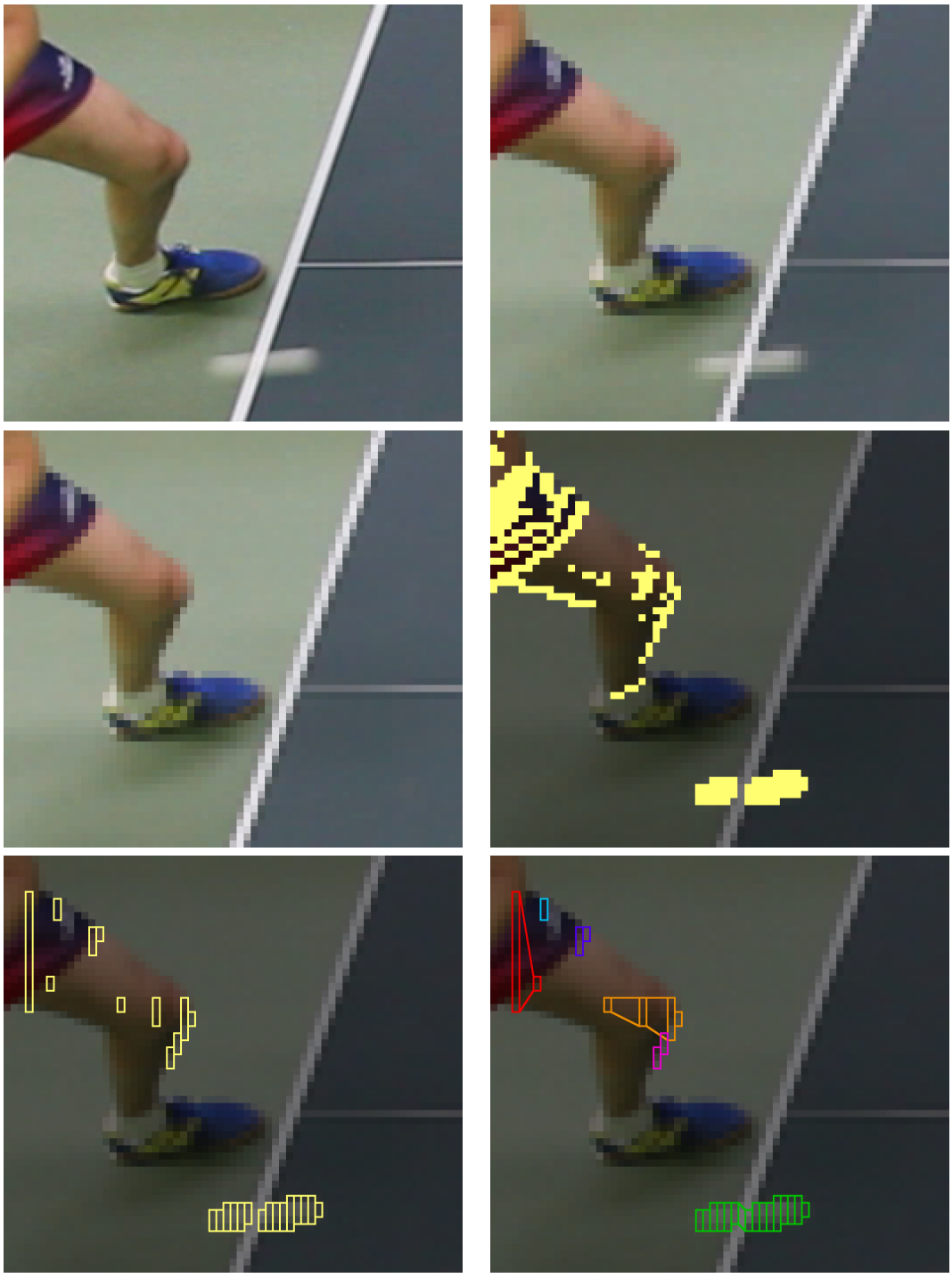


Figure 3.3. An example of subsampling, background subtraction, strip generation and component generation (sections 3.2 to 3.5). Top row: input image I_t, downscaled image $\hat{I}_t$. Middle row: background B_t, binary image D_t. Bottom row: strips $\mathcal{S}_t$, components $\mathcal{C}_t$.

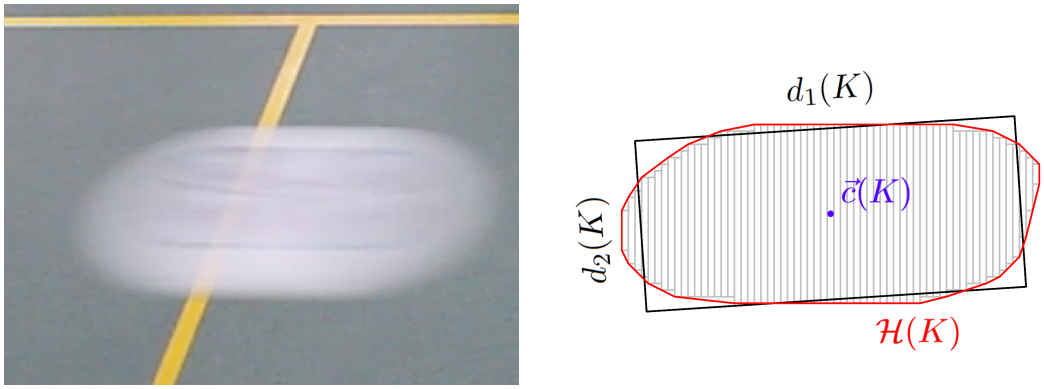


Figure 3.4. An example of component description (see section 3.6). The object is a frisbee disc. Left: input image. Right: strips (gray), convex hull (red), centroid of convex hull (blue), estimated dimensions (black).

- $d_R(K) = d_1(K)/d_2(K)$ — aspect ratio.

At this point, some of the components are discarded from further processing in order to omit components with a low probability of being a FMO. There are four conditions that a component K must pass in order to be retained:

$$|K| > \zeta \qquad \frac{A(K)}{A(\mathcal{H}(K))} > \zeta_A \qquad d_R(K) > \zeta_d \tag{3.6}$$

$$\forall K' \in \mathcal{C}_{t-2} : ||\vec{c}(K) - \vec{c}(K')|| > \frac{\zeta_{\vec{c}}}{d_1(K')} \tag{3.7}$$

where ζ, ζ_A, ζ_d, $\zeta_{\vec{c}}$ are parameters. Effectively, ζ removes noise by rejecting components with a low number of strips, ζ_A disallows components with non-convex shapes, and ζ_d requires that a component has a elongated, stroky shape. Finally, $\zeta_{\vec{c}}$ removes components close to any previously found component in frame $t-2$, which is useful for removing artifacts in B_t caused by long exposition time. The components in $\mathcal{C}_t$ that pass the above conditions form the set $\mathcal{O}_t$ and are called *objects*.

3.7. Object matching

In this step, objects in the current frame are assigned to similar objects in the previous frame. There are multiple criteria that influence the matching, such as distance of endpoints, similarity in area and aspect ratio, etc. A simple algorithm is used, testing each pair of objects with a complexity of $O(|\mathcal{O}_t||\mathcal{O}_{t-1}|)$. Algorithm 4 shows the general outline; for each of the objects in $\mathcal{O}_t$, at most one object in $\mathcal{O}_{t-1}$ is assigned as the predecessor based on the viability predicate $p(o_0, o_1)$ and the cost function $f(o_0, o_1)$ being minimized.

Let us define cost functions f_α, f_β, f_γ, f_δ as follows:

$$f_\alpha(o_0, o_1) = \frac{\max\{d_R(o_0), d_R(o_1)\}}{\min\{d_R(o_0), d_R(o_1)\}} \tag{3.8}$$

$$f_\beta(o_0, o_1) = \frac{\max\{A(\mathcal{H}(o_0)), A(\mathcal{H}(o_1))\}}{\min\{A(\mathcal{H}(o_0)), A(\mathcal{H}(o_1))\}} \tag{3.9}$$

Algorithm 4 Object matching.

let $\mathcal{M}_t := \emptyset$
for each o_0 **in** $\mathcal{O}_t$ **do**
  let $T := \emptyset$
  for each o_1 **in** $\mathcal{O}_{t-1}$ **do**
    if $p(o_0, o_1)$ **then**
      $T := T \cup \{o_1\}$
    end if
  end for
  if $T \neq \emptyset$ **then**
    $o_1 := \arg\min_{o \in T} f(o_0, o)$
    $\mathcal{M}_t := \mathcal{M}_t \cup (o_0, o_1)$
  end if
end for

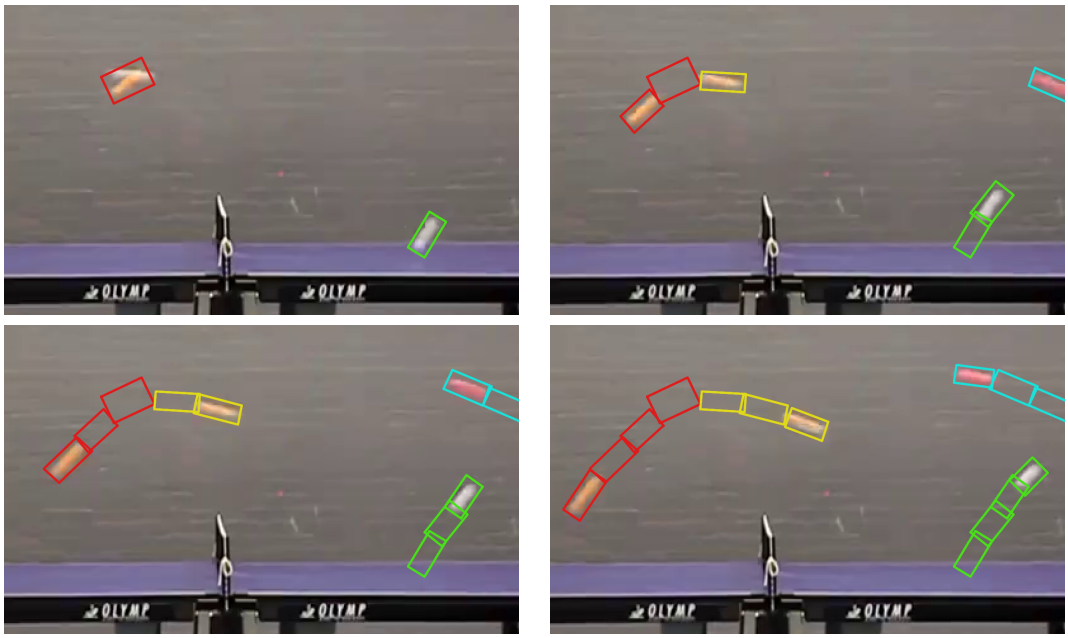


Figure 3.5. An example of object matching in 4 consecutive frames. Distinct objects are distinguished by color. In the first frame, two FMOs overlap.

$$f_\gamma(o_0, o_1) = ||\vec{m}|| \tag{3.10}$$

$$f_\delta(o_0, o_1) = \begin{cases} 1 \text{ if } d_R(o_0) < \tau_R \vee d_R(o_1) < \tau_R \\ \max\{|\sin(\angle\vec{v}_1(o_0)\vec{m})|, |\sin(\angle\vec{v}_1(o_1)\vec{m})|\} \text{ otherwise} \end{cases} \tag{3.11}$$

where $\vec{m} = \vec{c}(o_1) - \vec{c}(o_0)$ is the centroid motion vector, and τ_R is a parameter. Effectively, f_α indicates inconsistency of aspect ratios, f_β increases with differing area of convex hulls, f_γ is the distance of convex hull centroids, and f_δ detects changes in object orientation, provided that the direction of the objects is unambiguous. Given the cost functions above, $p(o_0, o_1)$ and $f(o_0, o_1)$ are defined as follows:

$$p(o_0, o_1) \Leftrightarrow f_\alpha(o_0, o_1) < \tau_\alpha \wedge f_\beta(o_0, o_1) < \tau_\beta \wedge f_\gamma(o_0, o_1) < \tau_\gamma \wedge f_\delta(o_0, o_1) < \tau_\delta \tag{3.12}$$

$$f(o_0, o_1) = \alpha f_\alpha(o_0, o_1) + \beta f_\beta(o_0, o_1) + \gamma f_\gamma(o_0, o_1) + \delta f_\delta(o_0, o_1) \tag{3.13}$$

There is a total of 9 parameters in the matching step: thresholds τ_α, τ_β, τ_γ, τ_δ, τ_R, and factors α, β, γ, δ. Finding good values for these has been crucial in order to increase precision of the detection algorithm while preserving sensitivity. A run of algorithm 4 produces the set of matches $\mathcal{M}_t$.

