Journal of Real-Time Image Processing manuscript No.
(will be inserted by the editor)

Christoph H. Lampert · Jan Peters

Real-Time Detection of Colored Objects In Multiple Camera Streams With Off-the-Shelf Hardware Components

Received: date / Revised: date

Abstract We describe RTblob, a high speed vision system that detects objects in cluttered scenes based on their color and shape at a speed of over 800 frames per second. Because the system is available as open-source software and relies only on off-the-shelf PC hardware components, it can provide the basis for multiple application scenarios. As an illustrative example, we show how RTblob can be used in a robotic table tennis scenario to estimate ball trajectories through 3D space simultaneously from four cameras images at a speed of 200 Hz.

1 Introduction

Many computer vision applications, *e.g.* in robotics or industrial inspection, make use of camera images as visual input. In particular, many interesting applications require a continuous visual input stream that has to be processed under real-time conditions. For example, vehicle driver assistance systems must observe street scenes [1, 2] or the driving person [3] in real-time. Similarly, gesture recognition systems or visual surveillance tools are most useful when they are able to process video material on-the-fly, such as [4, 5, 6]. Real-time visual processing is also required for continuous environmental mapping and localization for robots [7] or for wearable computers [8]. Certain robotics applications, such as visual servoing [9], or throwing and catching of balls [10] can even require it to process 100 images per second or more as they have to react to very fast changes in their environments.

In this work we study the field of real-time localization and tracking of objects in natural environments. We introduce a new system called *RTblob* that is based on four design criteria that we believe need to be fulfilled for any vision system that wants to find wide application in real-time and interactive scenarios. it has to be *simple*, *high performing*, *adaptable* and *affordable*.

Simplicity is relevant, because one often does not trust a system if one does not fully understand it. Such systems are useful in isolated or academic applications, but they are typically difficult to integrate into actual production systems. High performance, both in speed and in quality, is clearly necessary, otherwise one has no benefit from using such a system. Adaptivity is important, because every application scenario is different, and no black box setup will be able to fulfill everybody's needs simultaneously. Finally, it goes without saying that a system that is too expensive in either its software or its hardware components will not find wide application, either.

RTblob is designed to fulfill all of the four criteria simultaneously. It consists of a feed-forward processing pipeline for camera images that is illustrated in Figure 1. To achieve high throughput at low cost, the most computationally intensive tasks are executed on the GPU of a PC graphics card. In particular this is the central object detection module that localizes objects in images as the spatial maximum of a linear filtering operation [11]. The filter used can be changed with the same rate as the images are processed. This gives RTblob large flexibility what objects to detect: simple shapes such as circles of fixed size can be detected by a single filtering operation. For detecting not rotationally invariant objects in general positions a single filtering operation is not sufficient, because the filter response is typically rotation dependent. We can detect such objects by calling the filtering step multiple times with rotated version of the object filter and keeping the detection of maximal score. Objects that undergo significant size changes can be handled analogously by applying filters of different size, but we also discuss a more elegant solution at the end of the paper.

In the following, we will concentrate our description on the detection of uniformly colored balls. This is a particularly simple and robust task, since balls appear as circles in the image from every perspective viewpoint, and a uniform coloring allows the use of efficient color filtering

Christoph H. Lampert
Institute of Science and Technology Austria
Am Campus 1, 3400 Klosterneuburg, Austria
E-mail: chl@ist.ac.at

Jan Peters
Max-Planck Institute for Biological Cybernetics
Spemannstr. 38, 72076 Tübingen, Germany
E-mail: jan.peters@tuebingen.mpg.de

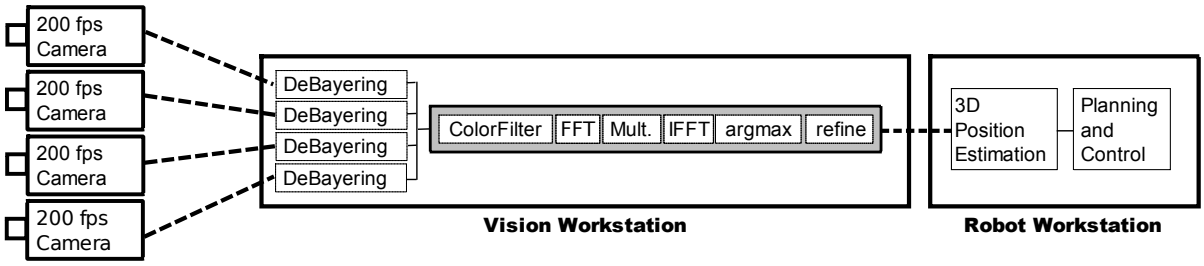

Fig. 1 Visual processing pipeline: The PC (central rectangle) receives image data through Ethernet connections (dashed lines) from the cameras. The image processing is performed in large parts on the PC's graphics card (gray rectangle), only the debayering step is performed in parallel threads on the CPU. The resulting object coordinates are passed on to a second PC (right rectangle) for further processing and robot control.

for preprocessing. As we will see from the experiments, RTblob can achieve over 800 frames per second throughput in this setup on ordinary PC hardware, and we use this to detect the ball in multiple camera streams in parallel. By triangulating the detections from overlapping camera view we obtain trajectories in three-dimensional space with high time resolution and low latency.

We believe that RTblob has the potential to become the seed for a standard platform for interactive applications as well as a valuable teaching tools for GPU accelerated high-speed vision tasks. To support this, the source code of RTblob is available under a free software license that permits modification as well as redistribution.

In the rest of this paper, we describe all relevant components of RTblob. We start by introducing related work, followed by an explanation of the design decisions that are necessary to make RTblob fast enough for performing real-time detection on off-the-shelf hardware. Subsequently, we explain the image processing steps that RTblob uses for object detection. To provide insight how RTblob works in a practical application, we demonstrate its performance on a task of ball tracking in 3D space, as part of a system in which a robot arm plays table tennis. At the end of the paper we give an overview of potential extensions and future work.

