Technical Report manuscript No.
(will be inserted by the editor)

David Exner · Erich Bruns · Daniel Kurz · Anselm Grundhöfer · Oliver Bimber

Fast and Reliable CAMShift Tracking

Abstract CAMShift is a well-established and fundamental algorithm for kernel-based visual object tracking. While it performs well with objects that have a simple and constant appearance, it is not robust in more complex cases. As it solely relies on back projected probabilities it can fail in cases when the object's appearance changes (e.g. due to object or camera movement, or due to lighting changes), when similarly colored objects have to be re-detected or when they cross their trajectories. We propose extensions to CAMShift that address and resolve all of these problems. They allow the accumulation of multiple histograms to model more complex object appearance and the continuous monitoring of object identities to handle ambiguous cases of partial or full occlusion. Most steps of our method are carried out on the GPU for achieving real-time tracking of multiple targets simultaneously. We explain an efficient GPU implementations of histogram generation, probability back projection, image moments computations, and histogram intersection. All of these techniques make full use of a GPU's high parallelization.

Keywords Kernel-Based Tracking, CAMShift, Appearance Modeling, Re-Detection, Object-Identification, Real-Time

1 Introduction

The CAMShift algorithm [1] was derived from the earlier Mean Shift algorithm [2] and is a simple, yet very effective, color-based tracking technique. It is applied as basic component in many advanced trackers.

David Exner, Erich Bruns, Daniel Kurz, and Anselm Grundhöfer
Bauhaus-University Weimar, Germany
E-mail: {firstname.lastname}@medien.uni-weimar.de

Oliver Bimber
Bauhaus-University Weimar, Germany
Johannes Kepler University Linz, Austria
E-mail: oliver.bimber@jku.at

CAMShift essentially climbs the gradient of a back-projected probability distribution computed from re-scaled (to a range of $[0, 1]$) color histograms to find the nearest peak within an axis-aligned search window. With this, the mean location of a target object is found by computing zeroth, first and second order image moments

$$M_{00} = \sum_x \sum_y P(x, y), \tag{1}$$

$$M_{10} = \sum_x \sum_y xP(x, y);\ M_{01} = \sum_x \sum_y yP(x, y), \tag{2}$$

$$M_{20} = \sum_x \sum_y x^2P(x, y);\ M_{02} = \sum_x \sum_y y^2P(x, y), \tag{3}$$

where $P(x, y) = h(I(x, y))$ is the back projected probability distribution at position x, y within the search window $I(x, y)$ that is computed from the histogram h of I. The target object's mean position can then be computed with

$$x_c = \frac{M_{10}}{M_{00}};\ y_c = \frac{M_{01}}{M_{00}}, \tag{4}$$

while its aspect ratio

$$ratio = \frac{\frac{M_{20}}{{x_c}^2}}{\frac{M_{02}}{{y_c}^2}}, \tag{5}$$

is used for updating the search window with

$$width = 2M_{00} \cdot ratio;\ height = 2M_{00}/ratio. \tag{6}$$

The position and dimensions of the search window are updated iteratively until convergence.
One of the main drawbacks of standard CAMShift tracking is, that it is prone to tracking failures caused by objects with similar colors. The reason for this is that only the peak of back-projected probability distribution is tracked without paying attention to color composition. For the same reason, objects with similar colors can not be distinguished. Stable tracking despite appearance changes (e.g., due to lighting or perspective) and partial or full occlusion, and re-detection of lost objects are

other problems of standard CAMShift.

Overview and Contribution: The remainder of this article is organized as follows: Section 2 reviews the related work and distinguishes our approach from existing ones. Section 3 explains our extensions to the basic CAMShift algorithm as outlined above. We describe how to accumulate multiple histograms for making CAMShift more robust against appearance changes (section 3.1), how to support object identification for tracking multiple targets in cases of partial of full occlusion (section 3.2), and how to realize a fast and reliable re-detection of lost targets (section 3.3). Section 4 presents a GPU implementation of our extended CAMShift tracker that enables the simultaneous and robust tracking of multiple targets in real-time. In particular, we explain how histograms can be generated fast (section 4.1), how probability distributions can efficiently be back-projected (section 4.2), how image moments can be computed in parallel 4.3, and how histogram intersection can efficiently be carried out on the GPU (4.4). Section 4.5 summarizes how all these components interplay – partially on the GPU and partially on the CPU. Finally, section 5 evaluates our approach, and compares it to the CAMShift implementation of OpenCV, which is most commonly applied. We compare our extended GPU CAMShift with both – the standard CPU implementation as available in OpenCV, and our own GPU re-implementation of standart CAMShift.