3.8. Fast movement detection and output

In this step, fast-moving objects are detected based on cached data from the last three frames. Only the objects in I_t that have an associated object in I_{t-2} are considered; more precisely, we construct the set $\mathcal{M}^2$:

$$(o_0, o_1, o_2) \in \mathcal{M}^2 \Leftrightarrow (o_0, o_1) \in \mathcal{M}_t \wedge (o_1, o_2) \in \mathcal{M}_{t-1}$$

Next, let us consider each triplet (o_0, o_1, o_2) in $\mathcal{M}^2$ as a potential fast-moving object. According to assumption 4, the motion of a FMO is supposed to be roughly linear. Obviously, this is a significant simplification, as trajectories in Earth's gravitational field are known to be parabolic. For our needs, however, this model is sufficient due to the objects moving at high velocities over a relatively short period of time. To verify the model, the expected position at time $t-2$ is calculated, given the position at $t-1$ and t; the actual position is then used to find the error $\vec{e}$:

$$\vec{e}(o_0, o_1, o_2) = \vec{c}(o_0) - 2\vec{c}(o_1) + \vec{c}(o_2)$$

Once the error vector $\vec{e}$ is acquired, its length weighted by estimated object size is used as validation of linear motion, which is the last requirement for declaring the objects as fast-moving:

$$(o_0, o_1, o_2) \in \mathcal{M}^2 : \frac{|\vec{e}(o_0, o_1, o_2)|}{\text{median}\{d_1(o_0), d_1(o_1), d_1(o_2)\}} < \epsilon \Rightarrow o_0,\ o_1,\ o_2 \text{ are FMOs} \tag{3.14}$$

where ϵ is a parameter.

The very last step is to report the detected FMOs. There is a two-frame latency introduced, i.e. in frame t, FMOs present in I_{t-2} are reported. This reflects the fact that objects might be detected up to two frames after they are observed in the input image. Special care is taken not to report a single object multiple times. Information reported about a FMO o includes: centroid of convex hull $\vec{c}(o)$, length $d_1(o)$, diameter $d_2(o)$, and, optionally, a listing of all object pixels in I_{t-2}.

4. Implementation

4.1. Overview of the products

This section lists and details the C++ and Java applications that have been created as part of this thesis. These include: a C++ library, an Android application, and a set of tools for the desktop. Source code for all the described software is available in repositories [10, 11].

4.1.1. Cross-platform library

The core library is at the heart of both the mobile and the desktop applications. Contained within are multiple algorithms for fast-moving object detection and an assortment of related data structures. It is written in portable C++14 in order to support as many architectures and operating systems as possible. The library is known to work with the instruction sets x86 and ARM, operating systems Windows, Linux and Android, and compilers MSVC, GCC and Clang. The following configurations have been tested:

- Android, ARMv7/ARMv8, GCC
- Windows, x86/AMD64, MSVC
- Linux, x86/AMD64, GCC/Clang
- Linux, ARMv7, GCC

To enable portability and simplify the build process, the set of dependecies is kept to a minimum. The only prerequisite is OpenCV [12], which is available as pre-compiled binaries for a wide range of platforms, and popular Linux distributions provide it via package repositories. Additionally, client code of the library is completely isolated from OpenCV header files, therefore the client code can be linked without OpenCV present; we use this fact in order to simplify the Android build. CMake [13] is used to generate build files for all targeted platforms and compilers.

Let us go through various features from the point of view of a library user. In the following discussion, all names of functions and classes are in the `fmo` name space unless specified otherwise. There are multiple detection algorithms available in the library: a common interface called `Algorithm` hides the complex details of the various object detection implementations. To make the algorithms easily interchangeable, all operations needed for object detection are covered by virtual member functions. To enable a common interface, all algorithms use the same set of parameters represented by an `Algorithm::Config` object, although a particular algorithm may make use of only some of the parameters. Subclasses of `Algorithm` are identified by strings and are automatically registered; the factory function `Algorithm::make()` is used to obtain an instance of the correct class.

To minimize code duplication among multiple variants of the object detection algorithm, most data structures and procedures are shared throughout the library. To satisfy the ubiquitous need to allocate image buffers, the `Image` class is used, which

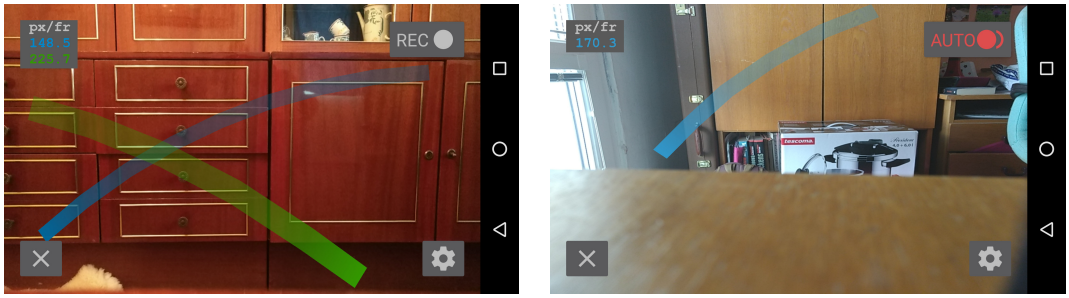


Figure 4.1. Left: screen capture of the Android application, showing the video feed and the traces of the last two detected objects. Right: the application allows to capture videos. In automatic recording mode, capture is triggered by an ongoing detection.

is also the primary means of isolation of user code from OpenCV. The related class `Region` does not own the image data, and is able to refer to a rectangular region within an image. Images may have a number of different formats, including single-channel grayscale, triple-channel interleaved BGR or YUV, and YUV with 4:2:0 subsampled chroma [14]. By YUV we always mean the YC_bC_r color space; though inaccurate, this unfortunately is how it is commonly done in the industry.

The `Mat` interface is the common superclass of `Image` and `Region`. Using this interface, a selection of useful image processing fuctions is provided, for example: `convert()` allows to convert image formats; `greater_than()` applies a threshold to a grayscale image; `absdiff()` calculates per-pixel absolute difference; `median3()` provides per-pixel median of three images; and others. Certain procedures cannot be effectively implemented as free-standing functions, because potentially large image buffers might need to be cached for performance. In that case we provide function objects, such as `Decimator`, which scales images down, halving their dimensions; or `StripGen`, which generates vertical strips as described in section 3.4.

All of the essential data structures and functions are covered by unit tests. We use the Catch testing framework [15], which is easy to integrate into the source code without introducing any extra build-time dependencies. This is due to it being distributed as a single header file under the permissive Boost software license. For a medium-sized project such as this library, unit testing has proven crucial in order to establish confidence in low-level functionality before building higher-level entities. The test runner application is a part of the desktop front-end (see section 4.1.3).

4.1.2. Android front end

The Android application is a tool for live demonstration of algorithm 1. It is targeted at operating system version 5.0 (Lollipop) and later, currently supporting about 70.3% of Android devices on the market, based on data collected in April and May 2017 [16]. The application requires the following tools to build: Android software development kit, Android native development kit (NDK), and Android Studio integrated development environment. All of the required tools are currently freely available for download, provided by Google, Inc., and Open Handset Alliance.

The main programming language on Android is Java, resulting in the binaries being largely architecture-independent. In order to use C++ code, however, multiple native libraries need to be built, one for each architecture supported by Android — currently there is 7 of them, encompassing multiple variants of x86, ARM and MIPS. To that end, NDK provides cross-compilers for all the supported platforms. In order to avoid

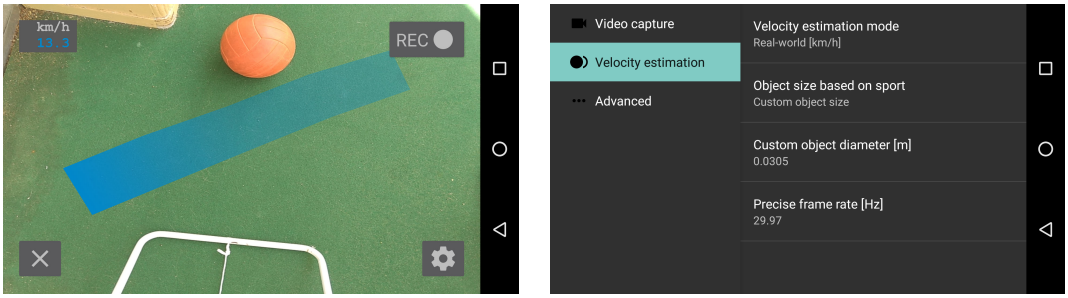


Figure 4.2. The application can be configured to show estimated real-world velocities. Left: speed of a ball is reported in kilometers per hour. Right: to enable the estimate, diameter of fast-moving objects is specified in the settings menu.

packaging all the sets of native code libraries into a single Android application package (APK), multiple architecture-specific packages have been created, each containing only a single set of libraries. Moreover, the architectures ARMv7 and ARMv8 are largely prevalent, which allowed us to focus on accelerating ARM code (see chapter 5 for details on performance).

For the most part, the Android application is implemented in Java, with the core C++ library being included as a shared library. Calling C++ code from Java and vice versa is facilitated by the Java native interface (JNI). The application is multi-threaded, with separate threads dedicated to various tasks that may run simultaneously. As shown in figure 5.3, the concurrency scheme consists of five threads: the main thread, which is the only one allowed to modify the UI; the thread for video capture, which awaits new camera frames and draws them into buffers via OpenGL; the thread for video encoding, which produces a H.264-encoded video stream; the thread for video saving, which stores encoded video into files while recording; and the detection thread, which runs algorithm 1.

Upon launch, the user is presented with the main view, which consists of a camera feed in the background, overlaid by user interface controls. See figure 4.1 for screen captures of the main view. The stream of camera images is used as input into algorithm 1, and the traces of detected objects are shown on screen. To avoid cluttering the view, only the last two detections are shown, color-coded by the direction in which the object had moved — green color indicates a left-to-right motion, while blue indicates the converse. For convenience, a clear button in the bottom left allows to erase the traces immediately.

In addition to observing the FMO detections, the user is able to record videos containing the raw camera feed, i.e. without any UI or detection visualization. There are two modes of recording: in manual mode, the user taps the record button to start and stop recording; in automatic mode, videos are recorded whenever FMOs are detected, along with 2 seconds of extra footage both before and after the event. While automatic recording is enabled, video is being continuously encoded and stored into a circular buffer. This is in contrast with the desktop front end (see section 4.1.3), where the circular buffer contains uncompressed frames; for mobile, we deem this approach too memory-intensive. Additionally, most consumer devices provide a hardware-accelerated H.264 codec, making continuous encoding feasible for real-time use.

As described in chapter 8, algorithm 1 can be used to estimate FMO velocities. The application facilitates this, showing speeds of the detected objects in the top left corner of the main view. By default, the speeds are measured in image plane, the unit being pixels per frame. As shown in figure 4.2, the application can be configured

to display real-world speeds, e.g. in kilometers per hour. For that, the application requires information about exact object size; for convenience, ball sizes in common sports such as tennis and golf are included as presets. An exact frame rate can also be entered; although being marketed as running at 30 Hz, the actual frame rate of the phone cameras varies device-to-device. For example, the integrated camera of Xiaomi Mi3 runs at 29.25 Hz, while Motorola Moto G4 Plus records at 29.97 Hz.

The settings menu is accessed by tapping on the button in the bottom right corner of the main view. Apart from velocity options described above, the settings menu allows to configure the input video stream: the preference for a front- or rear-facing camera can be selected, along with a preferred resolution (HD 720p or Full HD 1080p). Additionally, recording mode can be switched between manual and automatic mode, and the video feed can be configured to refresh only once per 2 seconds. Provided options allow to reduce the processing load, which may be required on slower devices. For example, Samsung J5 (ver. 2016) is not able to maintain 30 Hz operation at 1080p while recording a video, however lowering the resolution to 720p allows to detect and record without restrictions. In addition to the J5, the application has been tested on two other smartphones listed in table 5.1, however the other devices have shown no frame drops even on maximum settings. Indeed, the J5 is the worst-performing model of the three, as demonstrated in figure 5.1.

4.1.3. Desktop front end

The desktop front-end is collection of tools for developing, evaluating and demonstrating the object detection algorithm described in chapter 3. The binaries are not meant to be run on mobile; although they can be built on any architecture, a shell is required to launch the binaries, and a windowing system is needed to show visual output. There are three executables: a test runner, a benchmark runner, and the main application. The main application can be extensively configured using either command-line parameters or configuration files. On start-up, the following options are available:

- Mode selection. There are three modes available: demonstration, development and evaluation. See below for details about each mode.
- Input specification. An input is either a path to a video file, a URL with a video feed (such as one provided by an IP camera), or an identifier of a connected hardware camera. The video stream provided by an input will be passed to the algorithm for processing. Multiple inputs may be specified and will be processed in succession.
- Algorithm selection. Apart from the algorithm described in chapter 3, its previous versions are included for comparison.
- Algorithm configuration, i.e. tweaking algorithm parameters.
- Additional options relevant to a specific mode.

See appendix A for a complete list of parameters. All configuration is optional, except input specification. When no mode is explicitly selected, a default one will be picked based on provided inputs; the demonstration mode will be used when the input is a live camera feed, and the development mode will be used otherwise.

Figure 4.3. Screen captures of the desktop front end. Left: demonstration mode, highlighting detected object traces, and allowing to capture videos. Right: development mode, showing a detailed visualization of the detection procedure.

Demonstration mode

This mode is best suited for observing a live camera feed. There is a minimal amount of UI being shown on screen; the current input image is in the background, and a small status box is being displayed in the top left corner. When an object is detected, its path is highlighted for easier observation. There is also an event counter, which is incremented each time an object is detected after a significant delay; this is useful e.g. for counting the number of times a ball has been thrown in front of a camera. An optional sound signal can be enabled to notify that the event count has been increased.