1.1 Related Work

Because of the importance of the application, many commercial as well as academic solutions for localizing and tracking objects in image streams have been developed. Commercially available solution achieve high performance and robustness by the use of dedicated hardware, *e.g.* active illumination by pulsed LEDs and IR reflective markers for the VICON systems[1]. Marker-less systems typically rely on embedded hardware, *e.g.* a digital signal processor, to do full image stereo reconstruction. The resulting depth field can be used to estimate the 3D positions of textures objects, as done *e.g.* in the *Organic Motion*[2] or *Tyzx*[3] solutions.

Academic vision systems are typically software-based and therefore more affordable. For example, a software system for tracking a marked object has been developed in [12] for the task of robotics tweezer manipulation. The system identifies a marker on the tip of the tweezer by thresholding an intensity image. By working in a region of interest only, this is possible at a speed of up to 500 Hz.

For detecting more general objects, even generalizing to object categories, sliding-window based object detectors have proved very successful. These first learn a classifier function from training examples, *e.g.* by boosting [13] or with a support vector machine [14]. They then evaluate this function over many candidate regions in every image, which is possible efficiently by adopting a cascaded approach. Thereby, they can detect the positions of objects as rectangular image regions. When even higher throughput of hundreds or thousands of frames per second is required, systems with additional hardware support has been devised, in form of FPGAs [15, 16, 17] or specially designed chips [18]. Such systems can process all pixel in an image in parallel, allowing them to estimate local motion with very high speed. The global object motion can then be inferred by integrating over motion paths. However, because of the high cost of custom hardware they have not found widespread use so far.

2 RTblob Design and Hardware

As explained in the introduction, we have designed RTblob with the goal of being simple, high performing, and adaptable. This leaves us with two software-based possibilities how we could perform object localization in image sequences: *per-frame detection* or *tracking*. In a per-frame detection setup, each image of the sequence is processed independently by applying a still image object detection algorithm. As the result, one obtains robust detection results, but this comes at a cost of relatively high

[1] `http://www.vicon.com`

[2] `http://www.organicmotion.com`

[3] `http://www.tyzx.com`

computational effort, because all image pixels need to be processed in every time step. Tracking aims at reducing the computational cost by searching for the object only in a region of interest, and this is updated from image to image based on an estimate of the object motion. For RTblob we choose a per-frame detection setup and we overcome the computational bottleneck by making use of the massively parallel architecture of the GPU in a modern PC graphic card. This way, we achieve high throughput while keeping the necessary computational resources to an acceptable level. The integration of a tracking module would still have advantages, *e.g.* for cameras with programmable region of interest, or to improve the detection quality by making the system adaptable to specific object instances. We leave this to future work.

Because we also want to keep RTblob affordable, we avoid "intelligent" but expensive components such as cameras with built-in computing capacities, and we rely only on off-the-shelf components. Consequently, any modern PC can serve as the hardware basis of RTblob. In our setup we use a Dell Precision T7400 workstation with two 3.4 GHz Intel dual core CPUs. As we will see later, this alone would not be sufficient for the necessary image processing tasks. Instead, we benefit from the vast increase in compute capacities of PC graphics cards over the last years, and from the development of new programming languages that support general purpose computation on these platforms. Specifically, RTblob relies on the CUDA framework for NVIDIA graphics cards [19] under GNU/Linux. The CUDA programming language integrates seamlessly into C/C++, thereby making it possible to write hybrid programs that run partly on the CPU and partly on the GPU. Additionally, NVIDIA provides an extensive SDK[4] under a royalty-free open source license that, in particular, allows the creation and distribution of derivative works. The NVIDIA GeForce GTX 280 graphics card, which we use as hardware platform, is available at a street price of less than $300.

Most high frame rate digital cameras transfer their image data in uncompressed form, and this makes fast data transfer between cameras and PC a crucial aspect of any high speed vision system. Until recently, only costly frame grabber card, *e.g.* with the CameraLink interface [20] were able to stream images at high enough frame rates. The recently introduced *GigE Vision* standard [21] provides a more affordable alternative, allowing for the first time to transfer several hundred full-sized cameras images (VGA or even larger) per second using only off-the-shelf networking hardware. In our setup, we installed an Intel PRO/1000 PT Quad Port gigabit Ethernet card ($\approx$ $350) to which we connect four Prosilica GE640C cameras with Pentax 16mm lenses.

We link the cameras with a simple hardware trigger cable that ensures that the camera exposures are synchronized, and we use alternating image buffers, such that the next exposure can already be started when the previous image is still transferred to the PC. At a per-unit cost of approximately $1500, the cameras are the most expensive components in our setup. However, cheaper solutions can easily be thought of, *e.g.* when a throughput of fewer than 200 fps is sufficient, or when only images smaller than 640×480 are required. In many such cases, consumer webcams with USB connectors will be sufficient. Alternatively, the described setup can also be extended to systems with faster or more cameras, if necessary by using CUDA's capability to distribute computation between multiple graphics card.

[4] `http://www.nvidia.com/object/cuda_home.html`

3 Object Detection

Even for the reduced class of objects that we chose as targets, many different detection algorithms have been developed, based, *e.g.*, on binary contours [22], background subtraction [23] or the Hough transform [24]. Following our objective of simplicity and high speed, we use a feed-forward architecture based on a linear shift invariant filter (LSI) [11]. This choice results in a pipelined approach (see Figure 1) in which we have divided the computation between CPU and GPU to achieve minimal latency and maximal throughput.

Algorithm 1 (Object Detection Pipeline)

1: Transfer Image Data: Camera to Main Memory
2: Apply Debayering
3: Transfer Image Data: Main Memory to GPU
4: Color Filtering
5: Filter Response Maximization:
6: Compute Fourier Transform
7: Multiply with Transformed Filter
8: Compute Inverse Fourier Transform
9: Compute argmax of Response
10: Refine to Sub-Pixel Accuracy
11: Transfer Object Coordinates: GPU to Main Memory
12: Triangulate 3D Object Position from 2D

The individual steps of the algorithm are listed in Algorithm 1. In the following, we will explain their implementation, concentrating on the steps that specific to our real-time system with GPU support, whereas for classical image processing techniques we will rather refer to existing textbook material.

