Tracking and Clustering Salient Features in Image Sequences

Werner Bailer, Hannes Fassold, Felix Lee
JOANNEUM RESEARCH, DIGITAL - Institute of Information and Communication Technologies
8010 Graz, Austria
{firstname.lastname}@joanneum.at

Jakub Rosner
Silesian University of Technology
Faculty of Data Mining
44-100 Gliwice, Poland
jakub.rosner@joanneum.at

Abstract—**Many applications in media production need information about moving objects in the scene, e.g. insertion of computer-generated objects, association of sound sources to these objects or visualization of object trajectories in broadcasting. We present a GPU accelerated approach for detecting and tracking salient features in image sequences and we propose an algorithm for clustering the obtained feature point trajectories in order to obtain a motion segmentation of the set of feature trajectories. Evaluation results for both the tracking and clustering steps are presented. Finally we discuss the application of the proposed approach for associating audio sources to objects to support audio rendering for virtual sets.**

Keywords-tracking; motion segmentation; clustering; GPU.

I. INTRODUCTION

Segmentation is a fundamental problem in computer vision. In image sequences, motion is an important segmentation cue that allows determining regions that are subject to different motion. Typically these regions are the background, which is only subject to the camera motion, and a number of moving foreground objects. In media production there is a wide range of applications that need this information, e.g. insertion of computer-generated objects, association of sound sources to these objects or visualization of object trajectories in broadcasting.

A. Motivation

Increasingly information about moving objects in the scene needs to be available in or near real-time to support live use cases or on-set pre-visualization. Thus algorithms must be capable of providing sufficiently precise information in real-time, ideally on cost-efficient hardware.

We present a GPU accelerated approach for detecting and tracking salient features in image sequences. Then we propose an algorithm for clustering the trajectories by a 4-parameter motion model to obtain a motion segmentation of the set of feature trajectories. The approach supports moving camera and the motion model supports translation, rotation and scaling of the objects. As we want to support a non-translatory motion model, the main problem for trajectory clustering is the fact that the feature used for clustering is hidden: Only the sequence of point locations is observable, while the sequence of motion parameters that generating the trajectories is hidden. Measuring the similarity between the point trajectories thus consists of two tasks: (i) estimating sequences of motion parameters from the point trajectories and (ii) clustering the trajectories by similarity of the motion parameter sequences.

The rest of this paper is organized as follows. Section I.B discusses related work on motion trajectory clustering and GPU accelerated implementations of feature point tracking. Section II reviews the selected tracking algorithm, discusses its implementation on the GPU and presents results. The motion trajectory clustering algorithm is described in Section III. Section IV discusses applications of the algorithm and Section V concludes the paper.

B. Related Work

This section discusses the approaches for segmenting sets of point trajectories by motion that have been proposed in literature. The *factorization* method has originally been proposed to solve the shape from motion problem and decompose a matrix containing point trajectories [17]. The approach has later been extended to the case of multiple moving objects [4], which includes the problem of finding subsets of trajectories that belong to the same object.

The approach proposed in [10] works directly on the measurement data and is based on the same principle as earlier works on the factorization method. As discussed in literature, the approach is very sensitive to noise. Reliable results can only be achieved with very precise tracking data. An advantage is the simplicity of the method and the fact that the number of different motions need not be known in advance. In this simple formulation, the approach will only cluster by translational motion. There is also no straight forward way to use the approach for trajectories that are only defined for a part of the temporal range.

The approach in [10] actually uses the angle between the trajectory segments as similarity measure between the trajectories. Using the normalised scalar product significantly decreases the performance of the algorithm, and in the presence of moderate noise the results become unusable.

The authors of [3] propose a trajectory clustering method based on the *longest common subsequence* (LCSS) distance known from string matching. This approach is limited to clustering by translatory similarity. In [14] a clustering method based on the expectation-maximisation (EM) approach and *Gaussian mixture models* is proposed. Also this method considers only similarity by translatory motion.

In [16] an approach for clustering sequences with Hidden *Markov models* is proposed. The approach is based on deciding which of the sequences has been created with which

model. Thus the parameters of the different Markov models must be sufficiently different to allow a separation. A problem is that the calculations cannot be done in a recursive manner, if the hidden states are not discrete. This would lead to a product of Gaussian probability distribution functions, which cannot be described analytically in a simple form. In addition, the number of clusters needs to be determined separately, which is a hard problem on its own and the number of clusters might change over time.

In contrast to most other approaches the authors of [8] propose an algorithm that handles trajectories with *different life spans* and clusters them in a time window by affine motion. The approach uses the J-linkage method for clustering.

The clustering approaches need reliable trajectories as input and in order to track smaller objects reliably, a high number of trajectories needs to be estimated efficiently. For feature point tracking, approaches which use the tremendous processing power of Graphics Processing Units (GPUs) have proven to give a significant speedup compared to CPU implementations. Most of them implement the KLT algorithm [12][18], which is well known and has competitive performance in terms of feature point quality and runtime. In [15], a GPU implementation (using OpenGL and Cg [1] shading language) is reported to track approximately thousand features in real-time on a 1024x768 video. To save computation time, the detection of new feature points is done only every fifth frame, which is not sufficient for image sequences with fast camera motion. The approach in [13] modifies the feature point detection step of the KLT algorithm, in order to circumvent some steps (e.g. the sorting of the feature points according to their cornerness), which are hard to parallelize. In [21] a variant of the KLT algorithm is proposed which compensates for varying camera gain automatically by incorporating it into the estimation process. Detection of new features is done every 10 frames, and some levels of the image pyramid are skipped for faster processing.