Demonstration mode also features video capture. There is a manual and an automatic recording mode; in manual mode, the user presses a button in order to start or stop recording. In automatic mode, a cyclic buffer is employed to continuously store the last two seconds of input; no video is being recorded until an object is detected by the algorithm, and the recording ends when two seconds have passed since the last detection. Should there be a new detection occurring while video capture is in progress, the end of the recording is postponed. In this manner, video files are being produced containing object detections surrounded by a two-second margin.

Evaluation mode

This mode allows one to evaluate the quality of the algorithm by processing a series of video files as fast as possible. We can leverage that to quickly experiment with various parameters of the algorithm, with a run of the entire data set being as fast as 15 seconds. The run time depends mainly on the video decoding capabilities of the CPU; we estimate that about 85% of the time is spent reading video files. Evaluation mode has no graphical output — it may be used without a windowing system, which is useful for unattended execution.

For each input file, a ground truth file needs to be specified in order to enable evaluation (see appendix B for details about the ground truth text format). Evaluation proceeds according to our experimental method described in section 6.1, producing one or more events for each frame, of which there are four kinds: TP, TN, FP, and FN. During evaluation, we record all events and save them into a special evaluation report file (also detailed in appendix B). For convenience, additional human-readable information is both added to the evaluation report file and printed on screen once the run is complete. The information includes:

- Values of all options, regardless of whether they were modified by a command-line argument or not.

- A detailed table containing event counts, precision, and recall separately for each input sequence.
- Overall statistics, including average precision and recall.

A previously generated evaluation report can be loaded as baseline, enabling comparison between the previous and the current run. When a baseline file is available, any changes in algorithm performance will be highlighted in the new report, which is useful for manual regression testing and parameter tuning.

Development mode

This mode is best suited for analyzing a video file. There is significantly more information shown than in demonstration mode: the current image is shown in the background, with a visualization of the algorithm superimposed. The exact visualization depends on the selected algorithm; the default algorithm shows the difference image of the current frame vs. the median of three frames, the strips that have been generated in this frame, the objects that have been detected in the last three frames, interconnections between matched objects from consecutive frames, and the pixels of detected fast-moving objects. Additionally, strips that are not part of an object are color-coded to indicate the reason why their connected component was refused.

A ground truth file may be specified in the same fashion as in evaluation mode. If present, the pixels of objects in the ground truth file are displayed and color-coded based on whether they intersect with the pixels of detected fast-moving objects. Additionally, the events TP, TN, FP and FN that occurred in the last frame are shown in the status box in the top left corner. Yet more information is shown if a baseline file is specified.

The amount of on-screen information is impossible to keep track of at normal playback speeds. To enable thorough analysis of algorithm behavior, development mode features playback controls and additonal command-line parameters. Using keystrokes, the video can be paused, stepped one frame at a time, and navigated forward and backward by an increment of 10 frames. The options allow one to change the playback speed and pause when various conditions are met, including: on the first frame, on a frame specified by frame number, whenever there is a false positive or false negative according to ground truth, or whenever a regression or improvement over baseline is detected.

5. Performance

This section details the various optimization techniques used to accelerate the core library and the front ends described in chapter 4. The effort was focused on low-power mobile devices, with the goal of running algorithm 1 in real time, simultaneously with video recording and graphical output.

Despite some of the critical operations being highly data-parallel, we do not leverage GPU acceleration. The reason is that with Android being the primary target platform, there is no portable GPU general-purpose computing (GPGPU) API available. According to Konrad [17], the only GPGPU-capable API since Android version 4.3 is RenderScript, which is an Android-specific framework with a Java interface. Therefore, there is no way to integrate any RenderScript kernels into the C++ core library, let alone write a single computation kernel that runs both on desktop and mobile. OpenCL (Open Computing Language) would be the best tool for our purposes, unfortunately the support for it was dropped in Android 4.3.

5.1. Robust performance measurements and profiling

Precise measurements of actual, real-world performance of the mobile app have been used to evaluate algorithm speed on various devices. The Android operating system is available on a diverse range of hardware; we have picked three phones to test on, with the criteria being:

- a Full HD (1920 by 1080 pixels), 30 Hz camera must be available on the device,
- consumer adoption of the device must be high in order to represent the typical user,
- the above satisfied, the selection must be as diverse as possible, covering various manufacturers, architectures, CPU core counts and clock speeds, etc.

Based on the criteria above, the following devices have been chosen: Samsung J5, Motorola Moto G4, and Xiaomi Mi3. Table 5.1 shows detailed specifications of the phones.

There have been three main means of performance analysis used: live timing, profiling and benchmarking. Live timing measures the time to process a frame by the FMO detection algorithm during real-world use. The results of live timing depend not only on the speed of the algorithm itself, but also on the CPU load caused by the other threads and processes running in parallel. For example, refreshing the preview image in the mobile app is a CPU-intensive operation, and tends to significantly increase the

Name	Architecture	CPU cores	RAM
Samsung J5 (2016)	ARMv7a 32-bit	4 @ 1.2 GHz	2 GB
Motorola Moto G4 Plus	ARMv8 64-bit	4 @ 1.5 GHz + 4 @ 1.2GHz	4 GB
Xiaomi Mi3W	ARMv7a 32-bit	4 @ 2.3 GHz	2 GB

Table 5.1. Specifications of the devices used to test performance.

[ms]	J5	G4	Mi3	PC
median	5.26	3.58	1.97	0.44
95%-quantile	7.0	4.2	2.5	0.5
99%-quantile	10	5	4	1

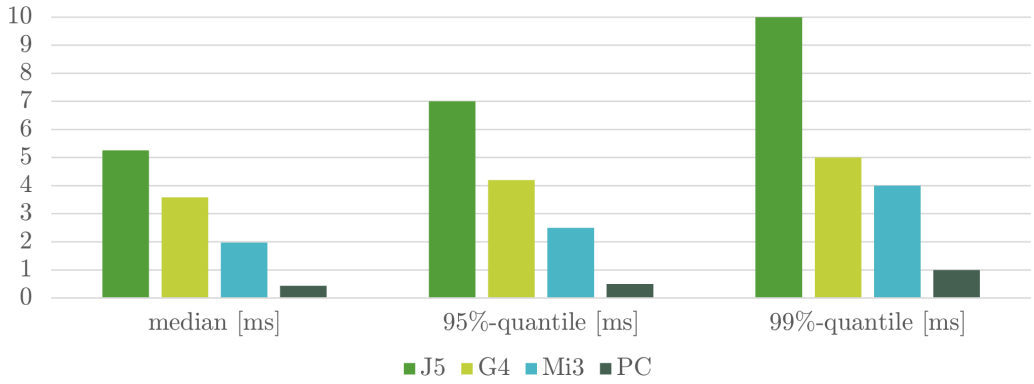


Figure 5.1. A comparison of CPU power of the tested devices (see table 5.1 for details) and a laptop PC with a Intel Core i7-5500U processor and 8 GB RAM. Times were obtained by running the `fmo::StripGen` benchmark. Lower is better.

frame time of the detection algorithm, especially on slower devices such as the J5. This is due to pre-emption; the CPU cores are shared among the running threads, therefore under load the cores tend to spend more time computing other tasks. Apart from indicating algorithm speed, live timing gives a general indication whether real-time operation is viable on the particular device with particular settings.

Profiling allows one to gain insight into how much resources (especially CPU time, but also memory) is being used by the individual components of a program, breaking down to processes, threads, functions and sometimes even individual lines of code. This is useful in order to empirically assess the performance and easily find performance bottlenecks, i.e. parts of the program that are particularly demanding on resources. On mobile, profiling has been used extensively to find bottlenecks in Java code; repeatedly, we have found that the GUI thread (see figure 5.3) does too much work due to user interface updates. Profiling C++ code on Android has proven too problematic to set up, therefore the core library has only been profiled on desktop. Figure 5.2 shows the results of preliminary profiling, before any optimizations took place. Note that the most time-intensive parts of the algorithm were: median of 3, subsampling, differentiation and strip generation. Optimization was thus focused on these parts.

In order to test and compare the performance of various processing steps on all targeted devices, a standard set of performance benchmarks has been made a part of the core library. Benchmarks include all significant image processing operations tested on a Full HD image (1920 by 1080 pixels), allowing one to quickly discover slow operations on various devices. The evaluation may be run both on desktop and on mobile; the desktop front end provides a dedicated executable, and the Android front end provides a graphical benchmarking tool. Both live timing and benchmarking use quantiles of a large number of runs to measure performance robustly, specifically the median is used to obtain the typical run time, while the 95%-quantile and 99%-quantile are representative of worst-case performance. Figure 5.1 shows the result of one of the benchmarks on all tested devices, with a common laptop PC added for comparison.

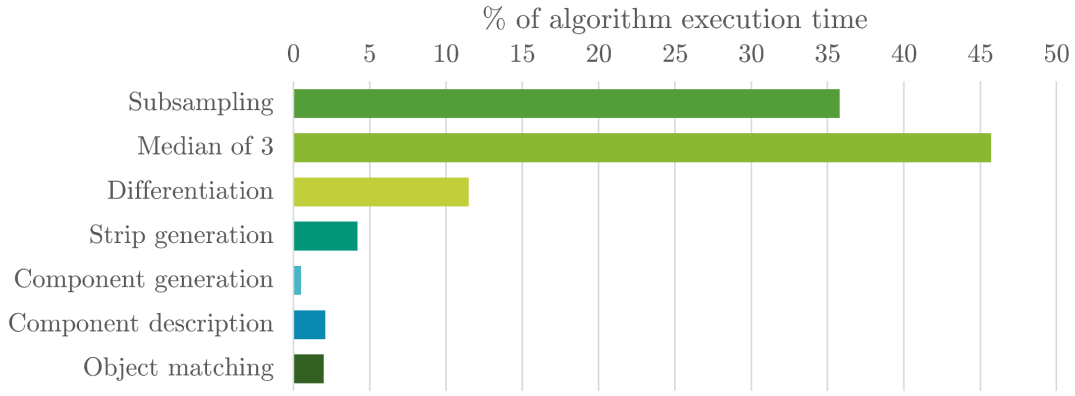


Figure 5.2. The results of profiling algorithm 1 on desktop before any optimizations took place. Compared are run times of various parts of the algorithm, lower is better. The values are percent of total run time. The CPU being used is an AMD FX-6200 with 8 GB of RAM.

5.2. Multi-threading

Leveraging the multi-tasking capabilities of current mobile devices was the most important step to achieving real-time operation of the software. As can be seen in our device selection in table 5.1, the current generation of mid-range to high-end devices tends to favor high core counts over single-core performance. Therefore, good use of threads is key to unlocking full potential of the CPU.

Multi-threading has been employed to achieve both concurrency and parallelism. In terms of concurrency, multiple threads exist that handle different, independent tasks. For example, there is a thread for encoding video, saving recorded video, running the detection algorithm, etc. This helps not only by improving performance once multiple tasks run simultaneously, but it also divides code into smaller, isolated entities which are easier to reason about. Figure 5.3 shows an overview of concurrency within the Android application: there are five threads dedicated to various independent activities.

Parallelism has been employed in various low-level image processing operations by dividing the image into smaller pieces, which can be processed by different threads simultaneously — in parallel. As there is a need for multiple, relatively short operations, it would be rather inefficient to create new threads for every operation. The solution is to make use of a thread pool, i.e. a set of cached threads that are available for reuse. Throughout the C++ code, we extensively use OpenCV, which employs parallelism with a thread pool whenever applicable. Custom algorithms, such as differentiation (equations 3.2 and 3.3 combined), median of three images, and strip generation (see section 3.4) employ parallelism via the `cv::ParallelLoopBody` interface, allowing us to use the very same thread pool as OpenCV.

5.3. SIMD

An alternate way to accelerate code is to employ instruction-level parallelism, i.e. simultaneous execution using special processor instructions. These instructions allow to leverage single instruction, multiple data (SIMD) behavior, i.e. simultaneously peforming identical operations on distinct operands. Recently, SIMD has been widely used in a variety of applications due to its widespread availability on the x86 family of CPUs (instruction sets SSE, AVX), as well as ARM (NEON instruction set).

