Fast and Robust CAMShift Tracking

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

Oliver Bimber
Johannes Kepler University Linz, Austria
oliver.bimber@jku.at

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 low-cost extensions to CAMShift that address and resolve all of these problems. They allow the accumulation of multiple histograms to model more complex object appearances 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 efficient GPU implementations of histogram generation, probability back projection, computation of image moments, and histogram intersection. All of these techniques make full use of a GPU's high parallelization capabilities.

1. Introduction

The CAMShift algorithm [5] was derived from the earlier Mean Shift algorithm [7] and is a simple, yet very effective, color-based tracking technique. It is applied as basic component for many advanced trackers and finds applications in fields such as perceptual user interfaces, face tracking, video surveillance, video editing, computer vision based game interfaces, and others.
CAMShift essentially climbs the gradient of a back-projected probability distribution computed from re-scaled 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 the 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

978-1-4244-7030-3/10/$26.00 ©2010 IEEE

re-detection of lost objects are other problems of standard CAMShift.

Overview and Contribution: The remainder of this paper 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 robust 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 (section 4.3), and how histogram intersection can efficiently be carried out on the GPU (section 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 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 standard 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. Mean Shift is a predecessor of CAMShift that does not update the search window size as explained in equation 6. Below, we will review the most related CAMShift and Mean Shift implementations and their extensions.

To compensate for different object appearances due to illumination changes or partial occlusion, Collins et al. [8] introduced an on-line feature ranking for Mean Shift tracking. The distributions of object and background pixels for a predefined image region are estimated based on 49 different color features. By computing the log-likelihood ratio of these distributions, a score is assigned to each feature by determining the variance ratio of the log-likelihood values. The continuously updated top-ranked features are then selected on-the-fly for tracking. This, however, will fail if the tracked object is lost.

Adam et al. [1] presented an algorithm that extends standard Mean Shift tracking by separating the search window into multiple patches. During tracking, the patches are individually tested by comparing histograms based on the Earth Mover's distance, and the results are combined. This make Mean Shift tracking more robust against partial occlusions.

Dadgostar et al. [10] explained two techniques for boundary detection of human faces with Mean Shift tracking. The search window size is increased or decreased based on the back-projected probability values at its edges. Resizing the search window is either based on a predefined threshold or through fuzzy rules. They compared both approaches with standard CAMShift and state that they are slower but could detect face boundaries more robustly in case of a higher noise level.

Allen et al. [3] applied 3D HSV histograms for CAMShift. The pixels in the search window that are used for histogram computation are weighted based on their locations within the search window as described in [9]. In addition, individual histogram bins are weighted based on a second histogram that is computed from the background as described in [9, 14]. This makes this approach robust for cases in which the object is similar to the background.

Xiang et al. [15] extended CAMShift with a particular focus on face tracking. They used a HS histogram for facial skin areas and a SV histogram for hair regions. These histograms are pre-computed from a database of face images. For tracking, the back-projected probabilities of both histograms are combined. This improves the tracking of faces in front of complex backgrounds.

Kok Bin et al. [4] explained another enhanced CAMShift implementation for face tracking. Pixels in the search window are also weighted as proposed in [3, 9]. Applying perceptual grouping to the back-projected probabilities helps to eliminate background noise as explained in [6]. Furthermore, they continuously adapt the target histogram by recomputing it for the current search window if the object was reliably tracked. A similar strategy is followed in [13] – yet, the histogram is always updated. This adapts to continuous appearance changes, but fails in case of abrupt changes.

Zhang et al. [16] combined Kalman filtering with CAMShift tracking to avoid converging to local maxima and to enable track recovery after full occlusions. A background-weighted histogram was used to distinguish the target from the background and from other targets.

All of the approaches that are summarized above extend CAMShift or Mean Shift in the one or the other way. However, none of them took adequate measures to enhance tracking in all aspects. Multiple objects cannot be tracked in [8, 1, 10, 3, 15, 4, 13], severe appearance changes (caused by changes of illumination or perspective) are not addressed in [1, 10, 3, 16], re-detection after track losses is not supported in [8, 1, 10, 3, 4, 13], and occlusions are not resolved in [8, 10, 3, 15, 4, 13]. We address all of these points and explain a fast GPU implementation.