1) Image Transfer from Cameras to CPU

Transferring the images of four 640×480 cameras at a frame rate of 200 Hz to the computer's main memory requires a network bandwidth of 234 MB/s plus minor communication overhead. By use of DMA-enabled network drivers and *jumbo* MTU packages, the chosen four port gigabit Ethernet card can sustain this datarate in a PCI Express slot with only low CPU load.

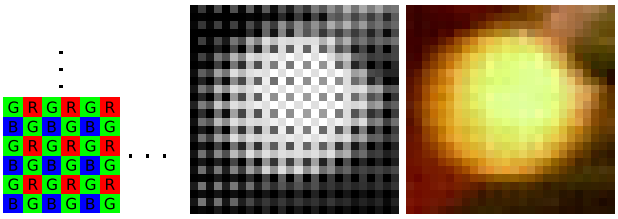

Fig. 2 Debayering: One-sensor color cameras have a built-in grid of color filters such that each pixel captures either red, green, or blue light (left). From the resulting intensity image (middle) one reconstructs a color image (right) by a procedure called *debayering*.

2) Debayering

In order to save manufacturing cost and transfer bandwidth, affordable digital cameras for machine vision typically contain only one CCD or CMOS sensor chip, and can therefore capture only a single-channel intensity signal. To nevertheless obtain color images that have three channels (red, green and blue) from this setup, small color filters are attached to each sensor cell, such that each pixel measures only either the red, green, or blue intensity of incoming light. The results is a single channel image representation with reduced color resolution, called *Bayer pattern*. The process of extrapolating full RGB images from the Bayer patterned images is called *demosaicing* or *debayering* [25], see Figure 2.

Bayer patterns come in different spatial arrangements and with filters of different spectral responses. Therefore, camera manufacturers usually provide debayering routines as part of their camera SDKs. In our case, we rely on the default routines provided by Prosilica. While in principle we could also perform the debayering on the GPU, this would require different CUDA routines depending on the Bayer pattern layout of the cameras' sensor chip. In addition, the routines provided by the camera manufacturer typically produce higher quality color images, because the spectral response curves of the sensor chips are taken into account. Performing debayering on the CPU is a computationally expensive task, but it can be parallelized, such that the four camera streams in our setup are processed by four independent CPU threads. Note that another positive side effect of CPU-bayed debayering is that the total amount of computation is split between GPU and CPU, which leads to higher overall throughput than for a system where all computation is performed either only on the CPU, or only on the GPU.

3) Image Transfer: CPU to GPU

Debayering alone already causes a CPU load of approximately 50% on each of the 3.4 GHz CPU cores of our system. Therefore, we perform the remaining image processing steps on the graphics card. Copying the image data from the PCs main memory to the graphics card's RAM requires a continuous memory transfer bandwidth of 703 MB/s, because the amount of image data has been tripled by the conversion from a Bayer pattern to a color image. Modern graphics cards with *PCI Express* interface can easily sustain this rate, *e.g.*, our local setup achieves a throughput of over 5000 MB/s using so-called *pinned memory* transfer routines of the CUDA language.

4) Color Filtering

The detection of objects with a characteristic color has the advantage that we can use fast, color-based preprocessing operations. In RTblob we can measure the *interestingness* of each image pixel by comparing its *RGB* value with the reference RGB tuple $(R_{ref}, G_{ref}, B_{ref})$. The result is a single-channel interest image $I[u,v]$, which in our case is calculated as

$$\begin{aligned} I[u,v] = 1 - \Big(& (g_R R^\gamma[u,v] - R_{ref})^{1/\gamma} \\ & + (g_G G^\gamma[u,v] - G_{ref})^{1/\gamma} \\ & + (g_B B^\gamma[u,v] - B_{ref})^{1/\gamma} \Big) / 3, \end{aligned} \tag{1}$$

where $R[u,v]$, $G[u,v]$ and $B[u,v]$) are the color channels of the image to be processed in floating point representation in the range $[0,1]$. $(g_R, g_G, g_B) = (1., 1.15, 1.3)$ are camera dependent gain factors and $\gamma = 0.5$ is a gamma-correction factor.

Color filtering is a step that operates independently on each image pixel. This makes it very easy to parallelize on modern GPUs that can manage hundreds of thousands of threads with virtually no overhead: we simply start one thread for each pixel. Note that, because no quantities have to be precomputed for Equation (1), we have the choice to change the reference color instantaneously, and, *e.g.*, search for a differently colored object in each of the camera streams.

5) –9) Object Detection

The step of actually detecting the most likely target object in an interest image is a two-step procedure: we apply a linear shift invariant filter, and we identify the position of maximal filter response. Figure 3 shows examples of the different processing stages for successful and unsuccessful object detections.

To formalize this, let F be a $w \times h$ sized filter mask that is constructed to have a high response at the object center. Examples of such filters will be discussed in Section 4.

The task of object detection can then be written as finding the pixel position of maximal filter response in the image, *i.e.*,

$$[\hat{u}, \hat{v}] = \mathrm{argmax}_{u,v} \;\; F_{u,v} * I \tag{2}$$

where $*$ is the 2D convolution operation, and $F_{u,v}$ denotes the filter mask F shifted to the position $[u,v]$ and

padded with zeros to be of the same size as I. While a naive implementation of the convolution requires four nested loops and has $\mathcal{O}(WHwh)$ runtime complexity, a more efficient way of evaluation is available by use of the *convolution theorem* [26]:

$$= \text{argmax}_{u,v} \mathcal{F}^{-1} \left(\mathcal{F}(F) \cdot \mathcal{F}(I) \right) [u, v] \quad (3)$$

where $\mathcal{F}$ denotes the Fourier transform and $\mathcal{F}^{-1}$ the inverse Fourier transform. As long as the filter does not change between frames, $\mathcal{F}(F)$ remains constant and can therefore be precomputed. Consequently, we can compute Equation (3) by only one forward and one backwards FFT operation, with a per-pixel multiplication in between, and a final pass through all pixels to identify the maximum. This results in a computational complexity of $\mathcal{O}(WH \log H + HW \log W)$, independent of the image contents.