There are multiple ways to introduce SIMD into a code base; firstly, compilers might

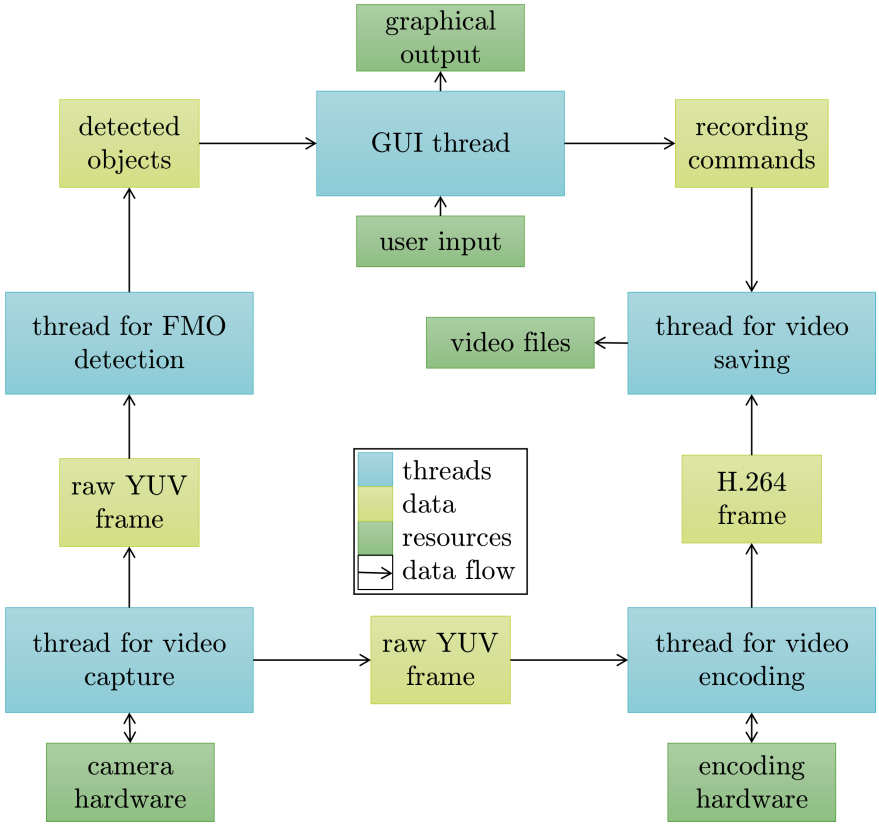


Figure 5.3. Concurrency in the Android app.

automatically use SIMD instructions in loops, via a process known as vectorization. To do that, the compiler must prove that a certain instruction set is available (e.g. SSE2 is always available on the AMD64 architecture). Secondly, one could a SIMD abstraction library that allows to write vectorized code for a variety of platforms without introducing code duplication, although typically introducing some overhead. The final option is to write multiple versions of performance-critical sections of code, one for each instruction set; this is what we do in the C++ library to accelerate certain compute-intensive loops. Specifically, SIMD is employed in differentiation (equations 3.2 and 3.3 combined) and median of three images.

5.4. Results

Figure 5.4 shows the benefits of multi-threading and explicit SIMD vectorization. Using both techniques simultaneously yields significant speedups in the benchmarks dedicated to the accelerated operations. In particular, the observed speedup of differentiation, median of three and strip generation is by a factor of 2, 11 and 3, respectively. Although we do not know the exact importance of these operations on mobile, based on the desktop profile (see figure 5.2) we expect a 40% performance gain just by optimizing these three operations.

fmo::Differentiator	J5 [ms]	G4 [ms]	Mi3 [ms]	average [ms]
NEON + threads	9.47	7	5.91	7.46
NEON only	9.81	7.44	7.82	8.36
threads only	10.2	12.49	7.14	9.94
no acceleration	22.42	18.87	7.98	16.42
fmo::median3	**J5 [ms]**	**G4 [ms]**	**Mi3 [ms]**	**average [ms]**
NEON + threads	2.59	1.65	1.25	1.83
NEON only	4.82	4.24	1.67	3.58
threads only	6.85	6.06	6.06	6.32
no acceleration	27.21	22.75	12.75	20.9
fmo::StripGen	**J5 [ms]**	**G4 [ms]**	**Mi3 [ms]**	**average [ms]**
threads only	5.08	3.33	2.02	3.48
no acceleration	17.35	10.38	7.93	11.89

Table 5.2. Acceleration using SIMD and multi-threading benchmarked on all tested devices (see table 5.1 for details). All values in the tables are medians of run times.

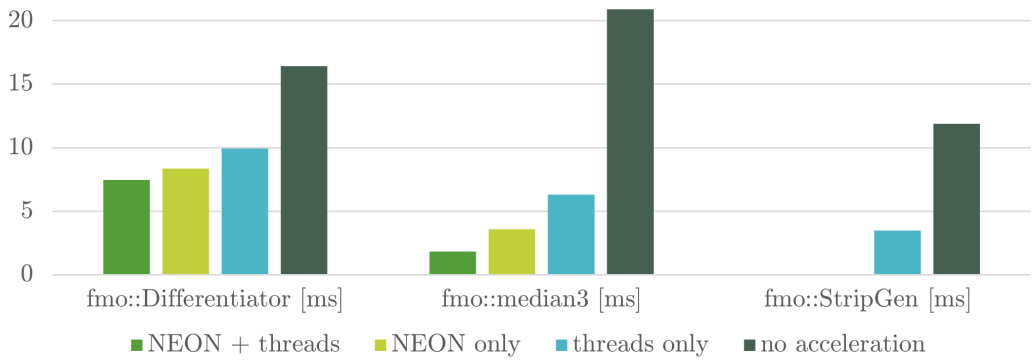


Figure 5.4. A visualization of table 5.2. The graph shows averages across all devices. Lower is better.

6. Detection quality

This chapter discusses the experiments conducted in order to assess the FMO detection quality of algorithm 1. The goal is twofold: firstly, to directly compare our approach to the algorithm described in [1], using the same data set and methodology. Secondly, to expand said data set to provide a more comprehensive baseline for future work.

6.1. Method

In order to be directly comparable to the results published in [1], our quality evaluation methodology mimics the one described therein. The goal of the experiment is to evaluate the quality of a FMO detection algorithm when confronted with real-world video sequences that are known to contain fast-moving objects. An algorithm that is being tested is provided frames from a video file as input. Metadata provided by the video is discarded — the algorithm receives no further information about the input frames, e.g., no frame timing, frame rate, exposure time. For each frame, the algorithm produces a set of objects that have been detected; for the purposes of this experiment, an object is a set of image pixels in an input frame.

A data set consists of multiple video sequences. To be able to evaluate the data set automatically, each sequence is accompanied by a ground truth file, which contains the expected output of the algorithm. The ground truth includes, for each frame, a set of objects that should be detected, where an object is again a set of image pixels. To store the data efficiently, the actual format contains run-length encoded binary images — see appendix B for details about the formats being used.

We investigate the quality of the algorithm in each individual frame by comparing the set of objects that the algorithm reports as detected, and the set of objects in the corresponding ground truth frame. Object comparison is based on treating the objects as sets of pixels, thus their intersection and union can be performed; the ratio of intersection magnitude over union magnitude, or intersection-over-union (IoU), serves as a measure of detection success. To inform about the quality of detection, events are reported in each frame, based on the following rules:

- If there are no ground truth objects and no objects reported by the algorithm, report a single *true negative* occurence.
- For each object provided by ground truth that has sufficient overlap with some object reported by the algorithm, report a single *true positive* occurence.
- For each object provided by ground truth that has insufficient overlap with all of the objects reported by the algorithm, report a single *false negative* occurence.
- For each object reported by the algorithm that has insufficient overlap with all of the objects provided by ground truth, report a single *false positive* occurence.

Overlap is deemed sufficient if IoU equals to or exceeds 50%. Throughout a video sequence, the number of occurences of each event — true positive and negative, false positive and negative — is summed to obtain the four quantities TP, TN, FP, and FN.

Figure 6.1. The FMO dataset.

For each sequence, these are combined to calculate *precision* π, *recall* ρ and F-score F_1 as follows:

$$\pi = \begin{cases} 1 & \text{if } FP = 0 \\ \frac{TP}{TP+FP} & \text{otherwise} \end{cases} \qquad \rho = \begin{cases} 1 & \text{if } FN = 0 \\ \frac{TP}{TP+FN} & \text{otherwise} \end{cases}$$

$$F_1 = \begin{cases} 0 & \text{if } \pi + \rho = 0 \\ \frac{2\pi\rho}{\pi+\rho} & \text{otherwise} \end{cases}$$

Precision, recall and F-score is averaged over all sequences. Average F-score serves as a measure of overall algorithm performance on the data set. Additionally, IoU for all ground truth objects in all frames may be arranged into a histogram, giving further insight into the quality of detection.

6.2. FMO data set

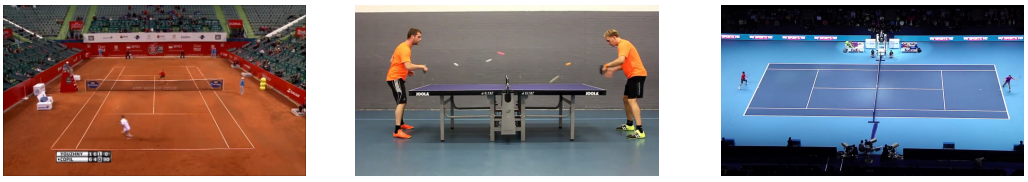
The FMO data set is the original set of videos with ground truth as described in [1]. No modifications have been done to the data, except that the ground truth has been converted from MATLAB data files to a text format (see appendix B for details). The new format is fast to load, provides more efficient storage, and also allows to have multiple, possibly overlapping objects in a single frame. Special care has been taken so that no information is lost during the conversion; in fact, the conversion is reversible as long as the new features of the format are not used. The sequences include:

- *volleyball* — A colorful ball is being thrown over a white and brown background. The object is large and there is a visible shadow cast by the object.

- *volleyball passing* — A colorful ball is moving over a white and green background. The lighting conditions make the ball difficult to see over certain parts of the background. The players cause a lot of extra motion.
- *darts* — Darts are being thrown over a green background. The object is not spherical, however it is clearly visible due to high contrast with the green wall.
- *darts window* — Darts are being thrown over a complex background. A lack of stabilization and the heterogeneous background make it challenging to observe the moving object in flight. Additionally, the object is not spherical.
- *softball* — A light ball is pitched and hit over a complex background. This is again a challenging sequence due to a lack of stabilization and the heterogeneous background.
- *archery* — An arrow pierces an apple in slow motion. The object is not spherical and appears to move slowly due to high frame rate. As the apple falls down from the head, it itself becomes a fast-moving object.
- *tennis serve side* — A yellow-green ball flies over a dark blue background. This is an example of a clearly visible, high-speed object.
- *tennis serve back* — A yellow-green ball flies over a dark blue and white background. The object moves away from the camera, shrinking in size.
- *tennis court* — A yellow-green ball flies over a blue background. The object is small and tends to move in a near-vertical direction. The players cause a lot of extra motion.
- *hockey* — Dark hockey pucks are being shot at the net over a white background. Lack of stabilization makes stationary objects appear moving. The player causes a lot of extra motion.
- *squash* — A white ball flies over a semi-complex background. The object is small and is often obstructed by various reflections or the players. The players cause a lot of extra motion.
- *frisbee* — A white disk is being thrown across a dark background. The object is very large and clearly visible, although the picture is not stabilized.
- *blue ball* — A blue ball is being thrown over a predominantly dark background. Useful for testing whether the algorithm uses color to distinguish the object from the background.
- *ping pong tampere* — An orange table tennis ball flies over a complex background. The object is not very visible due to its high speed.
- *ping pong side* — A white ball flies over a light green background. The object is hard to observe due to low contrast with the green wall. The players cause a lot of extra motion.
- *ping pong top* — A white ball flies over a dark blue background. The object is clearly visible, although certain shadows might appear like FMOs. The players cause a lot of extra motion.

FMO sequences	Frames	Max. TP	Max. TN	Stabilized
volleyball	50	11/13	39/37	yes
volleyball passing	66	66	0	yes
darts	75	36	39	yes
darts window	50	5	45	no
softball	96	26/28	70/68	no
archery	119	20/32	99/87	yes
tennis serve side	68	19/18	51/50	yes
tennis serve back	156	39/59	117/97	yes
tennis court	128/116	46/64	92/61	yes
hockey	350	60/61	294/293	no
squash	250	117/134	157/140	yes
frisbee	100	18/20	82/80	no
blue ball	53	21	32	yes
ping pong tampere	120	71/76	49/44	yes
ping pong side	445	374/436	74/29	yes
ping pong top	350	303/452	47/16	yes
New in FMOv2	**Frames**	**Max. TP**	**Max. TN**	**Stabilized**
tennis court 2	278	219	59	yes
more balls	300	1285	14	yes
tennis atp serves	655	463	192	yes

Table 6.1. Ground truth statistics of sequences in the FMO and FMOv2 data sets. Where the statistics have changed from FMO to FMOv2, the new value is after a slash.



tennis court 2 more balls tennis atp serves

Figure 6.2. New sequences in the FMOv2 dataset.