Besides extensions of CAMShift or Mean Shift trackers, several efficient realizations exist that benefit from parallel processing of SIMD architectures.

Allen et al [2] used the SSE vector instructions of the Pen-

tium microprocessor to implement standard Mean Shift. They report that 1-2 ms per frame was achieved on average for object tracking on a 1.6GHz Pentium-4 CPU (this did not include the 3-4 ms per-frame overhead, such as video decoding and displaying). Unfortunately, details on how this performance was reached (i.e., video, histogram, and search window resolutions and applied color spaces) have not been provided. For search window sizes ranging from 66x66 up to 640x480, our GPU implementation of standard CAMShift requires 0.5 ms (average and without per-frame overhead) and our implementation of extended CAMShift requires 1.4 ms (average and without per-frame overhead). The per-frame overhead is 1.3 ms in our cases and includes RGB2HSV conversion and image upload from RAM to VRAM for an image resolution of 640x480.

Recently, Li et al. [12] proposed a parallel Mean Shift tracking implementation on the GPU using CUDA. They apply k-means clustering for partitioning the color space of a tracked object in order to represent it with a minimum number of histogram bins. No measures have been taken to improve the quality or robustness of the Mean Shift. They report that their GPU implementation of standard Mean Shift tracking is by a factor of 2-3 faster than the corresponding CPU implementation. Compared to the OpenCV implementation of standard CAMShift, our GPU implementation of standard CAMShift is by a factor of 8.8 (average and including per-frame overhead) faster. Our extended GPU CAMShift implementation still outperforms OpenCV CAMShift by a factor of 6.0 (average and including per-frame overhead). Yet, these extensions make CAMShift tracking significantly more robust.

3. Extended CAMShift Tracking

In this section, we describe several low-cost 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 (which 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 off-line. Additional reference histograms can be added and deleted at any time for updating the accumulated histogram.

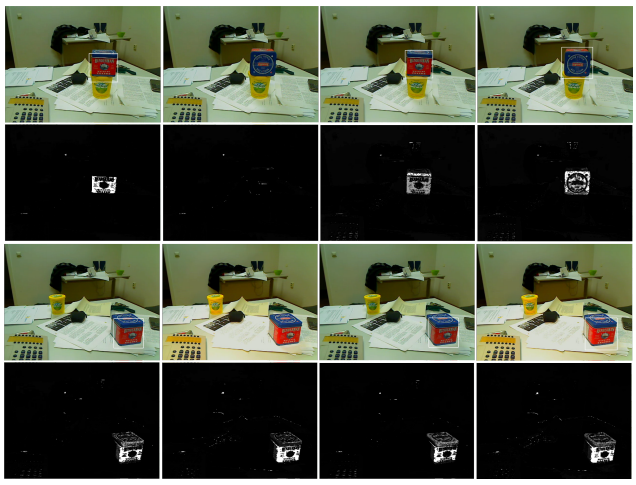

Figure 1. Left two columns: If an object is tracked with only one reference histogram, CAMShift fails if appearance changes strongly (e.g., object is rotated (top rows) or the lighting conditions change (bottom rows)). Right two columns: If two references histograms are accumulated for the two corresponding 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.

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 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.

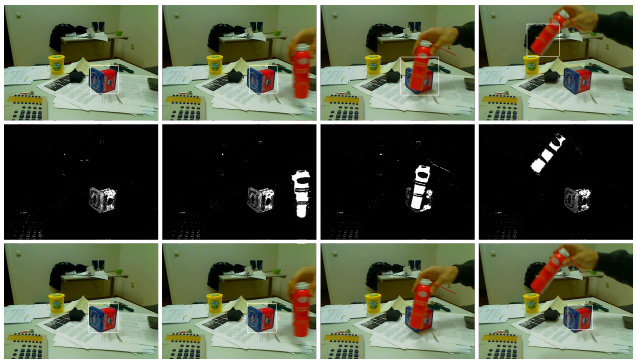

Figure 2. Upper row: Tracking with occlusion fails for standard CAMShift if objects have similar colors (the box is tracked while the bottle is used for interference). Lower row: Matching histograms to monitor all objects supports partial or full occlusion. The back-projected probability distributions 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 continuous 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 multidimensional histograms in various color spaces (e.g., HSV, YUV, or RGB) for computing back-projected probability distribu-