2 Related Work

Since its introduction as a technique for face tracking, CAMShift has been object to a variety of modifications to accommodate other tracking applications.
An adaptive background model was proposed in [4] for tracking targets in front of similar backgrounds.
Prior to histogram back-projection, multidimensional histograms have been weighted with monotonically decreasing kernels in [3] to reduce the influence of similar background colors.
Similar to this, hue-saturation and saturation-value color spaces have been applied in [5] for computing joint probability density distributions to model human faces and hair, respectively.
An adaptive histogram computation for CAMShift tracking has been introduced in [6]. This allow tracking an object with changing appearance. In order to avoid background pixels to negatively influence the histogram, it is only updated when the back projected probabilities are excellent (i.e., when the objects track is clear). In cases the object appearance changes rapidly, however, the histogram is not updated and the objects track is lost.

3 Extended CAMShift Tracking

In this section, we describe several extensions to the standard CAMShift algorithm for making it more robust against similar object colors and appearance changes, and to enable object identification and re-detection of lost objects.
Note, that each subsection validates the advantage of one individual extension only, while all other extensions are always enabled and used in addition. Comparing against standard CAMShift, that does not incorporate any of these extensions will make the full benefit of our approach clear. This comparison is presented in section 5.

3.1 Accumulating Multiple Histograms

Using a single histogram is a reasonable choice to represent simple objects, such as ones that have the same appearance on every side or objects that do not change their appearance over time (e.g., through different lighting). If objects have a more complex appearance, standard CAMShift will most likely fail. The reason for this is, that the back-projected probabilities are low for under-represented appearance conditions, such as differently colored sides of an object.
Our solution is simple, but effective: We utilize an arbitrary number of histograms to model different appearances of target objects. For every appearance condition that strongly varies in color, we pre-compute and store one re-scaled ($[0, 1]$) reference histogram. In addition, we sum all reference histograms that belong to the same object, and re-scale the result back to $[0, 1]$.
During run-time, this accumulated histogram is then used for computing the probability back-projection of the corresponding object, while the individual reference histograms are only applied for object identification, as explained in section 3.2. Figure 1 illustrates the difference between tracking with a single reference histogram and with multiple accumulated histograms to consider different appearances of the same object.
Accumulating multiple histograms for determining the probability distribution of the same object provides an ad-hoc and compact multi-view appearance representation. It does not cost additional performance, since the histogram accumulation and re-scaling is computed offline. Additional reference histograms can be added and deleted at any time for updateing the accumulated histogram.

3.2 Object Identification

In principle, CAMShift can track multiple objects simultaneously by converging individual search windows to each corresponding target object. This is straight forward, if the back-projected probabilities of all objects

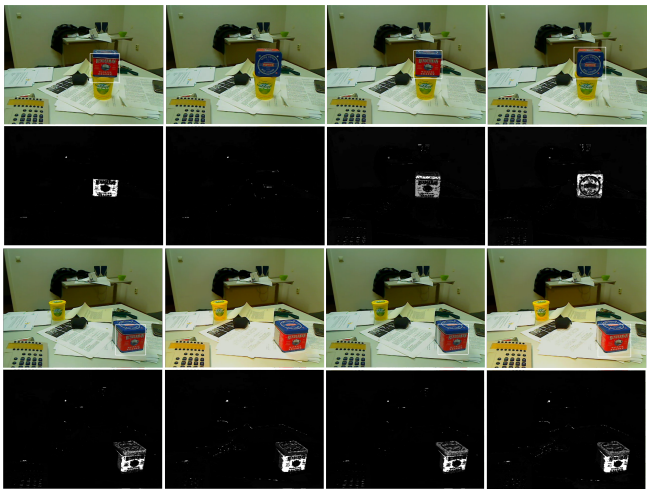

Fig. 1 Left two columns: If an object is tracked with only one reference histogram, CAMShift fails if appearance changes strongly (e.g., object is rotated (top rows) or the lighting conditions change (bottom rows)). Right two columns: If two references histograms are accumulated for the two corresponding appearance conditions, CAMShift succeeds tracking the object under both conditions. The respective lower rows visualize the back-projected probability distribution for one histogram (computed for the left case) and for the accumulation of both histograms.

are substantially different, or similar objects don't overlap. A problem arises, however, when similar objects do interact. Their back-projected probabilities then have almost the same values, which results in a drift as soon as one object occludes or crosses the trajectory of another, resembling object. In this case, the search window will stay attached to the occluding object while the occluded one is lost.