6.3. FMOv2 data set

The FMOv2 data set is the result of a collaborative effort between me and the authors of [1]. Going forward, this data set is to replace the FMO data set; all the sequences that are present in FMO are included in FMOv2 as well, and the data set is publicly available [18]. In addition to new videos, ground truth has been modified in most of the original sequences. The ground truth changes are of two kinds: firstly, there are fixes that pertain to missing or low-quality ground truth objects that have been discovered when investigating bad algorithm results. Secondly, FMOs that cease to be fast-moving are now still expected to be reported, up until the moment they disappear from view. This is to reward future algorithms that are able to continue tracking objects after slowdown. The new sequences include:

- *tennis court 2* — A yellow-green ball flies over an orange background. There are severe transcoding artifacts in the video: some frames are repeated, and some frames are blended in with others. Additionally, the motion of the object is often near-

	Proposed			Rozumnyi et al. [1]		
Sequence name	Precis.	Recall	F_1	Precis.	Recall	F_1
volleyball	62.50%	45.45%	52.63%	**100.0%**	**45.5%**	**62.5%**
volleyball passing	**80.39%**	**62.12%**	**70.09%**	21.8%	10.4%	14.1%
darts	50.00%	19.44%	28.00%	**100.0%**	**26.5%**	**41.7%**
darts window	100.00%	0.00%	0.00%	**25.0%**	**50.0%**	**33.3%**
softball	**76.92%**	**38.46%**	**51.28%**	66.7%	15.4%	25.0%
archery	0.00%	0.00%	0.00%	0.0%	0.0%	0.0%
tennis serve side	100.00%	52.63%	68.97%	100.0%	**58.8%**	**74.1%**
tennis serve back	**50.00%**	**17.95%**	**26.42%**	28.6%	5.9%	9.8%
tennis court	**100.00%**	0.00%	0.00%	0.0%	0.0%	0.0%
hockey	21.05%	13.33%	16.33%	**100.0%**	**16.1%**	**27.7%**
squash	0.00%	0.00%	0.00%	0.0%	0.0%	0.0%
frisbee	100.00%	72.22%	83.87%	100.0%	**100.0%**	**100.0%**
blue ball	95.24%	**95.24%**	**95.24%**	**100.0%**	52.4%	68.8%
ping pong tamper.	100.00%	**95.77%**	**97.84%**	100.0%	88.7%	94.0%
ping pong side	**75.88%**	**46.26%**	**57.48%**	12.1%	7.3%	9.1%
ping pong top	89.06%	75.25%	81.57%	**92.6%**	**87.8%**	**90.1%**
Average	**68.82%**	**39.63%**	**45.61%**	59.2%	35.5%	40.6%

Table 6.2. Algorithm 1 compared to the algorithm described in [1] on the FMO data set.

vertical. The players cause a lot of extra motion.

- *more balls* — Two players play table tennis with 5 balls at the same time. This is a long sequence, well-suited for testing the ability to detect multiple objects.

- *tennis atp serves* — Two players play tennis over a blue background. There is a minimal amount of extra motion, however the ball is very small and moves extremely fast.

6.4. Results

Figures 6.2 and 6.3 show the evaluation results for the FMO and the FMOv2 data set, respectively. The sequences that are shared by both data sets may show different results in each evaluation due to ground truth changes that have been incorporated into FMOv2. The configuration of the algorithm is identical in both cases, with all parameters set to their defaults (see appendix A for a list of parameters and their defaults). Since the FMO data set is the one described in [1] and our methodology is the same as described therein, it is possible to directly compare detection quality of algorithm 1 to the results published in [1]. Figure 6.3 compares the precision, recall and F_1-score achieved by both algorithms.

6.5. Analysis

Overall, algorithm 1 appears to perform slightly better than the algorithm by Rozumnyi et al. [1]. As shown in figure 6.3, there is improvement over [1] in terms of both average precision and average recall. However, there is still much room for improvement; we

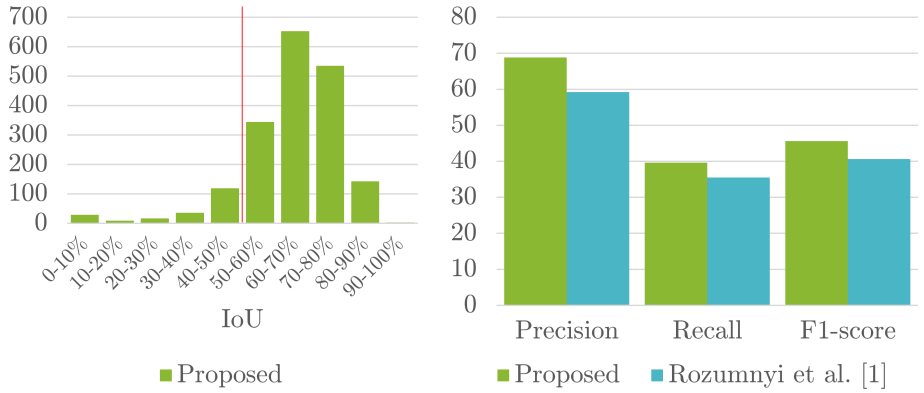


Figure 6.3. Left: histogram of non-zero intersection-over-union values of detections by algorithm 1, when evaluated on the FMOv2 data set. Right: a comparison of algorithm 1 and the algorithm by Rozumnyi et al. [1]. The graph shows averages of precision, recall and F_1-score over all sequences in the FMO data set.

Sequence name	TP	TN	FP	FN	Precision	Recall	F_1
volleyball	5	36	3	8	62.50%	38.46%	47.62%
volleyball passing	41	0	10	25	80.39%	62.12%	70.09%
darts	7	36	7	29	50.00%	19.44%	28.00%
darts window	0	45	0	5	100.00%	0.00%	0.00%
softball	10	68	3	18	76.92%	35.71%	48.78%
archery	0	87	6	32	0.00%	0.00%	0.00%
tennis serve side	10	50	0	8	100.00%	55.56%	71.43%
tennis serve back	11	97	3	48	78.57%	18.64%	30.14%
tennis court	0	61	0	64	100.00%	0.00%	0.00%
hockey	8	292	30	53	21.05%	13.11%	16.16%
squash	0	126	41	134	0.00%	0.00%	0.00%
frisbee	13	80	0	7	100.00%	65.00%	78.79%
blue ball	20	32	1	1	95.24%	95.24%	95.24%
ping pong tampere	68	44	0	8	100.00%	89.47%	94.44%
ping pong side	175	29	53	261	76.75%	40.14%	52.71%
ping pong top	229	13	27	223	89.45%	50.66%	64.69%
tennis court 2	9	58	22	210	29.03%	4.11%	7.20%
more balls	1061	11	66	224	94.14%	82.57%	87.98%
tennis atp serves	7	189	6	456	53.85%	1.51%	2.94%
Average					68.84%	35.36%	41.91%

Table 6.3. Detection quality evaluation of algorithm 1 on the FMOv2 data set.

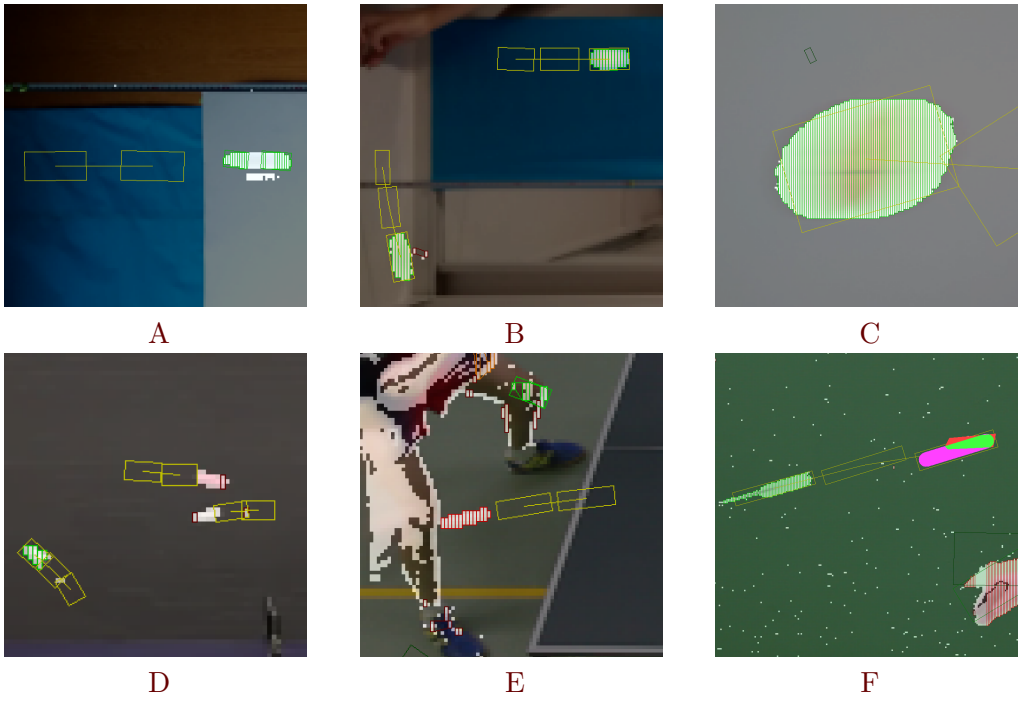


Figure 6.4. Failure cases described in section 6.5.

are aware of several causes of detection failure throughout the FMOv2 data set. Figure 6.4 shows various situations in which the algorithm behaves undesirably:

A Object is too similar to the background. If the color is not distinct enough, differentiation will fail to discern motion from background. In the image, we can see a white ball approach a white background. As soon as the ball enters the white area, detection fails.

B Shadow is considered an object. Algorithm 1 does not distinguish between an object and its shadow. In the image, the detection in the top right is an object, and the detection in the bottom left is its shadow.

C Stationary object appears moving. If the camera moves, there is no way to distinguish a moving object from a static one. The image shows an image of a basketball that has been detected due to being captured by a moving camera.

D Proximity to other FMOs. As two objects approach, they might both be filtered out based on the assumption that a FMO is isolated. In the image, we can see that the two detections on the right fail as soon as they get too close.

E Proximity to other motion. Once an object gets close to another source of motion, e.g. a player, it may be disregarded for the same reason as above. In the image, detection fails as the ball gets too close to the player.

F Unconventional shape of FMOs. Algorithm 1 assumes that the detected object is spherical; if not, the set of points being outputted may be somewhat unexpected. In the image, detection does not fail, but the reported shape (magenta and light green) does not resemble the object, which is a dart.

We believe that some of the above problems can be fixed; specifically, C can be resolved by applying image stabilization, and B could be solved using some advanced technique to distinguish shadows from regular objects. As for failure A, a modification of formula 3.2 could make the differentiation step more discriminative, however the algorithm will always fail if the object and the background are similar enough. Finally, there are also some good properties of algorithm 1 that should be pointed out: firstly, the additional two-frame latency has a positive effect on recall, as an object that is identified as fast-moving up to two frames later can still be reported. Secondly, adaptive thresholding based on the amount of noise (see section 3.4 for details) has been very successful at accommodating to a wide variety of scenes and lighting conditions.

7. Radius estimation

Algorithm 1 is able to estimate the radius of a detected object ω, as it calculates its estimated dimensions $d_1(\omega)$, $d_2(\omega)$ as part of the component description step (see section 3.6 for details). The inequality $d_1(\omega) \geq d_2(\omega)$ always holds, therefore $d_1(\omega)$ may be considered an estimation of the length of ω, while $d_2(\omega)$ is an estimation of its diameter (see figure 3.4). Consequently,

$$r(\omega) = \frac{1}{2} d_2(\omega) \tag{7.1}$$

estimates the radius of ω. Note that all of the mentioned quantities are measured in source image pixels, i.e. they are the dimensions of the object in the source image I_t, as opposed to the downscaled image $\hat{I}_t$. In the following sections, we describe three experiments pertaining to quality of radius estimation using equation 7.1.

7.1. Influence of downscaling

Due to the input subsampling step at the beginning of algorithm 1 (see section 3.2), the input image is downscaled before further processing. The aim is to increase detection efficiency and reduce noise in the processed image, at the cost of losing the ability to detect small objects, and introducing a degree of error. The following experiment aims to explore the effects of downscaling on the estimated object radius.

The radius has been measured throughout five sequences and in three scales — after one, two, or three applications of the subsampling operation. The sequences *ping pong tampere*, *ping pong side*, *ping pong top*, *blue ball*, and *frisbee* have been chosen for their good detection performance on at least two of the scales, and the fact that the FMOs in the sequences do not significantly change size. All of the used sequences are part of the FMO data set and are detailed in section 6.2. Parameters have been set to their defaults, except the ones that influence downscaling, radius correction and radius robustness — radius correction is introduced later in this section, and radius robustness is discussed in section 7.3. Table 7.1 summarizes the used parameters.

Only successful detections have been considered, having IoU with some ground truth object greater than 0.5 (see 6.1 for an explanation of IoU). For each sequence and scale, the mean μ and variance σ^2 of the reported radii have been calculated as if estimating parameters of a normal distribution. Table 7.2 shows the results; we observe that the estimated radius is sensitive to the number of downscaling steps, with the mean of the radius consistently increasing with each additional subsampling. Based on the results of the experiment, a correction constant has been devised to offset the radius mean accordingly. Subtracting the value 2^{d-1} px, where d is the number of subsampling operations, has the effect of equalizing the means at different resolutions, as demonstrated in figure 7.1.

7.2. Influence of video compression

In the context of video storage, lossy compression is a necessity. At the targeted full HD format with 3 bytes per pixel and 30 frames per second, a single second of video

number of subsampling ops.	1	2	3
`--p-max-image-height`	600	300	150
`--p-output-radius-corr`	0	0	0
`--p-output-no-robust-radius`	true	true	true

Table 7.1. Algorithm parameters used to perform the experiment described in section 7.1. Those not specified here are set to their defaults (see appendix A for defaults).

subsampling ops. →	1			2			3		
sequence ↓	#	$\mu(r)$	$\sigma^2(r)$	#	$\mu(r)$	$\sigma^2(r)$	#	$\mu(r)$	$\sigma^2(r)$
ping pong tampere	62	12.11	2.47	66	12.44	1.79	61	14.82	1.41
ping pong side	234	6.74	6.82	133	8.34	9.55	10	18.42	2.04
ping pong top	195	9.33	1.21	221	10.08	0.83	62	12.21	0.49
blue ball	19	14.11	2.47	20	15.14	2.89	19	17.88	2.70
frisbee	8	47.53	6.69	12	49.11	6.23	13	52.42	7.88

Table 7.2. The results of the experiment described in section 7.1. The number of detections, radius mean and variance are shown for three processing scales.