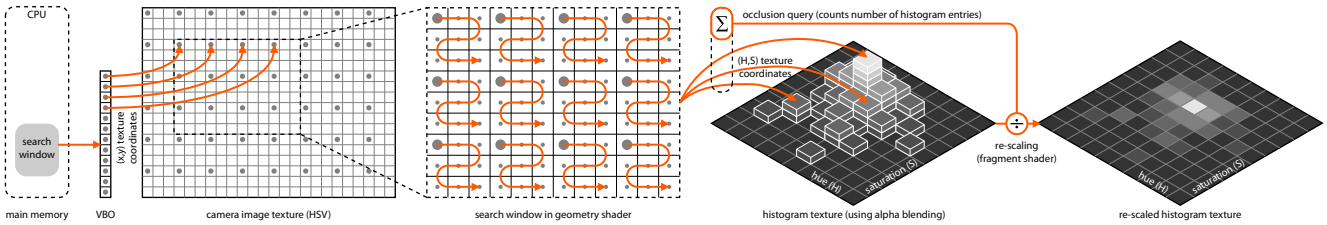

Figure 3. Histogram generation on GPU.

tions. Therefore, a fast implementation of histogram generation is essential for a fast CAMShift tracker.

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 [11]. We render only every ith vertex (in both dimensions) instead of 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 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, HSV and 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, HSV, 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 exists, 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 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 RG-BRG 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 is the values of the five image moments for the entire search window that are stored in the RGBRG channels of the two 1x1 textures which are 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.

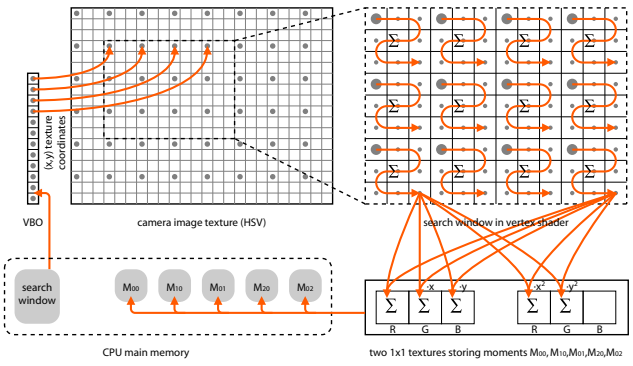

Figure 4. Computation of image moments on GPU.

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 whose individual texture coordinates point to the corresponding bin 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.

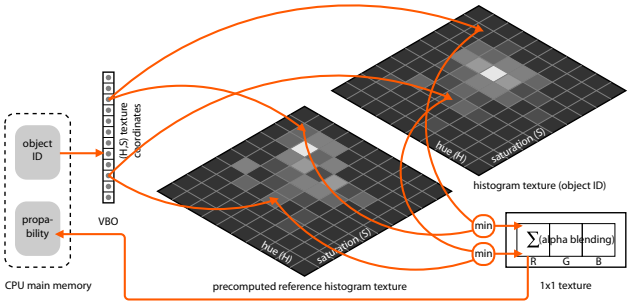

Figure 5. Histogram intersection on GPU.

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 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 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 current mid-level devices.
The color coding in figure 7 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