Note that the detection equation (2) is not invariant against rotation or scale changes. If such properties are required, one needs to perform multiple searches, typically with rotated or scaled versions of the same filters, and keep the best matching detecting. This is not required in our setup of colored ball, because balls themselves are rotationally invariant objects, and a single linear filter can detect them even under certain changes of scale.

It is possible to modify Equation (2) to a template based maximum-likelihood prediction, such as [27]: given a $w \times h$ sized template T of the expected object appearance and assuming Gaussian image noise, the maximal likelihood position $[\hat{u}, \hat{v}]$ of best match of T within I is given by

$$[\hat{u}, \hat{v}] = \text{argmin}_{u,v} \left\| T - \Omega^{u,v}(I) \right\|^2 \quad (4)$$

where $\Omega^{u,v}(I)$ denotes the $w \times h$ sized rectangular region in I around $[u, v]$. We can decompose this using

$$\left\| T - \Omega^{u,v}(I) \right\|^2 = \|T\|^2 + \left\| \Omega^{u,v}(I) \right\|^2 - 2\tilde{T}_{u,v} * I, \quad (5)$$

where $\tilde{T}$ denotes the *mirrored* version of T, *i.e.* the template in which left and right as well as top and bottom directions are swapped. Because the first term does not depend on $[u, v]$, we can disregard it for the minimization. The second term can be precomputed with constant time effort per pixel using *e.g. integral images* [28]. Consequently, the position of best template match can be calculated using a variant of Equation (3) using $\tilde{T}$ as the filter mask.

Clearly, computing several hundred Fourier transforms and inverse Fourier transforms of 640×480 images per second is computationally demanding. However, FFTs are also operations that can be parallelized well on the GPU. Because these implementations are most efficient for image sizes that are powers of 2, we perform the FFT on a version of the image that has been rescaled to the size of 512×512. All modern graphics cards contain hardware units for texture rescaling, which enables them to change the image resolution basically free of computational overhead. Subsequently, we multiply the resulting Fourier spectrum with the Fourier transformed filter mask (also sized 512×512). As a per-pixel operation this again is possible very efficiently by starting as many GPU treads as there are pixels. After the subsequent inverse FFT step, we need to identify the position of maximal filter response in the image. While on a sequential processor, no better solution exists than a linear scan through the data, the highly parallel GPU architecture can solve the task in time that is logarithmic in the number of pixels [29]: starting with the full image, each step runs multiple threads, each of which compares two positions of the output of the previous step and stores the position of larger response value. After $\lceil \log n \rceil$ iterations, the maximal value and its position have been identified. To achieve maximal efficiency of such an *argmax* routine knowledge of the internal GPU hardware architecture is required in order to make optimal use of cache memory and to avoid read/write conflicts. Luckily, finding the maximal entry of a matrix is a common enough that adjustable templates in the CUDA language are already available as open source software [30]. For RTblob, we adapted these to perform the *argmax* search instead of the *max*. In total, the necessary steps for detecting the object in a 640×480 images takes only little more than 1 ms on an NVIDIA GeForce GTX 280 graphics card. Consequently, we achieve a throughput of over 800 frames per second, see Section 6.

10) Sub-Pixel Refinement

Computing the object position by a filter or template match results in integer valued object coordinates. We can improve this estimate by refining the solution found to sub-pixel accuracy. Using finite differences to the neighboring positions, we expand the quality surface of Equation (3) or (4) into a second order Taylor polynomial around $[\hat{u}, \hat{v}]$. A better estimate of the object position is then given by the coordinates of the extremal point of the resulting quadratic surface, and we can calculate these analytically, see [31].

11) Transfer Object Coordinates: GPU to CPU

In contrast to step *3)* where we copied the image data from the CPU to the GPU, the task of transferring the detected 2D object position back to the CPU's main memory is of negligible effort, because it consists of only a few bytes of data. In our table tennis setup, we subsequently send the data via a real-time network link to a different workstation that is responsible for robot planning and control. In general, such a split between two PCs is not necessary, but we found that in our setup we had to avoid interrupt conflicts between the CUDA-enabled graphics driver and the CAN-bus interface for