The problem of search window drift is inherent to many probability-based trackers, such as standard CAMShift, since these techniques only track the peak of a probability distribution – not taking into account the composition of probabilities.

We solve this problem by continuously monitoring the identity of every target to regain stable tracks after partial or full occlusions: After successfully tracking an object, a histogram of its search region is computed, re-scaled and matched against the appearances reference histograms (see section 3.1) of the tracked object via histogram intersection.

Histogram intersection is defined by

$$d(h,g) = \sum_{b=0}^{B-1} \min(h(b), g(b)), \tag{7}$$

where $h(b)$ and $g(b)$ are the values stored in the bins b of the histograms of size B.

The best match with the target histogram t among all possible (m) reference histograms $r_j, 0 \leq j < m$ is the one that maximizes $d(t, r_j)$.

If the maximum of $d(t, r_j)$ falls below a certain threshold, the object's track is marked lost and the re-detection process is launched. This is explained in section 3.3.

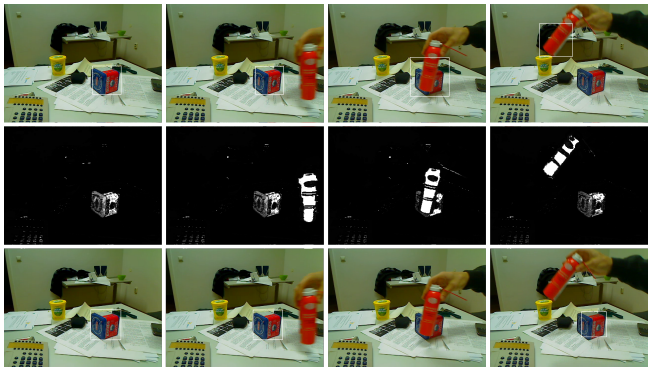

Fig. 2 Upper row: Tracking with occlusion fails for standard CAMShift if objects have similar colors (the box is tracked while the bottle is used for interference). Lower row: Matching histograms to monitor all objects supports partial or full occlusion. The back-projected probability distributions are shown in all cases. Center row: The same probability back-projection is used for both cases.

The frame-by-frame computation, re-scaling and matching of histograms for every object is an additional effort. However, it provides far more stable tracks in cases of occlusions than omitting the object identification. An example is shown in figure 2, and in the accompanying video.

3.3 Hierarchical Re-Detection

Since objects frequently become fully occluded and cannot be monitored with the technique described above, or temporarily leave the camera's field of view, a reliable method has to be employed to stably regain tracks as soon as the objects reappear.

For a probability-based method, such as standard CAMShift, the re-detection is likely to fail when –due to similar colors– more than one back-projected probability blob is present in the actual search region. The reason for this is, that standard CAMShift always converges to the largest blob of probabilities within that region.

To overcome this, we apply a hierarchical quad-tree re-detection strategy. If one or multiple objects are lost, we begin the re-detection with a search window that covers the whole image. The zeroth moment of the actual search windows serves as one exit condition for our recursion. If it is substantially small, the object is not present within its boundaries and further processing is skipped. Otherwise, the region is split into four, and their moments are recomputed.

In each quadrant, our extended CAMShift tracker is applied to re-detect the lost object. By doing so, the tracker will, in all four cases, readjust all four search window instances accordingly. If these search windows converge

to the same region as the search window of their parent level, and the best match of the identified object is above a threshold, we stop the recursion. In this case, a lost object was re-detected within this region, its track is marked as valid again and tracking continues normally. Otherwise, the recursive subdivision continuous and is applied to each of the four quadrants.

4 GPU Implementation

This section explains how various aspect of our extended CAMShift tracker can be optimally implemented to reach real-time performance. Thereby, we try to achieve an optimal load-balancing between CPU and GPU, reduce up- and down-load sizes of exchanged data structures, and take as much advantage as possible of SIMD parallel processing on the GPU. Our techniques are described for the OpenGL API and its programmable shading pipeline.