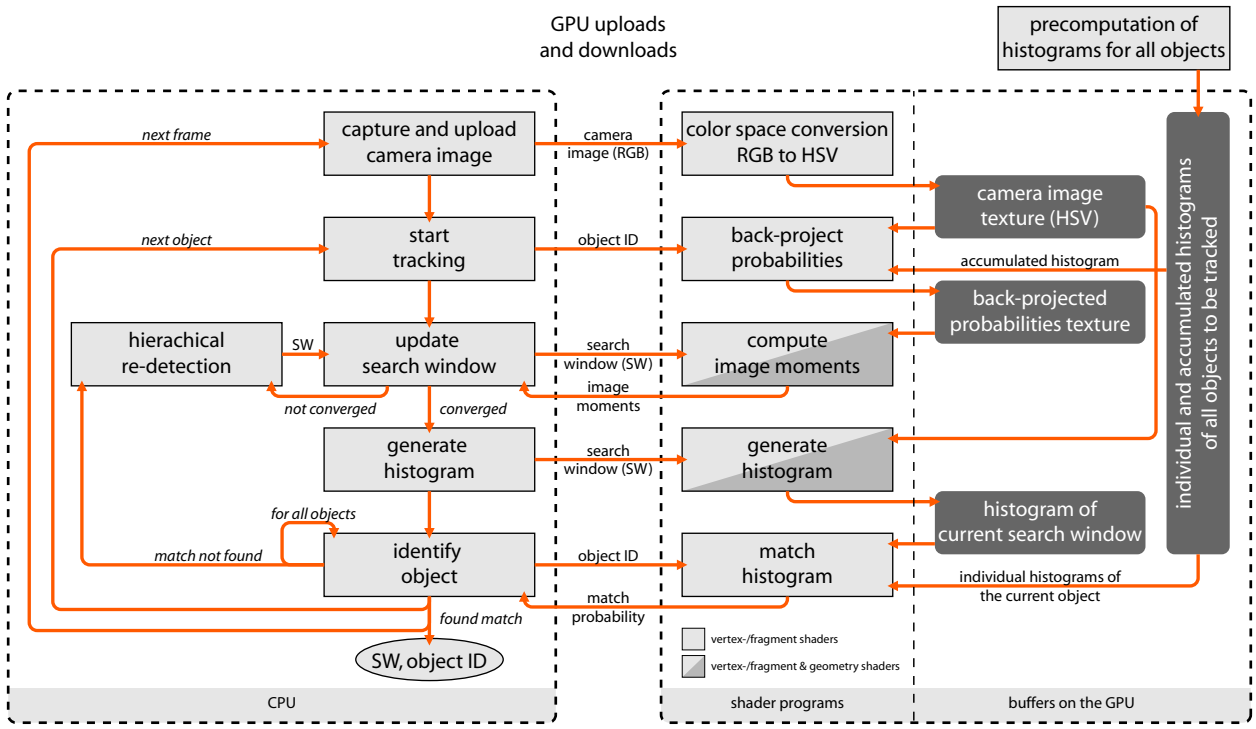

Figure 6. Overview of CPU and GPU steps of extended CAMShift with up- and downloads. Computation of image moments, histogram generation, and histogram intersection are illustrated in more detail in figures 3 - 5.

CAMShift (GPU)

object size / search window size		66x66	75x75	110x110	150x150	210x210	254x254	300x300	360x360	425x425	500x480	525x480	640x480
upload (image resolution: 640x480)		1,0500	1,0500	1,0500	1,0500	1,0500	1,0500	1,0500	1,0500	1,0500	1,0500	1,0500	1,0500
RGB2HSV (image resolution: 640x480)		0,2525	0,2525	0,2525	0,2525	0,2525	0,2525	0,2525	0,2525	0,2525	0,2525	0,2525	0,2525
probability back projection (image resolution: 640x480)		0,1773	0,1773	0,1773	0,1773	0,1773	0,1773	0,1773	0,1773	0,1773	0,1773	0,1773	0,1773
image moments		0,2129	0,2214	0,2204	0,2432	0,2653	0,2619	0,3077	0,3423	0,3750	0,4403	0,4474	0,4528
object identification (histogram size: 32x6)	histogram generation	0,2066	0,2118	0,2497	0,2925	0,3947	0,5049	0,5942	0,6849	0,8222	1,0227	1,0471	1,2251
	histogram normalization	0,1000	0,1000	0,1000	0,1000	0,1000	0,1000	0,1000	0,1000	0,1000	0,1000	0,1000	0,1000
	histogram match	0,1614	0,1614	0,1614	0,1614	0,1614	0,1614	0,1614	0,1614	0,1614	0,1614	0,1614	0,1614
total (extended CAMShift with RGB2HSV and image upload)		**2,1606**	**2,1743**	**2,2112**	**2,2769**	**2,4011**	**2,5079**	**2,6430**	**2,7683**	**2,9383**	**3,2042**	**3,2356**	**3,4190**
total (extended CAMShift without RGB2HSV and image upload)		**0,8582**	**0,8718**	**0,9087**	**0,9744**	**1,0986**	**1,2054**	**1,3405**	**1,4658**	**1,6359**	**1,9017**	**1,9332**	**2,1165**
total (standard CAMShift with RGB2HSV and image upload)		**1,6926**	**1,701158**	**1,700173**	**1,72297**	**1,74506**	**1,741598**	**1,787431**	**1,821992**	**1,854775**	**1,920057**	**1,927174**	**1,932555**
total (standard CAMShift without RGB2HSV and image upload)		**0,3901**	**0,3987**	**0,3977**	**0,4205**	**0,4426**	**0,4391**	**0,4850**	**0,5195**	**0,5523**	**0,6176**	**0,6247**	**0,6301**