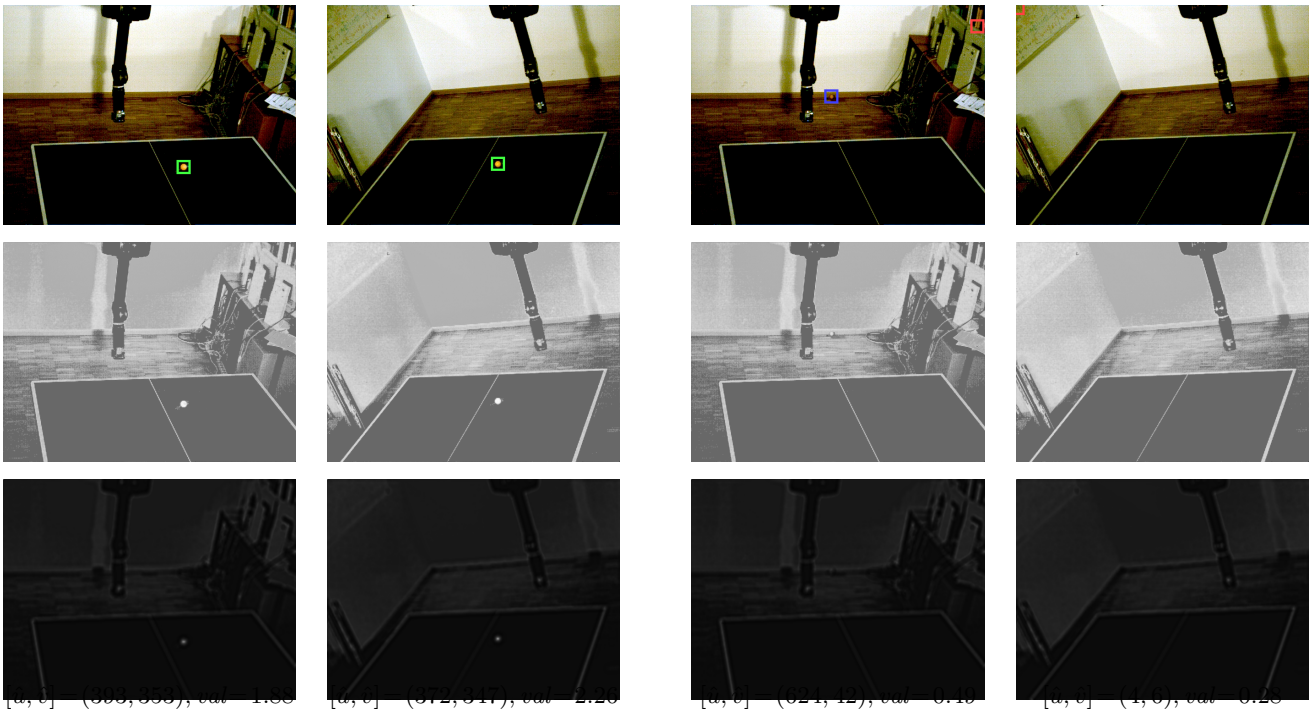

Fig. 3 Example steps of the processing pipeline: camera images after *debayering* (first row), *color filtering* (second row) and *shift invariant filtering* (third row). The last row contains the positions and values of the maximal filter score (Equations (2)), with $(0, 0)$ being the top left corner, which are also visualized as boxes in the first row. The left two images show a stereo pair where in both images the ball position is identified correctly and with high confidence (green squares). In third column, a ball is also present in the image, but it is small and dark in the background (blue square). The *argmax* prediction results in a false positive detection (red square). The fourth column image does not contain a ball, so prediction with the maximal filter position (red square) also leads to a false positive. Note that the maximal filter value in the incorrect examples is lower than for the correct score. By correctly adjusting a threshold most misdetections will be avoided. (the images were gamma corrected and contrast enhanced for better visibility).

robot control, both of which make use of real-time privileges for hardware access.

12) Triangulation

After having the 2D object positions from all cameras available, we can recover the object's position in 3D world coordinates by triangulation. For each camera pair (i, j) we use the cameras' *projection matrices* (P_i, P_j) to recover the *lines of sight* passing from the cameras' optical centers through the image planes at the detected position $([u_i, v_i], [u_j, v_j])$ using the projection equations

$$P_i \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = \lambda_i \begin{pmatrix} u_i \\ v_i \\ 1 \end{pmatrix} \quad \text{and} \quad P_j \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = \lambda_j \begin{pmatrix} u_j \\ v_j \\ 1 \end{pmatrix}. \quad (6)$$

where λ_i and λ_j are unknown proportionality factors. Ideally, the lines will intersect at the object's location $(X, Y, Z) \in \mathbb{R}^3$. Because in practice numerical and estimation errors prevent this from happening exactly, we estimate the object position as the point of closest distance between the lines in 3D space, see [32]. This triangulation step is of negligible computational cost, so we do not require GPU support for it. Consequently, it can be performed on the fly either on the vision workstation or the robot workstation.

If the object is in the field of view of more than just one camera pair, we can obtain multiple estimates of (X, Y, Z). To increase the solution stability, we use the position for which the 2D object detectors had higher confidence values. Alternatively, one could increase the solution accuracy by averaging over all detected position. We did not follow this approach, as the region where more than two camera's field of view overlap is small in our setup, and we would like to avoid the risk of introducing outliers by averaging over misdetections.

4 Parameter and Filter Selection

By choosing a linear detection model we made sure that the number of free parameters remains small. Apart from hardware dependent quantities, such as the camera projection matrices, camera gain and gamma factor, only two choices need to be made: the reference color (R_{ref}, G_{ref}, B_{ref}), and the filter mask F. Both have an important influence on the quality of the detection process.

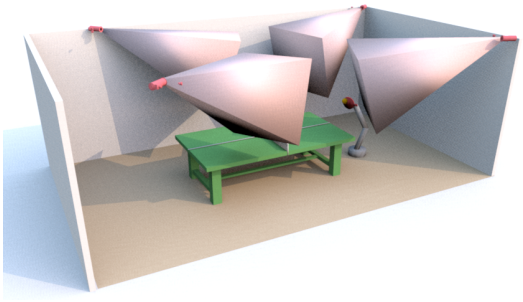

Fig. 4 Schematic visualization of the camera setup used for robot table tennis: the left camera pair observes the robot and part of the table, the right camera pair observes the rest of the table and the opponent. A central area of overlap allows easier camera calibration.

The reference color can typically be chosen as the dominant color of the object itself, *e.g.* a bright orange for the ball in the example of Figure 2. However, the tracking stability can sometimes be increased by adjusting the reference color in order to avoid high responses on background objects. Such adjustment are typically easiest to do in an interactive way, because the interest image of Equation (1) can be recomputed and visualized on-the-fly.