4.1 Histogram Generation

CAMShift makes heavy use of 2D or 3D color histograms (e.g., HSV, YUV, or RGB) for computing back-projected probability distributions. Therefore, a fast implementation of histogram generation is essential for a fast CAMShift implementation.
Given that a histogram generation is merely a counting and sorting of pixels corresponding to their values, vertex or geometry shaders are efficient tools for supporting this process in real-time (cf. figure 3).
In principle, we can assign a vertex to each pixel in the camera image and store it in a vertex buffer object (VBO). For a given search window region for which a histogram has to be computed, the corresponding list of indexed vertices are rendered by the GPU. We assign a texture coordinate to each vertex that links it to its corresponding pixel position in the camera image texture. This texture coordinate serves as a look up of the actual color value linked to each vertex. Depending on these values the vertices' positions are transformed to the according bin position of a texture bound to a frame buffer object (FBO) which will store the histogram data. The alpha value of each transformed vertex is set to 1.0, which, in combination with additive alpha blending, is used to fill the bins of the histogram. After rendering the vertices of all required pixels, the histogram is complete, but still not scaled to $[0, 1]$. An occlusion query allows to count the number of valid histogram entries during the generation. This query returns the number of pixels that pass the rasterizer. All pixels with invalid colors (e.g. colors with undefined hue or saturation values) are not written to the histogram texture in the FBO, and are therefore not rasterized and counted. After the histogram is complete, we can use the final count determined by the occlusion query to subsequently re-scale the histogram by means of a fragment shader.
Since the size of the VBO is constant, it has to be initialized only once while the indices of vertices that need to be rendered are computed on the fly – depending on the actual search window. Obviously the number of rendered vertices influences the processing time of the histogram generation. This has multiple reasons: The required additive blending step only can be carried out sequentially for each histogram bin and the number of shading units of the used GPU constrains the number of vertices which can be processed in parallel. The latter can be optimized for our approach in a way similar to the method proposed in [7]. We render only every ith vertex (in both dimensions) instead a vertex for each individual pixel. This reduces the computational load on the vertex shader. For each incoming seed vertex a geometry shader then generates all vertices for the corresponding sub-region (i.e., the seed vertex and its i^2-1 neighboring vertices at, for example, the lower right area of each seed vertex, as shown in figure 3). This is be done in parallel for each seed vertex, until the number of geometry shader units is exceeded. The optimal choice of i is therefore related to the number of available geometry shader units. For unified shaders, this number is not constant, but depends on the actual load balancing of the GPU. Empirically we found that an i of 4 is optimal in our case (i.e., 15 neighbors per seed vertex).
While HS and YU histograms are 2D, RGB histograms are 3D. However, we still store 3D histograms in 2D textures by tiling 2D color planes. A 16x16x16 RGB histogram, for instance, would be tiled into 16 (indexing B) 16x16 RG color planes. For this example, they are indexed with the simple modulo operation $x = B * 15/4$ and $y = B * 15\%4$, where $0 \leq B \leq 1$ is the blue color value.

4.2 Probability Back-Projection

The back-projection of probability distributions is carried out through indexing the histogram texture by using the pixel colors (e.g., RGB, HS, or YUV, depending on the histogram type being used) as texture coordinates. If, as explained in section 3.1, more than one reference histogram exist, the corresponding accumulated histogram is indexed. This is done in an individual fragment shader.

4.3 Computations of Image Moments

The computation of image moments in each iteration is another crucial step of the CAMShift process, as explained in section 1.
Our GPU implementation applies a similar strategy as explained for the histogram generation in section 4.1. For every ith pixel in the given search window region of a back-projected probability distribution texture, we directly render a vertex with texture coordinates that point

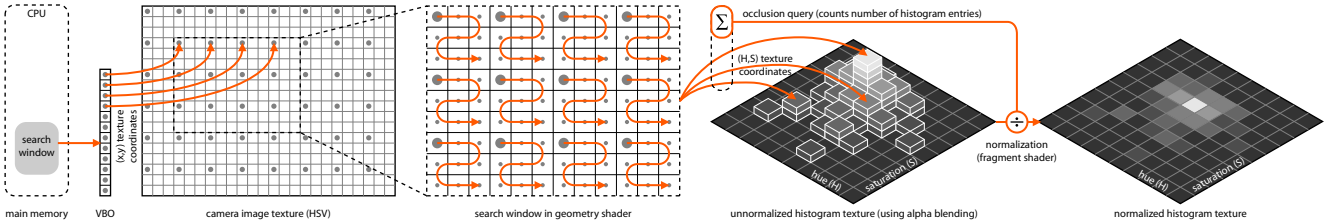

Fig. 3 Histogram generation on GPU.