CAMShift (CPU)

object size / search window size	66x66	75x75	110x110	150x150	210x210	254x254	300x300	360x360	425x425	500x480	525x480	640x480
RGB2HSV (image resolution: 640x480)	12,8759	12,8759	12,8759	12,8759	12,8759	12,8759	12,8759	12,8759	12,8759	12,8759	12,8759	12,8759
probability back projection	1,6248	1,6248	1,6248	1,6248	1,6248	1,6248	1,6248	1,6248	1,6248	1,6248	1,6248	1,6248
image moments	0,0591	0,0704	0,1300	0,2224	0,4239	0,5958	0,8002	2,2153	3,0853	3,0551	3,1392	2,5212
total (standard CAMShift with RGB2HSV)	**14,5598**	**14,5711**	**14,6308**	**14,7231**	**14,9246**	**15,0965**	**15,3010**	**16,7160**	**17,5861**	**17,5558**	**17,6399**	**17,0219**
total (standard CAMShift without RGB2HSV)	**1,6839**	**1,6952**	**1,7549**	**1,8472**	**2,0487**	**2,2206**	**2,4251**	**3,8401**	**4,7102**	**4,6799**	**4,7640**	**4,1460**

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

Figure 7. Timings (in ms) of different components of extended/standard CAMShift on GPU and standard CAMShift on CPU.

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 our re-implementation of standard CAMShift without extensions. On the CPU, we timed the OpenCV 1.1 implementation of standard CAMShift. We timed all three cases with and without per-frame overhead (i.e., color space conversion and image upload), since this is independent from the number of objects that are being tracked.

For standard CAMShift, our GPU implementation outperforms the CPU implementation of OpenCV by average factors of 6.1 and 8.8 (without and with per-frame overhead). For extended CAMShift, the processing time for probability back-projection, image moments computation and object identification increases linearly with each additional object. The latter also increases linearly for each additional reference histogram. It outperforms OpenCV CAMShift by average factors of 2.2 and 6.0 (without and with per-frame overhead), and makes CAMShift tracking significantly more robust. The accompanying video compares the robustness of standard CAMShift and extended CAMShift under different conditions.

With 1.3 ms for a video resolution of 640x480, the per-

frame overhead (image upload from RAM to VRAM and RGB2HSV color space conversion) is the bottleneck of our system. It becomes less relevant, the more objects are being tracked. Thus, our system is particularly suited for tracking a large number of objects (an example with 8 targets is shown in the accompanying video). The image upload can be significantly reduced by using modern capture cards, such as Nvidia's new Quadro SDI system.

All our measurements above have been compared with the OpenCV 1.1 implementation of standard CAMShift. We have also timed the standard CAMShift implementation of OpenCV 2.0 and measured an average speedup factor of 1.5 when compared with the OpenCV 1.1 implementation. Both of our GPU implementations are still significantly faster.

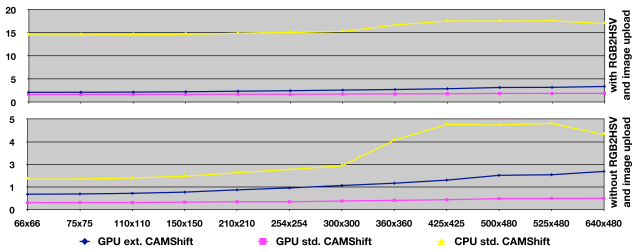

Figure 8. Overall performance (in ms) of CPU/GPU standard/extended CAMShift with (top) and without (bottom) processing per-frame overhead for a 640x480 image resolution, and with respect to an increasing search window size.