For setting the filter mask different approaches are possible. A classical setup for finding balls in images consists of applying a fixed band-pass filter, *e.g.*, a Laplacian of Gaussian (LoG) or difference of Gaussian (DoG) filter. Both have been found to be reliable detectors for circular "blobs" of fixed size [33, 34]. Alternatively, matched filters [35] or adaptive correlators [36, 37] can be used, which have the advantage that they can also detect more general patterns. Finally, when labeled training data is available, discriminative machine learning techniques can be used to compute an optimal filter mask [38]. In our setup, we have found a hybrid of manually designed and learned filters to yield good results: we first learned a filter in an active learning setup [39]. Subsequently we manually constructed a weight vector based on the characteristics that the learned filter has identified. It consists of a central ellipse of positive value, surrounded by a larger elliptic annulus of negative value, such that overall the filter value sum up to zero. The decision to use ellipses instead of circles comes from the analysis of the learned filter weights: The learning system had identified that the object appearance is not fully rotationally symmetric, see Figure 2, possibly due to the non-isotropic illumination in our lab environment.

5 Camera Setup

The computing power of our GPU based object tracker allows for the processing of over 800 2D image per second, which we currently utilize to process four 200 Hz image streams in an interleaved setup. We organize the four cameras as two camera pairs, where each pair observes approximate half of the 3D space that has to be monitored. A small central area of overlap allows the calibration of one camera pair using estimates of object positions obtained from the other pair. Figure 4 illustrates the setup chosen: the four cameras are installed at the corners of a $3.2\,m$ times $5\,m$ rectangle in a height of $2.75\,m$. Together, they observe a non-rectangular working space that is approximately $2.7m$ long and $1.4m$ wide at the height of the table ($0.76m$), and that reaches a height of $1.3m$ above the ground at the position of the net.

Alternatively, one could make all cameras observe the same working volume, which provides us with more independent measurements of the objects position, potentially increasing the system's overall accuracy. It would also be possible to utilize the higher GPU processing capacity with only a single camera pair, *e.g.* by filtering an image multiple times to detect objects that are not rotationally invariant, to track multiple objects of different color. The latter, for example, allows the implementation of a simple marker-based tracking for applications such as pole balancing that require not only the position but also the orientation of an articulated object.

5.1 Camera Calibration

In the camera setup we chose there is only a small amount of overlap between of the field of views of the camera pairs, because we aim for a large overall volume to be monitored. A disadvantage of this setup is that the robot arm itself is visible only in one of the pairs and a direct, ground truth based calibration of the other cameras is not possible. We overcome this problem by a two-stage calibration procedure: first, we calibrate the internal and external parameters of those cameras that monitor the robot's working space. This is achieved by mounting a calibration object, in our case just a table tennis ball, to the robots end-effector, and moving the robot along prespecified trajectories. With object positions $[u, v]$ as they are detected in the images and the object coordinates (X, Y, Z) provided by the robot, the only unknowns in the projection equations (6) are the camera matrices P_i. Because we cannot be sure that the measurements are error free, we use least-median-of-squares (LMedS) estimation to determine P_i [32]. This technique is slower (it takes seconds to minutes) than a simpler least-square estimate (which takes only milliseconds), but it often superior in terms of the robustness of results achieved, because it allows up to 50% of outliers to be present.

For the remaining cameras, which do not have the robot in their field of view, we perform a similar, but indirect, procedure. We move a calibration object manually through the area of overlap between the field of view the already calibrated cameras and the so far non-calibrated ones, and we record the detected positions in all camera

images. We do not need ground-truth world coordinates for these trajectories, because we can use the coordinates obtained by triangulation from the calibrated camera pair for world coordinates. Based on these, we solve Equation (6) for the remaining cameras using LMedS.

A subsequent parameter refinement would be possible using, *e.g.*, bundle-adjustment [40], but in our tests this did not lead to improved overall accuracy.

Of course, the RTblob setup is not specific to robot based camera calibration. Any classical self-calibration technique could be applied as alternative to the described setup, as long as it provides the camera matrices P_i as output.

6 Experiments

In the following, we show how RTblob performs in realistic setups. In particular, we can use ground truth data of a robot trajectory to obtain quantitative results on throughput, latency and accuracy in a realistic tracking task.

6.1 Throughput

We first report the performance of the RTblob system in terms of processing speed. For this, we run RTblob on a system with four streaming cameras and we change the number of image streams that RTblob processes. Table 1 summarizes the resulting CPU load and detection frame rates. The main result is that using only off-the-shelf hardware, RTblob can perform close to 830 detection operations per second when processing a single 200 Hz image stream, while causing a CPU utilization of 20% (user load). An additional 21% CPU utilization occurs due to operating system activity (system load).

The detection frame rate can be higher than the camera's fixed capture speed here, because we do not enforce the GPU's image processing loop to be synchronized with the image capture process. This allows us, *e.g.*, to filter the same camera image multiple times, either to detect multiple objects of different colors in the same camera stream, or to search for not rotationally invariant objects using differently rotated filter masks. Alternatively, we can interleave the image streams from multiple cameras: as the table shows, the additional debayering required makes the CPU utilization increase up to 46% for four cameras stream. As the total GPU throughput remains roughly constant, the per-stream detection speed is linearly lower when processing multiple stream than when processing a single one. Note that during the measurement for Table 1 all four cameras are running, we only change if their image contents is processed or not. Consequently the system load, which mainly consists of network transfer overhead, is constant.

# of cameras	CPU load (user/system)	detection speed
1	$\approx 20\%$ / $\approx 21\%$	829 ± 1 Hz
2	$\approx 31\%$ / $\approx 21\%$	419 ± 1 Hz
3	$\approx 38\%$ / $\approx 21\%$	282 ± 1 Hz
4	$\approx 46\%$ / $\approx 22\%$	207 ± 2 Hz

Table 1 Processing speed when a varying number of cameras streams are processed. CPU utilization caused by RTblob itself (user load) grows roughly linearly with the number of camera streams to be processes. CPU utilization due to operating system calls (system load) remains stable.