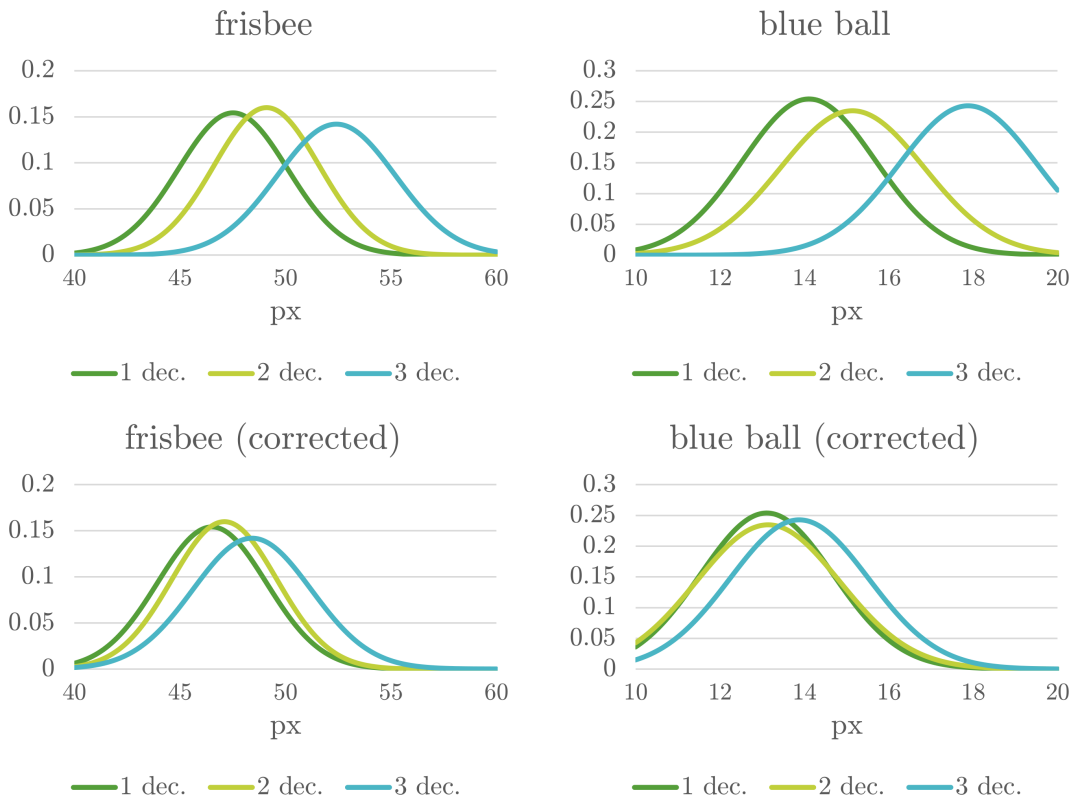


Figure 7.1. Top row: a visualization of selected sequences from table 7.2; with an increasing number of subsampling operations, radius mean increases. Bottom row: the effect of subtracting the correction constant 2^{d-1} px, where d is the number of subsampling operations.

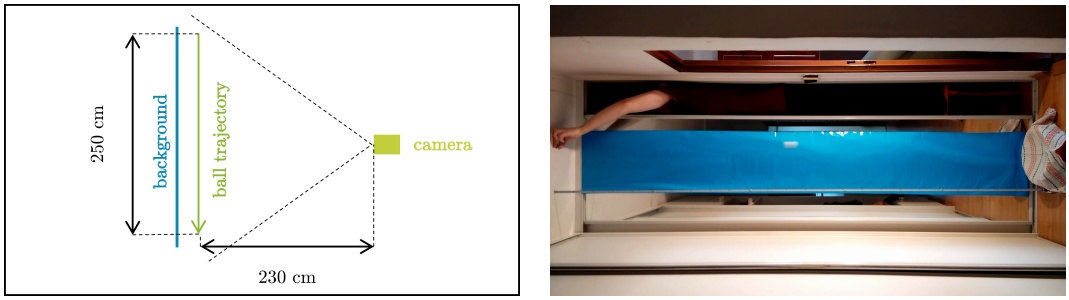


Figure 7.2. Capturing the video sequence for the experiment described in section 7.2. Left: used layout. Right: a typical frame from the captured sequence. The camera is rotated 90 degrees.

amounts to 179 MiB of data, which is beyond any practicality to store. Methods of lossless video compression do exist, the most popular being H.264 in lossless mode, however the compression ratio is still too low — a second of compressed video is 60 MiB in size. A certain loss of quality is thus unavoidable; luckily, the current industry-standard lossy encoders are able to achieve massive compression ratios with little visual degradation. The quality of the encoded video is determined by bit rate, i.e. the amount of data that the encoder is allowed to store per second of video.

While analyzing the results provided by algorithm 1 on various video sequences, it became apparent that the shape of FMOs was being affected by compression artifacts. To demonstrate this, a new sequence has been created using the built-in camera of Xiaomi Mi3, in which a ping pong ball moves over a blue background. In the video, the ball falls freely a total of 16 times, each time covering a distance of 250 cm from a fixed point to the ground; the camera is at a distance of 230 cm, and its optical center is perpendicular to the trajectory of the motion. See figure 7.2 for a drawing of the used layout. The goal of this setup was to create a repeatable experiment which could be captured multiple times, and which would involve a fast-moving object with a near-constant radius in the image plane.

The original captured footage has a constant bit rate of 15 Mb/s. To test algorithm performance with various levels of compression, the video has been re-encoded using the lossy H.264 encoder with 28 different bit rate settings, ranging from 100 kb/s to 10 Mb/s. Figure 7.3 shows visible effects of compression; the image becomes blurrier, and bright objects seem to glow. As a result, the boundary of a FMO becomes inexact, making it appear somewhat larger. Based on the value of difference threshold Δ (see equation 3.3), algorithm 1 might consider an object from a compressed video either smaller or larger than actual size.

All 28 re-encodings of the sequence have been used as inputs into the algorithm, with all parameters set to their defaults except that robust radius estimation was turned off (robust radius estimation is introduced in section 7.3). Remarkably, the ability to detect objects was not very much affected by compression, with 270-277 good detections reported at all bit rates except the ones below 300 kb/s, where about 15% of detections have been lost. Unfortunately, the same cannot be said for radius estimation, as demonstrated in figure 7.4 (pertinent to this discussion are only the graphs labeled "non-robust"). With an increased amount of compression (i.e., lower bit rate), variance of reported radii increases steadily, which is a sign of measurement unreliability. Moreover, the mean also increases, indicating that there is a systematic error introduced by the compression artifacts. Both mean and variance stabilize at a bit rate of about 6

bit rate: 100 kb/s
ratio: $5.8 \cdot 10^3$

bit rate: 1 Mb/s
ratio: $1.4 \cdot 10^3$

bit rate: 10 Mb/s
ratio: $1.5 \cdot 10^2$

Figure 7.3. Detail of a video frame containing a FMO. Compared are various fixed bit rates at which the video was encoded. Ratio is the effective compression ratio based on actual file sizes.

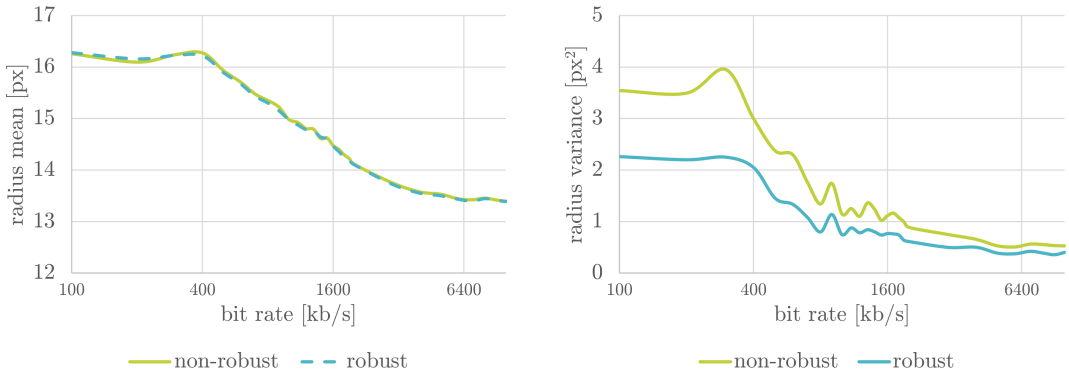


Figure 7.4. Results of the experiment described in section 7.2, comparing radius estimation quality for various levels of video compression. The graphs labeled "robust" show the effect of increasing robustness as described in section 7.3.

Mb/s.

We conclude that radius estimation should only be attempted on high-quality videos, with a sufficient bit rate. Fortunately, no such concern applies to using the Android application for live detection (see section 4.1.2), where the video stream is being processed raw, before any compression is applied (as shown in figure 5.3). A specific optimal bit rate or encoding settings cannot be recommended, as that largely depends on the amount of motion in the video.

7.3. Increasing robustness

In order to further increase robustness of the reported radii, we leverage the fact that every FMO detected by algorithm 1 has a predecessor, a successor, or both (see section 3.7 for details on how objects are matched). For an object ω detected at frame t, median can be used to extract radius from the estimated width of ω and its available neighbors:

$$\Lambda(\omega) = \{\omega\} \cup \{\omega' | (\omega, \omega') \in \mathcal{M}_t\} \cup \{\omega' | (\omega', \omega) \in \mathcal{M}_{t+1}\} \tag{7.2}$$

$$r(\omega) = \frac{1}{2}\text{median}_{\omega' \in \Lambda(\omega)}\{d_2(\omega')\} \tag{7.3}$$

Note that such a computation is only possible due to algorithm 1 having a non-zero output latency; the set $\mathcal{M}_{t+1}$ would normally not be available while processing frame t. Also note that in our implementation, median of two values is the mean.

The effect of using equation 7.3 to estimate radius $r(\omega)$ is shown in figure 7.4, which compares two evaluations of the experiment described in section 7.2; only the graphs labeled "robust" employ the equation. We observe that variance decreases significantly across all bit rates, thus making the radius measurement more reliable. This is due to the median operation automatically filtering outliers; as the amount of errors decreases for greater bit rates, the variance improvement becomes less significant. Still, for the maximum evaluated bit rate of 10 Mb/s, variance drops from 0.527 px^2 to 0.403 px^2. Unsurprisingly, the mean of reported radii is not affected by using equation 7.3 whatsoever.

We conclude that using equation 7.3 to estimate radius is strictly an improvement over equation 7.1. As a result, we use it by default, followed by the radius correction described in section 7.1.

8. Velocity estimation

In this chapter, we assess the viability of algorithm 1 for estimating velocities of detected objects. In general, measuring speed in images is problematic due to multiple reasons. Firstly, assuming that the camera conforms to the pinhole camera model, captured objects undergo a linear perspective projection. The projection transforms 3-dimensional objects onto a 2-dimensional image plane. The dimension reduction inherently causes a loss of information; in this case, we lose the notion of depth. Closer objects appear larger, and there is in fact no way to distinguish between an object getting closer and growing in scale. In terms of velocity, the problem is that farther objects appear to be moving slower than the closer ones.

Secondly, three-dimensional motion (and thus velocity) is not necessarily observable in the captured footage. In the worst case, the object moves directly towards or away from the camera projection center, its movement apparent only due to the change of scale. This again comes as a consequence of depth loss, which denies us information about distance from the camera. The motion is entirely visible only when the object is constrained to a plane parallel with the image plane. Next, any movement of the camera will inadvertently make the objects in the image appear moving. Therefore, camera stabilization is required in order to do any precise measurements. Finally, there is the matter of camera calibration; strictly speaking, calibration is a requirement for conforming to the pinhole camera model, and thus establishing the image plane and linear perspective projection. For any high-precision computer vision to take place, camera calibration is vital, but the calibration parameters are generally not available and hard to estimate unless there is a lot of footage captured by the same camera.

To recover object velocity, some additional prior knowledge is needed in order to overcome the problems described above. If we had an ideal, stabilized pinhole camera, and if we knew that all objects are constrained to planes parallel to the image plane, then the velocity measured in the image plane would determine real-world velocity up to scale. Algorithm 1 assumes all the conditions — calibration, stabilization, image-plane-parallel motion — to be properties of the input video. For each detected object ω in frame t that has a matched predecessor ω_1 in frame $t-1$, the algorithm reports the image-plane velocity $v'_t(\omega)$ in pixels per frame, simply by measuring the distance between the midpoints of the objects:

$$(\omega, \omega_1) \in \mathcal{M}_t : v'_t(\omega) = ||\vec{c}(\omega) - \vec{c}(\omega_1)||$$

See section 3.7 for details about how the set of matches $\mathcal{M}_t$ is formed. The last step is to convert the quantity $v'_t(\omega)$ to actual real-world velocity $v_t(\omega)$ in meters per second, and for that we need a factor that encompasses scale (i.e., pixel size) and frame duration. Let us make one of the following additional assumptions:

1. The FMOs are constrained to a single common plane $\bar{\pi}$ that is parallel to the image plane. Consequently, the real-world velocity of an object may be computed as follows:

$$v_t(\omega) = kfv'_t(\omega) \tag{8.1}$$

Figure 8.1. Left: a frame from the sequence *tennis atp serves*. Right: in an alternate footage of the same sporting event, ball speeds can be seen in the corner of the playing field.

where the constant k is the distance in meters that a point in $\bar{\pi}$ moves when it moves a distance of 1 pixel in the image plane, and the constant f is the frame rate in frames per second.