to the associated pixel. The probability of this seed pixel is summed with the probabilities of the remaining i^2-1 neighbors using a vertex shader. Within this neighborhood, pixels are accessed by iterating texture look-ups in a loop. To compute all five image moments at the same time, as in equations 1,2 and 3, the x, y, x^2 and y^2 factors are directly multiplied during summation and the final five values are stored in the RGBRG channels of two 1x1 textures of a single FBO. Since all seed vertices are actually rendered to the same FBO pixel (i.e., a texel of one of the two textures that are bound to the FBO), again, additive alpha blending is applied to sum the single contributions of all seed vertices within the moment-individual color channels.
The final result are the values of the five image moments for the entire search window stored in the RGBRG channels of the two 1x1 textures, assigned to one FBO pixel. Only these five values have to be read back to the CPU, thus minimizing data transfer between VRAM to normal RAM. This process is illustrated in figure 4.

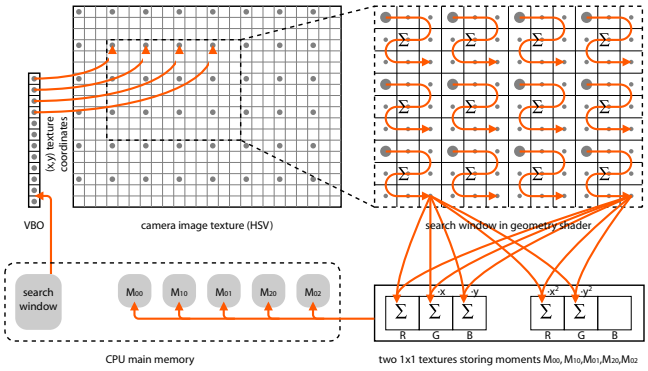

Fig. 4 Computation of image moments on GPU.

Once again, a good choice of i (i.e., the seed pixel resolution in both dimensions) is important to optimally exploit the GPUs parallelization. Each seed vertex can be processed in parallel until the number of vertex shader units is exceeded. The number of per-vertex loop iterations sets another limit, as each loop invokes costly texture lookups. Similar to the histogram generation (section 4.1), the ideal number of seed vertices is related to the number of available vertex shader units. Again, for unified shaders, this number is not constant, but depends on the actual load balancing of the GPU. Empirically we found that an i of 8 is optimal in our case (i.e., 63 neighbors per seed vertex).

4.4 Histogram Intersection

To compute the intersection of two histograms (eqn. 7), we also apply a VBO containing as many vertices as histogram bins that each point to the coordinates of the bins of the two histogram textures. A vertex shader renders all of these vertices into the same FBO pixel (i.e., a texel of a 1x1 texture that is bound to the FBO), where the minimum of both corresponding bin entries are summed over all vertices (i.e., bins) through additive alpha blending. The result of the histogram intersection is then stored in this single FBO pixel, which can efficiently be read-back to the CPU. This is illustrated in figure 5.

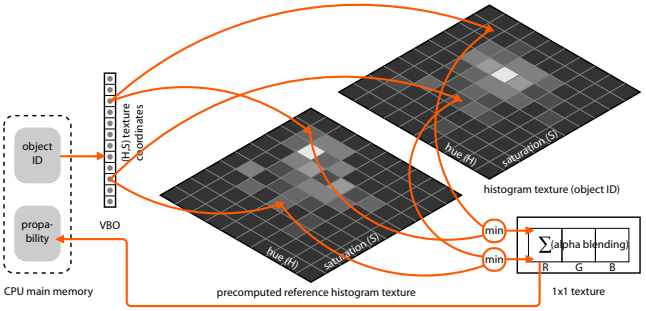

Fig. 5 Histogram intersection on GPU.

A lower resolution sampling of the VBO as it is the case for computing the image moments or for generating the histograms is not necessary in this case, since the histogram resolutions are significantly lower than the image or search window resolutions.