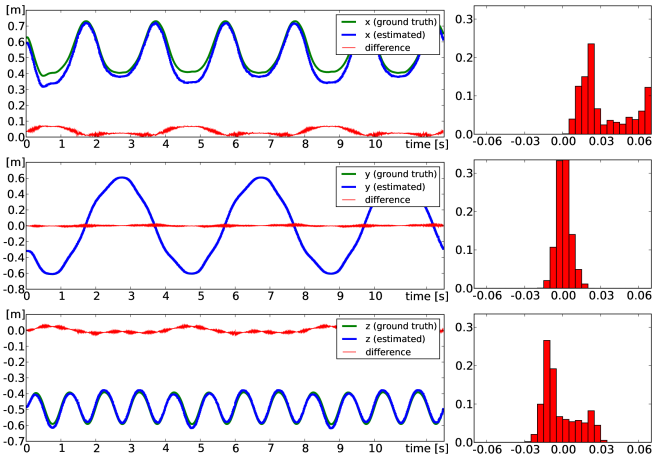

Fig. 5 Illustration of tracking accuracy. Left: detected 3D trajectory (blue) compared to ground truth as measured by the robot (green). Right: histogram of differences between the curves.

6.2 Accuracy and Latency

Another important aspect of a real-time system it its latency, *i.e.* the delay between the point of time that an event occurs, and the time its effect occurs in the system's output. For RTblob, we can estimate the total latency by attaching a trackable object directly to the robot arm's and moving the arm in a trajectory that is an overlay of sinusoids with different frequency and phase. The latency can then be read off as the time offset between the trajectories as recorded by the robot, and the trajectories measured by the vision system. In our setup, latency is typically one, sometimes frame units, so not more than 10ms when the cameras run at their maximal speed.

With the same experimental setup, we can also measure the tracking accuracy. For this, we calculate the mean distance between the trajectories after having compensated for the time delay. Figure 5 shows an excerpt of the resulting X-, Y- and Z-curves for a tracking run with 60 Hz. It shows that apart from an occasional overestimation of the x-coordinate (which is the *depth* direction and therefore most susceptible to calibration artifacts), the accuracy is generally below 20 millimeters in each coordinate. This is a very satisfactory result in our situation where the observed scene is located more than 5 meters from the cameras.

7 Extensions and Future Work

As explained in the introduction, it is part of RTblob's design decisions to provide a simple and modular base platform for object detection. Several possible extensions and methods for achieving increased performance and robustness come to mind.

In particular, many camera setups support higher frame rates when using *regions of interest*. In combination with a tracking approach, this could yield even higher framerates support without increased hardware requirements. A problem of the current linear detection step is the dependence of the optimal filter mask to changes of the environmental conditions, in particular global illumination. Computer vision research has invented a number of technique to overcome this, *e.g.* gradient-based features and these could also be integrated into RTblob without sacrificing too much of its simplicity and speed. Alternatively, the detection module could be exchanged for a Hough-transform based setup, which is known to be more robust to illumination changes, but requires more effort to parallelize on the GPU.

A similar drawback of the chosen LSI detection is that a single filtering step only detects objects of fixed size. While complete scale independence would not be a desirable property, because the observed size is a valuable cue to distinguish between the object and background components of the same color, a limited tolerance to scale changes is necessary in order to deal with the effects of perspective. Currently, the filter masks has to be chosen accordingly, which limits the object shapes we can search for, or one has to process the image multiple times with filters that correspond to differently sized objects. A more elegant solution would be to work in a scale space representation, see [33]. Adapting the filtering stage accordingly should introduce only relatively little overhead, because all modern graphics cards have builtin routines to efficiently rescale images. Introducing rotation invariance in this way is more difficult, but at least for subclasses of object, such as elliptic ones, we consider steerable filters [41] a promising direction and we plan to explore this in future work.

8 Conclusion

We have described RTblob, to our knowledge the first software system for high-speed 3D object detection that requires only off-the-shelf hardware components and is publicly available. RTblob is not the most precise object detection system available. Commercial solutions, *e.g.* the *Vicon MOTUS* system achieve higher precision even for articulate objects, but they are also less flexible and much more expensive. It is also not the overall fastest solution for object detection: some hardware-based solutions, *e.g.* [16, 18], achieve higher throughput, even though at a lower resolution and only by using much more expensive hardware. RTblob is designed with the objective of finding an optimal compromise: on the one hand, it is fast and accurate enough for challenging realistic applications, as we have demonstrated by its use as visual input source for a robot table tennis player. At the same time, because it consists only of free software components and runs on standard PC hardware, it is inexpensive and almost effortless to set up. Consequently, we believe that RTblob has the potential to become the seed for a standard platform for interactive applications as well as a valuable teaching tools for GPU accelerated high-speed vision tasks.

References

1. U. Handmann, T. Kalinke, C. Tzomakas, M. Werner, and W. Seelen, "An image processing system for driver assistance," *Image and Vision Computing*, vol. 18, no. 5, pp. 367–376, 2000.
2. J. Miura, T. Kanda, and Y. Shirai, "An active vision system for real-time traffic sign recognition," in *IEEE Intelligent Transportation Systems*, 2000, pp. 52–57.
3. D. Tock and I. Craw, "Tracking and measuring drivers' eyes," *Image and Vision Computing*, vol. 14, no. 8, pp. 541–547, 1996.
4. T. Kirishima, K. Sato, and K. Chihara, "Real-time gesture recognition by learning and selective control of visual interest points," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, no. 3, pp. 351–364, 2005.
5. R. H. Liang and M. Ouhyoung, "A real-time continuous gesture recognition system for sign language," in *International Conference on Automatic Face and Gesture Recognition*, 1998, pp. 558–567.
6. C. Stauffer and W. E. L. Grimson, "Learning patterns of activity using real-time tracking," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 8, pp. 747–757, 2000.
7. B. Williams, G. Klein, and I. Reid, "Real-time SLAM relocalisation," in *International Conference on Computer Vision*, 2007.
8. A. Davison, W. Mayol, and D. Murray, "Real-time localisation and mapping with wearable active vision," in *Proceedings of the IEEE International Symposium on Mixed and Augmented Reality, Tokyo*, 2003.
9. A. Comport, E. Marchand, and F. Chaumette, "Statistically robust 2d visual servoing," *IEEE Transactions on Robotics*, vol. 22, no. 2, pp. 415–421, April 2006.
10. U. Frese, B. Bäuml, S. Haidacher, G. Schreiber, I. Schaefer, M. Hähnle, and G. Hirzinger, "Off-the-shelf vision for a robotic ball catcher," in *IEEE/RSJ International Conference on Intelligent Robots and Systems, 2001. Proceedings*, vol. 3, 2001.