6. Summary and Future Work

We have presented several low-cost extensions that make CAMShift more robust, and we have explained how they can be implemented efficiently on the GPU. By reviewing previous implementations, we believe that ours is currently the fastest. We have also taken several measures to support multi-object tracking, handle appearance changes, resolve full and partial occlusions, and re-detect objects after their tracks were lost.

Computing the object orientation from image moments as initially suggested in [5] was easy to integrate in our implementation, but proofed to be very unstable and imprecise. We therefore omitted this. To determine the orientation reliably and fast belongs to our future tasks. Furthermore, a better load balancing between GPU and CPU processing needs to be investigated – in particular with respect to a combination of multi-core GPUs and CPUs.

We believe that robust CAMShift tracking of many (10-30) targets at common video rates (25-30 fps) can also be beneficial for visual game-interaction.

References

[1] A. Adam, E. Rivlin, and I. Shimshoni. Robust fragments-based tracking using the integral histogram. In *CVPR (1)*, pages 798–805. IEEE Computer Society, 2006.

[2] J. G. Allen and J. S. Jin. Mean shift object tracking for a SIMD computer. In *ICITA '05: Proceedings of the Third International Conference on Information Technology and Applications (ICITA'05) Volume 2*, pages 692–697, Washington, DC, USA, 2005. IEEE Computer Society.

[3] J. G. Allen, R. Y. D. Xu, and J. S. Jin. Object tracking using camshift algorithm and multiple quantized feature spaces. In *VIP '05: Proceedings of the Pan-Sydney area workshop on Visual information processing*, pages 3–7, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[4] A. S. K. Bin and L. Y. Kang. Face detection and tracking utilizing enhanced camshift model. *International Journal of Innovative Computing, Information and Control*, 2006.

[5] G. R. Bradski. Computer vision face tracking for use in a perceptual user interface, 1998.

[6] J. Campbell. Shaogang gong, stephen j. mckenna, alexandra psarrou, dynamic vision: From images to face recognition - review. *Artif. Intell. Rev.*, 14(6):619–621, 2000.

[7] Y. Cheng. Mean shift, mode seeking, and clustering. *IEEE Trans. Pattern Anal. Mach. Intell.*, 17(8):790–799, 1995.

[8] R. T. Collins and Y. Liu. On-line selection of discriminative tracking features. In *ICCV*, pages 346–352. IEEE Computer Society, 2003.

[9] D. Comaniciu and P. Meer. Robust analysis of feature spaces: color image segmentation. In *CVPR*, pages 750–. IEEE Computer Society, 1997.

[10] F. Dadgostar, A. Sarrafzadeh, and S. P. Overmyer. Face tracking using mean-shift algorithm: A fuzzy approach for boundary detection. In J. Tao, T. Tan, and R. W. Picard, editors, *ACII*, volume 3784 of *Lecture Notes in Computer Science*, pages 56–63. Springer, 2005.

[11] F. Diard. *GPU Gems 3*, chapter 41, pages 895–897. Addison-Wesley Professional, 2007.

[12] P. Li and L. Xiao. Mean shift parallel tracking on gpu. In H. Arajo, A. M. Mendona, A. J. Pinho, and M. I. Torres, editors, *IbPRIA*, volume 5524 of *Lecture Notes in Computer Science*, pages 120–127. Springer, 2009.

[13] R. Stolkin, I. Florescu, and G. Kamberov. An adaptive background model for camshift tracking with a moving camera. In *Proc. of the 6th International Conference on Advances in Pattern Recognition*, 2007.

[14] M. J. Swain and D. H. Ballard. Color indexing. *International Journal of Computer Vision*, 7(1):11–32, 1991.

[15] G. Xiang and X. Wang. Real-time follow-up head tracking in dynamic complex environments. *Journal of Shanghai Jiaotong University (Science)*, 14(5):593–599, 2009.

[16] C. Zhang, Y. Qiao, E. Fallon, and C. Xu. An improved camshift algorithm for target tracking in video surveillance. In *Proceedings of 9th. IT and T Conference*, number 12, 2009.