2. All FMOs have a single common radius. In that case, the real-world velocity may be obtained using the following formula:

$$v_t(\omega) = \frac{r'}{r(\omega)} f v'_t(\omega) \tag{8.2}$$

where the constant r' is the object radius in meters, $r(\omega)$ is the estimated radius of object ω in pixels, and f is the frame rate in frames per second. See chapter 7 for details on how object radius can be estimated using algorithm 1.

Note the similarity of formulas 8.1 and 8.2; indeed, $r'/r(\omega)$ is a per-object estimation of k. Regardless of whether assumption 1 or 2 is made, a single constant — k or r' — is all we need to recover real-world velocity. The Android application (see section 4.1.2) reflects this by allowing the user to enter a constant in the options menu.

8.1. Experiment

To assess the ability of algorithm 1 to estimate velocity, an experiment was conceived in which a reported tennis ball velocity is being compared to a known maximum ball speed during serve. A total of 10 serves has been gathered, all of which were captured from the side of the court by a spectator. Specifically, the serves in question are the last 10 serves of the final match of the 2010 ATP World Tour. Conveniently, due to the match being quite high-profile, additional footage is available that shows serve speeds on displays in the corners of the playing field, which we use as ground truth.

The footage from the side of the court has been used as input into algorithm 1, although some pre-processing was required first. As the source material is quite long, the footage has been cut to contain only the serves and the first returns. Since the video has been captured on a hand-held device and camera stability is a concern for velocity estimation, the sequence was stabilized and re-rendered using Blender. The resulting pre-processed sequence has been named *tennis atp serves*, annotated with ground truth object locations and added to the FMOv2 data set (see section 6.3). Algorithm 1 was not able to consistently detect the ball using default settings, due to the ball being too small in the image. To fix this, the experiment was run with a special set of settings that decrease the amount of downscaling and set the differentiation threshold to a constant value — see table 8.1 for a list of used parameter values.

parameter	value	rationale
`--p-max-image-height`	`600`	limits downscaling (see section 3.2)
`--p-diff-thresh`	`28`	sets threshold Δ to a constant (see equation 3.3)
`--p-diff-min-noise`	`0.0`	disables automatic adjustment of Δ
`--p-diff-max-noise`	`1.0`	disables automatic adjustment of Δ

Table 8.1. Algorithm parameters used to perform the experiment described in section 8.1. Those not specified here are set to their defaults (see appendix A for defaults).

no.	frames	detect.	lag	max $v_t(\omega)$	GT	extrapolated
1	67	46	2	99.0	108	105.6
2	70	51	1	100.7	101	103.8
3	62	28	4	93.7	104	106.5
4	75	29	5	90.1	113	101.7
5	82	42	6	83.8	104	91.9
6	30	11	2	117.3	127	127.4
7	34	7	6	97.0	112	116.1
8	78	43	4	105.4	125	123.2
9	67	38	4	83.7	99	88.3
10	90	31	1	107.0	108	110.2

Table 8.2. Velocity estimation results, comparing the maximum measured velocity and extrapolated velocity to ground truth. All speeds are in miles per hour. Lag is the number of frames between the serve and the first FMO detection.

Despite the increased resolution, the ball was still much too small for radius estimation to be reliable. Therefore, the formula 8.1 has been used, making the assumption that the effects of projective transformation are insignificant due to the great distance from the observer to the object. To find the constant k, the length of the court has been measured in image plane (1519 pixels) and divided by its known length according to tennis rules (78 ft), and the constant f is 29.97 frames for this particular sequence, conforming to the NTSC standard. Finally, the velocity has been converted to miles per hour in order to be comparable with ground truth velocities.

Table 8.2 and figure 8.2 compare the maximum reported velocities to ground truth. Ideally, the two speeds should match, however the algorithm does not achieve that in

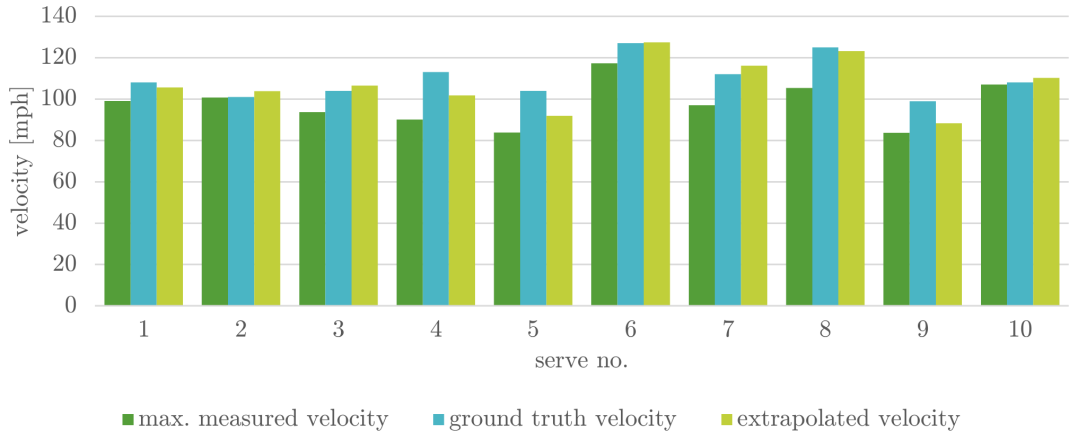


Figure 8.2. A visualization of table 8.2.

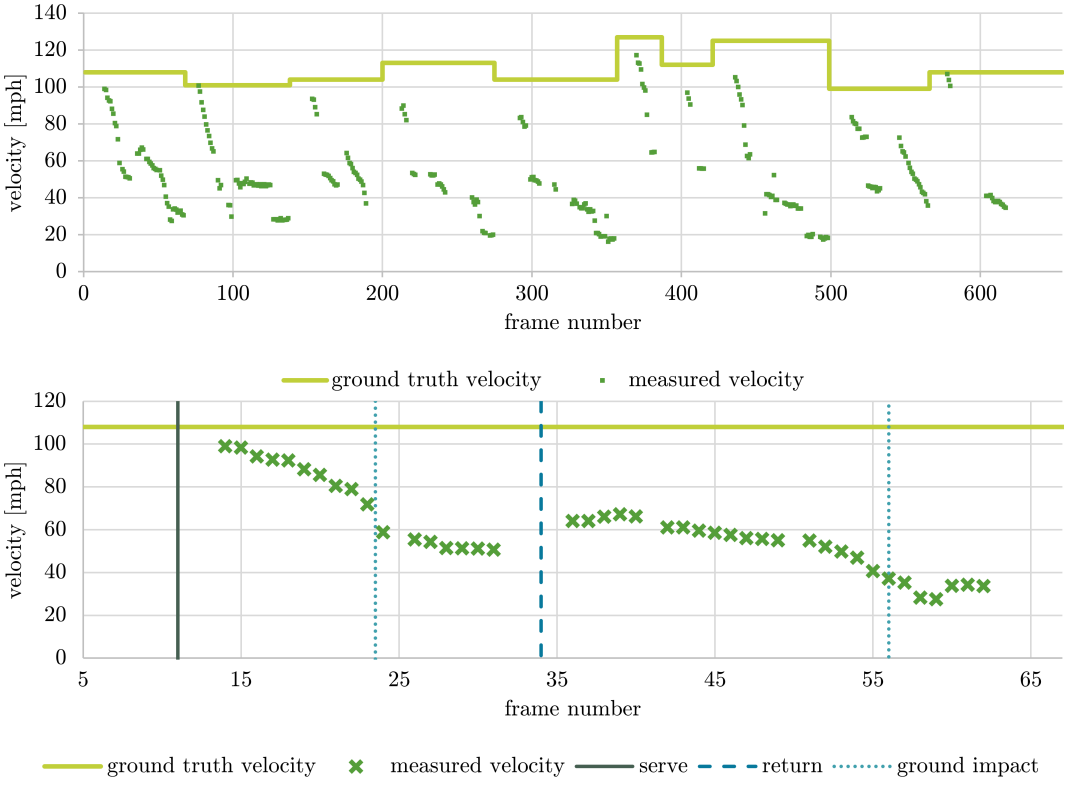


Figure 8.3. Top: speeds reported throughout the entire *tennis atp serves* sequence, comprised of 10 serves. Bottom: Detail of serve no. 1.

most cases. The reason is that the FMO is generally not detected immediately after the serve, which is when the ball has the most speed. For example, the measured velocity of serves no. 5 and 7 is so low because the ball is detected 6 frames after the initial impact. This is in contrast with serves 2 and 10, where the FMO is detected 1 frame after the impact, and thus the reported speed almost matches the ground truth. To estimate the actual maximum speed, we leverage the known time of the serve, producing an extrapolated value. Note that algorithm 1 does not have any information about when the ball was hit, and thus the extrapolation is not possible outside of an experiment. The extrapolation model is very simple, assuming that speed decreases linearly in the first 10 frames after a serve. As shown in table 8.2, linear extrapolation improves some of the velocity estimates, in particular serves no. 1, 3, 6, 7, 8, while the results for serves 2 and 10 remain excellent; the same cannot be said for serves 4, 5 and 9, though.

Our conclusion is that algorithm 1 cannot realiably measure serve speed, unless its recall is significantly improved. On the other hand, we have demonstrated that in 7 cases out of 10, the estimated velocity is close to ground truth when extrapolated to serve time, which indicates that the measurements are mostly precise. Moreover, once the FMO is detected, the velocity is immediately available, which allows one to measure the velocity throughout the entire duration of the point. Figure 8.3 shows that while the maximum speed in serve no. 1 is not captured, the speed during the rest of the play is there. This enables one to immediately draw conclusions such as: the ball hit the service box at 70-80 mph, slowing down to about 50 mph before the return; after the return, the ball flew at about 65 mph, etc. Thus it is demonstrated that using algorithm 1, real-time velocity estimation is feasible.

9. Conclusion

Fast-moving object (FMO) detection is a recently-introduced topic in the field of computer vision, applicable in the domain of sports and elsewhere. A new approach to highly-efficient FMO detection has been proposed, relying on the objects being unobstructed in three consecutive frames. The proposed method outperforms the best previously known approach, based on evaluation results using a common data set. To allow more comprehensive evaluation in the future, the data set has been extended to contain a wider variety of scenarios.

The above-mentioned detection algorithm has been implemented in C++, accomplishing the goal of portability, efficiency, and applicability to mobile devices. The core product is a cross-platform library, which is shared by an Android application and a set of desktop-bound tools. Performance has been thoroughly analyzed on the mobile platform, and it has been shown that real-time FMO detection is feasible on a range of typical consumer smartphone devices. Additional optimizations have been carried out based on data acquired by robust performance measurements and profiling.

It has been demonstrated that the detection algorithm can be extended to estimate additional properties of a spherical FMO, in particular its radius and velocity. Given a high-quality video without significant compression artifacts, radius estimation has been shown to have low variance. A modified algorithm has been able to continuously estimate ball velocities in a tennis match captured from the audience, although it was found unsuitable for recovering maximum speeds during serves.

Going forward, there is still much to be done on the subject of FMO. The detector itself could be significantly improved by implementing counter-measures against the documented failure cases. Furthermore, we have not tackled the subject of image stabilization, which could potentially improve detection quality by a wide margin. In terms of data, we feel that more contributions are still needed, especially outside the domain of sports. Moving on from efficient FMO detection, which this thesis proves to be attainable, the next logical step is to make FMO tracking as efficient, and then move on to more complex — or perhaps simpler, but in any case better — new approaches in this exciting new direction of research.

Appendices

A. List of parameters

This appendix lists all the parameters of the desktop application and their defaults.

A.1. Algorithm parameters

The following parameters pertain to algorithm 1.