4.5 Summary of Steps

Figure 6 summarizes the sequence of all steps that are carried out on the CPU as well as on the GPU. Initially, the reference histograms of all objects under all appearance conditions are uploaded to and stored on the GPU. Initially, the reference histograms of all objects under all

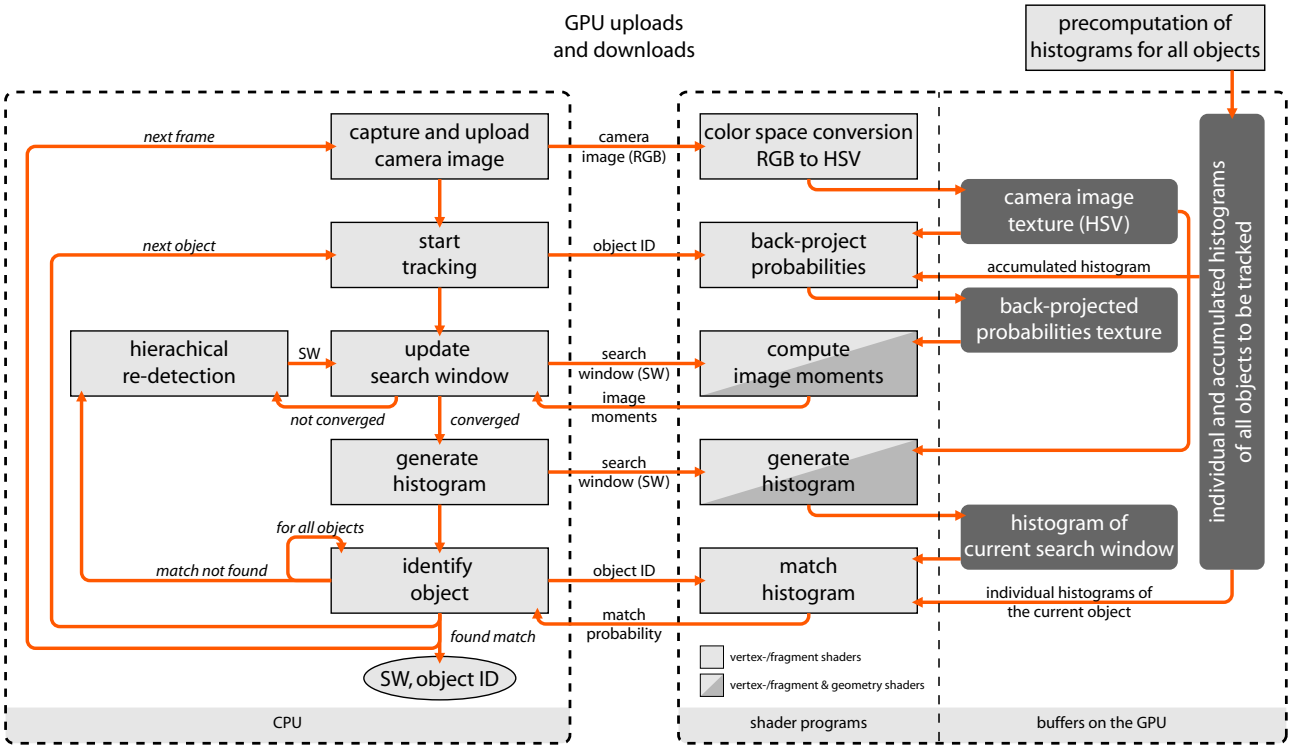

Fig. 6 Overview of CPU and GPU steps of extended CAMShift with up- and downloads. Computation of image moments, histogram generation, and histogram intersection are illustrated in more detail in figures 3 - 5.

appearances are computed, re-scaled and stored to the GPU. Additionally, the accumulated histograms of all objects are computed, re-scaled and also stored.
During runtime, each camera image is uploaded and optionally converted into the desired color space (e.g. HSV, as shown in the examples). For each target object, the CPU triggers the tracking process, and the object individual probability distribution is back-projected on the GPU. The CAMShift iterations are managed on the CPU, while all time consuming operations are carried out on the GPU. For a given search window (full camera image initially), the image moments are computed and returned. The CPU computes the updated position and size. If these parameters have converged (compared to previous iterations), the histogram of the search window area is generated and intersected with all reference histograms to identify the object. If a match was found, the object has been identified and its position and size are determined. If no match was found, the object's track was lost and the hierarchical re-detection is triggered for this object. The same applies, if the search window does not converge. The re-detection recursion is also managed on the CPU to update all hierarchical search sub-windows. The recursion terminates as soon as one of the two exit conditions (i.e., low zeroth moment or convergence of search window in two hierarchy levels) are fulfilled, as explained in section 3.3. This is repeated for each target object and each camera image.