11. B. Jähne, *Digital image processing: concepts, algorithms, and scientific applications*. Springer Berlin, 1995.
12. S. Mizusawa, A. Namiki, and M. Ishikawa, "Tweezers type tool manipulation by a multifingered hand using a high-speed visusal servoing," in *IEEE/RSJ International Conference on Intelligent Robots and Systems, 2008. IROS 2008*, 2008, pp. 2709–2714.
13. P. Viola and M. J. Jones, "Robust real-time face detection," *International Journal of Computer Vision*, vol. 57, no. 2, pp. 137–154, 2004.
14. S. Avidan, "Support vector tracking," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 8, pp. 1064–1072, 2004.
15. K. Shimizu and S. Hirai, "CMOS+FPGA vision system for visual feedback of mechanical systems," in *IEEE International Conference on Robotics and Automation*, 2006, pp. 2060–2065.
16. Y. Watanabe, T. Komuro, and M. Ishikawa, "955-fps real-time shape measurement of a moving/deforming object using high-speed vision for numerous-point analysis," in *IEEE International Conference on Robotics and Automation*, 2007, pp. 3192–3197.
17. U. Mühlmann, M. Ribo, P. Lang, and A. Pinz, "A new high speed CMOS camera for real-time tracking applications," in *International Conference on Robotics And Automation*, 2004, pp. 5195–5200.
18. Y. Nakabo, M. Ishikawa, H. Toyoda, and S. Mizuno, "1ms column parallel vision system and its application of high speed target tracking," in *IEEE International Conference on Robotics and Automation*, 2000, pp. 650–655.
19. "Compute Unified Device Architecture Programming Guide," NVIDIA. Santa Clara, CA, 2007.
20. "Camera link: Specifications of the camera link interface standard for digital cameras and frame grabbers," http://www.machinevisiononline.org/public/articles/index.cfm?cat=129, 2000.
21. "GigE vision: Camera interface standard for machine vision," http://www.machinevisiononline.org/public/articles/index.cfm?cat=167, 2005.
22. X. Tong, H. Lu, and Q. Liu, "An effective and fast soccer ball detection and tracking method," in *International Conference on Pattern Recognition*, vol. 4, 2004, pp. 795–798.
23. G. S. Pingali, Y. Jean, and I. Carlbom, "Real time tracking for enhanced tennis broadcasts," in *IEEE Conference on Computer Vision and Pattern Recognition*, 1998, pp. 260–265.
24. T. d'Orazio, C. Guaragnella, M. Leo, and A. Distante, "A new algorithm for ball recognition using circle Hough transform and neural classifer," *Pattern Recognition*, vol. 37, pp. 393–408, 2003.
25. R. Kimmel, "Demosaicing: image reconstruction from color CCD samples," *IEEE Transactions on Image Processing*, vol. 8, no. 9, pp. 1221–1228, 1999.
26. J. Lewis, "Fast template matching," in *Vision Interface*, vol. 10, 1995, pp. 120–123.
27. R. Brunelli and T. Poggio, "Template matching: Matched spatial filters and beyond," *Pattern Recognition*, vol. 30, no. 5, pp. 751–768, 1997.
28. F. Crow, "Summed-area tables for texture mapping," *Computer Graphics*, vol. 18, no. 3, 1984.
29. G. E. Blelloch, *Prefix Sums and Their Applications*. Morgan Kaufmann, 1991.
30. M. Harris, S. Sengupta, and J. Owens, "Parallel prefix sum (scan) with CUDA," *GPU Gems*, vol. 3, no. 39, pp. 851–876, 2007.
31. Q. Tian and M. Huhns, "Algorithms for subpixel registration," *Computer Vision, Graphics, and Image Processing*, vol. 35, no. 2, pp. 220–233, 1986.
32. R. Hartley and A. Zisserman, *Multiple view geometry*, 2nd ed. Cambridge University Press, 2000.
33. T. Lindeberg, *Scale-Space Theory in Computer Vision*. Kluwer Academic Publishers Norwell, MA, USA, 1994.
34. S. Hinz, "Fast and subpixel precise blob detection and attribution," in *IEEE International Conference on Image Processing*, vol. 3, 2005, pp. 457–460.
35. G. Turin, "An introduction to matched filters," *IRE Transactions on Information Theory*, vol. 6, no. 3, pp. 311–329, 1960.
36. J. Zeidler, J. McCool, and B. Widrow, "Adaptive correlator," 1982, US Patent 4,355,368.
37. D. Flannery and S. Cartwright, "Optical adaptive correlator," in *Third Annual Aerospace Applications of Artificial Intelligence Conference*, 1987, pp. 143–154.
38. K. Sung and T. Poggio, "Example-based learning for view-based human face detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 1, pp. 39–51, 1998.
39. C. H. Lampert and J. Peters, "Active structured learning for high-speed object detection," in *Symposium of the German Pattern Recognition Society*, 2009, pp. 221–231.
40. B. Triggs, P. McLauchlan, R. Hartley, and A. Fitzgibbon, "Bundle adjustment – a modern synthesis," in *Vision Algorithms: Theory and Practice*, ser. LNCS, vol. 1883, 2000, pp. 298–372.
41. W. T. Freeman and E. H. Adelson, "The design and use of steerable filters," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 13, no. 9, pp. 891–906, 1991.