- `--p-iou-thresh <float>` — The intersection-over-union value that needs to be exceeded for the detection to be deemed a true positive. Default: `0.5`
- `--p-max-image-height <int>` — $h_{\max}$ in section 3.2. Maximum image height for processing. The input image will be downscaled by a factor of 2 until its height is less or equal to the specified value. Default: `300`
- `--p-min-strip-height <int>` — ν_h in section 3.4. Strips that have less than this number of pixels in the downscaled image will be ignored. Default: `2`
- `--p-diff-thresh <uint8>` — Initial value for Δ in equation 3.3. Default: `24`
- `--p-diff-adjust-period <int>` — ν_N in section 3.4. The period of adjusting Δ. Default: `4`
- `--p-diff-min-noise <float>` — $\nu_{\min}$ in section 3.4. If the noise level is below this value, Δ is decreased to make differentiation more sensitive. Default: `0.0035`
- `--p-diff-max-noise <float>` — $\nu_{\max}$ in section 3.4. If the noise level is above this value, Δ is increased to make differentiation less sensitive. Default: `0.0047`
- `--p-max-gap-x <float>` — κ_x in section 3.5. Strips that are close to each other will be considered as part of the same connected component. This value is relative to image height. Default: `0.02`
- `--p-min-gap-y <float>` — κ_y in section 3.4. Strips will be ignored unless they are at a distance from other strips in the image. This value is relative to image height. Default: `0.046`
- `--p-min-strips-in-object <int>` — ζ in equation 3.6. Candidate objects that have less than this number of strips will be ignored. Default: `4`
- `--p-min-strip-area <float>` — ζ_A in equation 3.6. Candidate objects will be discarded if the area of strips contained within is small. This value is relative to the area of the objects convex hull. Default: `0.43`
- `--p-min-aspect <float>` — ζ_d in equation 3.6. Candidate objects will be discarded if their shape is too round. The aspect ratio must be greater than the specified value. Default: `1`
- `--p-min-dist-to-t-minus-2 <float>` — $\zeta_{\vec{c}}$ in equation 3.7. Candidate object will be discarded if its distance to an object found in frame t - 2 is below this value, weighted by the length of the older object. Default: `1.9`

- `--p-min-aspect-for-relevant-angle <float>` — τ_R in equation 3.11. The angle of the object is considered indistinguishable if its aspect ratio is below this value. Default: `1.62`
- `--p-match-aspect-max <float>` — τ_α in equation 3.12. Objects cannot be matched if the ratio of their aspect ratios exceeds this value. Default: `1.57`
- `--p-match-area-max <float>` — τ_β in equation 3.12. Objects cannot be matched if the ratio of their areas exceeds this value. Default: `2.15`
- `--p-match-distance-min <float>` — Objects cannot be matched if their distance (weighted by their average length) is below this value. Default: `0.55`
- `--p-match-distance-max <float>` — τ_γ in equation 3.12. Objects cannot be matched if their distance (weighted by their average length) exceeds this value. Default: `5`
- `--p-match-angle-max <float>` — τ_δ in equation 3.12. Objects cannot be matched if they do not lie on a line. The sine of the greater angle must not exceed this value. Default: `0.37`
- `--p-match-aspect-weight <float>` — α in equation 3.13. The weight of aspect ratio comparison when matching objects. Default: `1`
- `--p-match-area-weight <float>` — β in equation 3.13. The weight of area comparison when matching objects. Default: `1.35`
- `--p-match-distance-weight <float>` — γ in equation 3.13. The weight of mutual distance when matching objects. Default: `0.25`
- `--p-match-angle-weight <float>` — δ in equation 3.13. The weight of mutual angle when matching objects. Default: `5`
- `--p-select-max-distance <float>` — ϵ in equation 3.14. A triplet of objects will be discarded if the an object is too far from the expected location based on the linear motion assumption. This value is relative to object size. Default: `0.6`
- `--p-output-radius-corr <float>` — Enables radius correction described in section 7.1. Default: `1`
- `--p-output-radius-min <float>` — Reported radius cannot be lower than this value. Default: `2`
- `--p-output-raster-corr <float>` — Additional radius correction for rasterization (i.e. outputting points) only. Default: `1`
- `--p-output-no-robust-radius` — Disables robust radius estimation described in section 7.3.

A.2. General parameters

A.2.1. Algorithm selection

- `--algorithm <name>` — Specifies the name of the algorithm variant. Use `--list` to list available algorithm names.
- `--list` — Display available algorithm names. Use `--algorithm` to select an algorithm.

A.2.2. Mode selection

- `--headless` — Evaluation mode. Don't draw any GUI unless the playback is paused. Must not be used with –wait, –fast.
- `--demo` — Demonstration mode. This visualization method is preferred when `--camera` is used.
- `--debug` — Development mode. This visualization method is preferred when `--input` is used.

A.2.3. Input

- `--include <path>` — File with additional command-line arguments. The format is the same as when specifying parameters on the command line. Whitespace such as tabs and end-of-line is allowed.
- `--input <path>` — Path to an input video file. Can be used multiple times. Must not be used with `--camera`.
- `--gt <path>` — Text file containing ground truth data. Using this option enables quality evaluation. If used at all, this option must be used as many times as `--input`. Use `--eval-dir` to specify the directory for evaluation results.
- `--name <string>` — Name of the input file to be displayed in the evaluation report. If used at all, this option must be used as many times as `--input`.
- `--baseline <path>` — File with previously saved results (via `--eval-dir`) for comparison. When used, the playback will pause to demonstrate where the results differ. Must be used with `--gt`.
- `--camera <int>` — Input camera device ID. When this option is used, stream from the specified camera will be used as input. Using ID 0 selects the default camera, if available. Must not be used with `--input`, `--wait`, `--fast`, `--frame`, `--pause`.

A.2.4. Output

- `--record-dir <dir>` — Output directory to save video to. A new video file will be created, storing the unmodified input video. The name of the video file will be determined by system time. The directory must exist.
- `--eval-dir <dir>` — Directory to save evaluation report to. A single file text file will be created there with a unique name based on system time. Must be used with `--gt`.
- `--tex` — Format tables in the evaluation report so that they can be easily used in the TEX typesetting system. Must be used with `--eval-dir`.
- `--detect-dir <dir>` — Directory to save detection output to. A single XML file will be created there with a unique name based on system time.
- `--score-file <file>` — File to write a numeric evaluation score to.

A.2.5. Playback control

The following parameters pertain to development mode only.

- `--pause-fp` — Playback will pause whenever a detection is deemed false positive. Must be used with `--gt`.
- `--pause-fn` — Playback will pause whenever a detection is deemed a false negative. Must be used with `--gt`.
- `--pause-rg` — Playback will pause whenever a regression is detected, i.e. whenever a frame is evaluated as false and baseline is true. Must be used with `--baseline`.
- `--pause-im` — Playback will pause whenever an improvement is detected, i.e. whenever a frame is evaluated as true and baseline is false. Must be used with `--baseline`.
- `--paused` — Playback will be paused on the first frame. Shorthand for `--frame 1`. Must not be used with `--camera`.
- `--frame <int>` — Playback will be paused on the specified frame number. Must not be used with `--camera`.
- `--fast` — Sets the maximum playback speed. Shorthand for `--wait 0`. Must not be used with `--camera`, `--headless`.
- `--wait <int>` — Specifies the frame time in milliseconds, allowing for slow playback. Must not be used with `--camera`, `--headless`.

B. Text formats

Following is the specification of the text formats used as inputs or outputs of the desktop application. What these formats have in common is that the encoding is plain ASCII (Unicode is not allowed) and the endlines are Unix-style (line feed only).

B.1. Ground truth text format

This format is used when loading ground truth into the application (using the `--gt` command-line argument). The MATLAB code uses its own ground truth format in the form of binary `.mat` files; a MATLAB script is provided in companion material to convert files from binary to text and back.

The file begins with the space-separated integers `W`, `H`, `F`, `O`, `L` on a separate line. These denote: width, height, number of frames, frame offset for playback, and the number of non-empty frames. In most cases, `O` is zero.

Next are `L` lines with the following format: integer `I`, `1 <= I <= F`, specifies the frame number that is being described on this line. Frame numbers use MATLAB's one-based indexing, i.e. the first frame is frame number 1. Integer `N`, `1 <= N < W*H`, specifies the length of the run-length encoding of the frame.

The following `N` integers are lengths of runs of alternating color, starting with black. The image is considered to be a one-dimensional, contiguous array created by joining image rows together from top to bottom (this representation is commonplace in C and C++ graphics software). The length of the last run is omitted, and the importer needs to add the last run up to the number of pixels in the image.

Example file:

```
3 2 4 0 3
1 2 0 2
2 2 3 2
3 1 4
```

The file above encodes the following binary frames, where `.` is black and `X` is white:

```
1     2     3     4
XX.   ...   ...   ...
...   XX.   .XX   ...
```

B.2. Evaluation text format

This format is produced by the application using the `--eval-dir` command-line argument, and also read as input when via the `--baseline` argument.

Evaluation data starts with the string `/FMO/EVALUATION/V3/` on a separate line. Everything before this line is ignored. On the next line there is the integer `N`, specifying the number of sequences.

The following pattern of 6 lines is repeated `N` times. The first line contains a sequence name, the integer `F`, denoting the number of frames in the sequence, and the integer `O`, specifying the number of detected objects with non-zero IoU. The sequence name must be stripped of any directories or extensions (remove _gt, .mat, .txt, .avi, .mp4,

.mov, etc.) and spaces must be replaced with underscores. The next four lines contain information about FN, FP, TN, TP, respectively and strictly in this order. Each of these lines contains the name of the result (FN, FP, TN or TP) followed by `F` integers separated by spaces, specifying the number of the particular kind of result in each frame. The last line of the pattern contains the string `IOU`, and following are `O` integers in range 0 to 1000.

Example file:

```
/FMO/EVALUATION/V3/
2
example 5 4
FN 0 0 0 0 4
FP 0 0 2 0 0
TN 1 1 0 0 1
TP 0 0 0 1 0
IOU 512 799 14 501
example_2 2 1
FN 0 0
FP 0 0
TN 1 0
TP 0 1
IOU 698
```

B.3. Detection text format

This format is produced by the application using the `--detect-dir` command-line argument. It is a simple XML file containing all the information about the detected objects. As opposed to the evaluation text format, ground truth is not required to generate the file.

Example file:

```
<?xml version="1.0" ?>
<run>
    <date>2017-05-21 14:26:45 +0200</date>
    <sequence input="series2 compilation raw.avi">
        <frame num="15">
            <detection id="1">
                <center x="106" y="517"/>
                <direction x="0.996608" y="-0.0822904"/>
                <length unit="px">33.8015</length>
                <radius unit="px">13.0327</radius>
                <velocity unit="px/frame">32.0156</velocity>
                <points>104 505 105 505 ...</points>
            </detection>
        </frame>
        <frame num="16">
            <detection id="2">
                <predecessor>1</predecessor>
                ...
            </detection>
        </frame>
        ...
    </sequence>
</run>
```

Bibliography

[1] D. Rozumnyi, J. Kotera, F. Sroubek, L. Novotný, and J. Matas, "The world of fast moving objects," *CoRR*, vol. abs/1611.07889, 2016. 2, 4, 7, 28, 29, 31, 32, 33

[2] T. Vojir, J. Noskova, and J. Matas, "Robust scale-adaptive mean-shift for tracking," in *Scandinavian Conference on Image Analysis*, pp. 652–663, Springer, 2013. 2

[3] M. Danelljan, G. Häger, F. Khan, and M. Felsberg, "Accurate scale estimation for robust visual tracking," in *British Machine Vision Conference, Nottingham, September 1-5, 2014*, BMVA Press, 2014. 2

[4] J. Zhang, S. Ma, and S. Sclaroff, "MEEM: robust tracking via multiple experts using entropy minimization," in *European Conference on Computer Vision*, pp. 188–203, Springer, 2014. 2

[5] M. Danelljan, G. Hager, F. Shahbaz Khan, and M. Felsberg, "Learning spatially regularized correlation filters for visual tracking," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 4310–4318, 2015. 2

[6] S. Hare, S. Golodetz, A. Saffari, V. Vineet, M.-M. Cheng, S. L. Hicks, and P. H. Torr, "Struck: Structured output tracking with kernels," *IEEE transactions on pattern analysis and machine intelligence*, vol. 38, no. 10, pp. 2096–2109, 2016. 2

[7] A. W. Smeulders, D. M. Chu, R. Cucchiara, S. Calderara, A. Dehghan, and M. Shah, "Visual tracking: An experimental survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 36, no. 7, pp. 1442–1468, 2014. 2

[8] M. Kristan, J. Matas, A. Leonardis, M. Felsberg, L. Cehovin, G. Fernández, T. Vojir, G. Hager, G. Nebehay, and R. Pflugfelder, "The visual object tracking vot2015 challenge results," in *Proceedings of the IEEE international conference on computer vision workshops*, pp. 1–23, 2015. 2

[9] Y. Wu, J. Lim, and M.-H. Yang, "Online object tracking: A benchmark," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2411–2418, 2013. 2

[10] "Fast-moving object detection in C++." https://github.com/tufak/fmo-cpp. Accessed on April 22nd, 2017. 7, 17

[11] "Fast-moving object detection on Android." https://github.com/tufak/fmo-android. Accessed on April 26th, 2017. 17

[12] "About OpenCV library." http://opencv.org/about.html. Accessed on April 20th, 2017. 17

[13] "About CMake." https://cmake.org/overview/. Accessed on April 20th, 2017. 17

[14] "Chroma subsampling - Wikipedia." https://en.wikipedia.org/wiki/Chroma_subsampling. Accessed on April 20th, 2017. 18

[15] "Catch: A modern, C++-native, header-only, framework for unit-tests, TDD and BDD." https://github.com/philsquared/catch. Accessed on April 20th, 2017. 18

[16] "Android dashboards." https://developer.android.com/about/dashboards/index.html. Accessed on May 24th, 2017. 18

[17] M. Konrad, "Parallel computing for digital signal processing on mobile device GPUs," 2014. 23

[18] "FMO home page and data sets." http://cmp.felk.cvut.cz/fmo/. Accessed on April 26th, 2017. 31