5 Evaluation

Figures 7 and 8 illustrate the the timings that were taken on a Core2Duo 3GHz PC with 3.25GB RAM (CPU) and a NVIDIA GeForce GTX 285 with 1024 MB VRAM (GPU). CPU and GPU are comparable mid-level devices.
The color coding indicates which portions of the algorithm has to be executed per frame and per target object, and which ones are variant to the search window size or to the histogram resolution. In all cases, CAMShift was computed in HSV color space.
We evaluated a fixed VGA image resolution, a HS histogram resolution of 32x6 bins, and tested against different search window sizes.
On the GPU, we timed our extended CAMShift tracker and a re-implementation of a standard CAMShift tracker without extensions, and with and without color space conversion. On the CPU, we timed standard CAMShift (OpenCV implementation) with and without color space conversion.
Our extended CAMShift on the GPU outperforms the standard CAMShift on the CPU by a factor of approximately 7.2-13.0 (10.3 on average). To allow a direct comparison, all timings are for one tracked object and one reference histogram. For extended CAMShift, the processing time for probability back projection, image moments computation and object identification increases

CAMShift (GPU)

object size / search window size		66x66	75x75	110x110	150x150	210x210	254x254	300x300	360x360	425x425	500x480	525x480	640x480
upload (image resolution: 640x480)		0,0078576	0,0078576	0,0078576	0,0078576	0,0078576	0,0078576	0,0078576	0,0078576	0,0078576	0,0078576	0,0078576	0,0078576
RGB2HSV (image resolution: 640x480)		0,2524690	0,2524690	0,2524690	0,2524690	0,2524690	0,2524690	0,2524690	0,2524690	0,2524690	0,2524690	0,2524690	0,2524690
probability back projection (image resolution: 640x480)		0,1772700	0,1772700	0,1772700	0,1772700	0,1772700	0,1772700	0,1772700	0,1772700	0,1772700	0,1772700	0,1772700	0,1772700
image moments		0,2128640	0,2214190	0,2204340	0,2432310	0,2653210	0,2618590	0,3076920	0,3422530	0,3750360	0,4403180	0,4474350	0,4528160
object identification (histogram size: 32x6)	histogram generation	0,2066480	0,2117740	0,2496630	0,2925430	0,3946590	0,5049310	0,5942130	0,6849230	0,8221900	1,0227400	1,0470900	1,2250700
	histogram normalization	0,0999987	0,0999987	0,0999987	0,0999987	0,0999987	0,0999987	0,0999987	0,0999987	0,0999987	0,0999987	0,0999987	0,0999987
	histogram match	0,1613750	0,1613750	0,1613750	0,1613750	0,1613750	0,1613750	0,1613750	0,1613750	0,1613750	0,1613750	0,1613750	0,1613750
total (extended CAMShift)		**1,1184823**	**1,1321633**	**1,1690673**	**1,2347443**	**1,3589503**	**1,4657603**	**1,6008753**	**1,7261463**	**1,8961963**	**2,1620283**	**2,1934953**	**2,3768563**
total (standard CAMShift with RGB2HSV)		**0,6504606**	**0,6590156**	**0,6580306**	**0,6808276**	**0,7029176**	**0,6994556**	**0,7452886**	**0,7798496**	**0,8126326**	**0,8779146**	**0,8850316**	**0,8904126**
total (standard CAMShift without RGB2HSV)		**0,3979916**	**0,4065466**	**0,4055616**	**0,4283586**	**0,4504486**	**0,4469866**	**0,4928196**	**0,5273806**	**0,5601636**	**0,6254456**	**0,6325626**	**0,6379436**

CAMShift (CPU)

object size / search window size	66x66	75x75	110x110	150x150	210x210	254x254	300x300	360x360	425x425	500x480	525x480	640x480
RGB2HSV (image resolution: 640x480)	12,8759000	12,8759000	12,8759000	12,8759000	12,8759000	12,8759000	12,8759000	12,8759000	12,8759000	12,8759000	12,8759000	12,8759000
probability back projection	1,6248200	1,6248200	1,6248200	1,6248200	1,6248200	1,6248200	1,6248200	1,6248200	1,6248200	1,6248200	1,6248200	1,6248200
image moments	0,0590979	0,0703719	0,1300480	0,2223960	0,4238720	0,5958240	0,8002350	2,2153200	3,0853400	3,0550700	3,1391800	2,5212200
total (standard CAMShift with RGB2HSV)	**14,5598179**	**14,5710919**	**14,6307680**	**14,7231160**	**14,9245920**	**15,0965440**	**15,3009550**	**16,7160400**	**17,5860600**	**17,5557900**	**17,6399000**	**17,0219400**
total (standard CAMShift without RGB2HSV)	**1,6839179**	**1,6951919**	**1,7548680**	**1,8472160**	**2,0486920**	**2,2206440**	**2,4250550**	**3,8401400**	**4,7101600**	**4,6798900**	**4,7640000**	**4,1460400**

per frame (independent of #objects, dependent on image resolution) — dependent on search window size
per frame, per object — dependent on histogram size

Fig. 7 Timings of different components of extended/standard CAMShift on GPU and standard CAMShift on CPU.

linearly with each additional object. The latter also increases linearly for each additional reference histogram. The main bottleneck of standard CAMShift on the CPU is the color space conversion. Disabling color space conversion, however, our own GPU implementation of standard CAMShift (i.e., without the extensions explained in section 3, but with GPU implementations of the remaining parts, as described section 4) is still by a factor of approximately 4.2-8.4 (5.7 on average) faster than the CPU implementation of standard CAMShift.

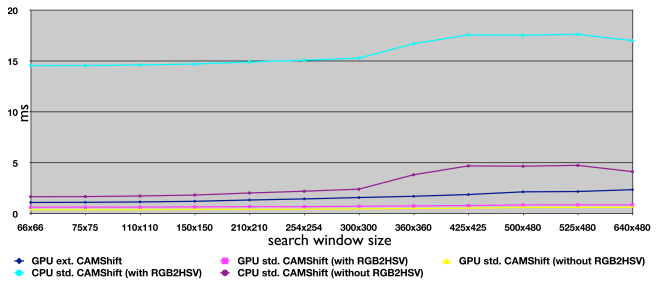

Fig. 8 Overall performance of CPU/GPU standard/extended CAMShift with respect to an increasing search window size.

Figure 8 plots the overall performances of the individual CPU and GPU variations for different search window sizes.
Compared to standard CAMShift, the extensions do cost extra performance. Our extended CAMShift is by a factor of approximately 1.7-2.7 (2.1 on average) slower, than our implementation of standard CAMShift on the GPU. Yet, these extensions make CAMShift tracking significantly more reliable. The accompanying video compares the robustness of standard CAMShift and extended CAMShift under different conditions.

References

1. Bradski, G.R.: Computer Vision Face Tracking For Use in a Perceptual User Interface. Intel Technology Journal **Q2** (1998)
2. Cheng, Y.: Mean Shift, Mode Seeking, and Clustering. IEEE Transactions on Pattern Analysis and Machine Intelligence, **17(8)**, 790-799 (1995)
3. Allen, J.G., Xu, R.Y.D., Jin, J.S.: Object tracking using CamShift algorithm and multiple quantized feature spaces. Proc. of the Pan-Sydney area Workshop on Visual Information Processing, 3-7 (2004)
4. Stolkin, R., Florescu, I., Kamberov, G.: An Adaptive Background Model for CamShift Tracking with a Moving Camera. Proc. of the 6th International Conference on Advances in Pattern Recognition (2007)
5. Xiang, G., Wang, X.: Real-time follow-up head tracking in dynamic complex environments. Journal of Shanghai Jiaotong University (Science), **14(5)**, 593-599 (2009)
6. See, A., Bin, K., Kang, L.Y.: Face detection and tracking utilizing enhanced CamShift model. International Journal of Innovative Computing, Information and Control (2006)
7. Diard, F.: Using the Geometry Shader for Compact and Variable-Length GPU Feedback. GPU Gems 3, **chap. 41**, Addison-Wesley Professional, 895-897 (2007)