II. GPU Accelerated Feature Point Tracking

In the following, we describe the approach we use for detection and tracking feature points in the image sequence and its implementation on the GPU. We employ the widely used KLT algorithm [12][18]. It first detects a sparse set of feature points with sufficient texture in the current image and adds them to the already existing ones. Afterwards, the positions of feature points in the subsequent image are estimated by minimizing the dissimilarity measured as sum of squared differences (SSD) in a window around the positions. in Section II.A we give a short overview about GPU programming and CUDA, the general purpose GPU programming environment introduced by NVIDIA and then we describe the detection and tracking step of the KLT algorithm. Subsection II.D shows how to port them efficiently to CUDA and compares the CPU and GPU implementations in terms of quality and runtime. The GPU accelerated feature tracking algorithm is described in more detail in [7].

A. GPU Programming & CUDA

In recent years, the implementation of computer vision algorithms on the GPU has received increasing interest. For algorithms which are sufficiently parallelizable, significant speedups (e.g. an order of magnitude) have been reported. In this work, we focus on CUDA, which is supported by NVIDIA GPUs and is currently the most mature programming environment. A CUDA-capable GPU has the following properties:

- Massively parallel architecture. Current GPUs have a couple of hundred processing cores which are grouped into multiprocessors. Algorithms should be massively parallelizable to take advantage of all cores.
- Global memory. The random-access GPU memory can be read and written by all threads, but has a high latency.
- Various other memory types. Every multiprocessor owns 16KB of shared memory, which is a very fast on-chip read-write cache and has to be explicitly managed by the algorithm developer. Texture memory offers a read-only, cached access to a one- or two-dimensional buffer.

A CUDA program is typically composed of a control routine, which calls a couple of CUDA kernels. A kernel is similar to a function, but is executed on the GPU in parallel by a large number of threads (typically thousands).

B. Feature Point Detection

In the feature point detection step, new features are found based on a cornerness measure in an image and added to the already existing feature points. We denote I as the current image and J as the subsequent image in the sequence. We write the spatial gradient of I as $\nabla I = \partial I / \partial(x, y)$ and a small (e.g. 5x5) rectangular region centered at a point p as $W(p)$. Note that p can have non-integer coordinates, the values are then calculated using bilinear interpolation.

As a measure of the cornerness of a feature centered at a pixel p, first the structure matrix G defined by

$$G = \sum_{x \in W(p)} \nabla I(x) \cdot \nabla I(x)^T \qquad (1)$$

is calculated. Its eigenvalues λ_1 and λ_2 (which are ≥ 0 as the matrix is positive-semidefinite) can be used to derive information about the image structure in the neighborhood of p. The smaller eigenvalue $\lambda = \min(\lambda_1, \lambda_2)$ is employed in the KLT algorithm as cornerness measure of region W. The detection of new points can be summarised in the following steps:

- Cornerness calculation. For each pixel in the image I, its structure matrix G and cornerness is calculated. Furthermore, the maximum cornerness λ_{max} which occurs in the image is calculated.
- Thresholding and non-maxima suppression. From the image pixels, only the pixels which have a cornerness λ larger than a certain percentage (e.g.

[1] http://developer.nvidia.com/object/cg_toolkit.html

- 5%) of λ_{max} are kept. Afterwards, non-maxima suppression within the neighborhood of the remaining points is performed.
- Minimum-distance enforcement. From the remaining points new ones are added, starting with the points with the highest cornerness values. To avoid points concentrated in some area of the image, newly added points must have a specific minimum distance d (e.g. 5 or 10 pixels) to points already added.

C. Feature Point Tracking

In the feature point tracking step, we want to calculate for each feature point p in image I its corresponding motion vector v so that its tracked position in image J is $p + v$. As 'goodness' criterion of v we take the SSD error function

$$\varepsilon(v) = \sum_{x \in W(p)} (J(x+v) - I(x))^2 . \quad (2)$$

It measures the image intensity deviation between a neighborhood of the feature point position in I and its potential position in J. Setting the first derivative of $\varepsilon(v)$ to zero and approximating $J(x+v)$ by its first order Taylor expansion around $v = 0$ results in a better estimate v_1. By repeating this multiple times, we obtain an iterative update scheme for v which is summarized in Algorithm 1. Typically, v converges after 4 - 7 iterations, for static feature points the convergence is even faster.

Note that the Taylor expansion around zero is only valid for small motion vectors v. In order to handle large motion (e.g. fast moving objects), we apply the scheme in a multi-resolution representation. For this, we first generate an image pyramid, by repeatedly convolving the image with a small Gaussian kernel and subsampling it. Then we apply the scheme in each pyramid level (for all feature points), going from the highest levels to lower ones. In each level, the properly scaled motion vector v of the previous level is used as the initial guess in the current level. In this way, the estimated motion vector v is refined in each level.

1. Set initial motion vector $v_1 = (0,0)^T$
2. Spatial image gradient $\nabla I = \partial I / \partial(x,y)$
3. Calc. structure matrix $G = \sum_{x \in W(p)} \nabla I(x) \cdot \nabla I(x)^T$
4. for k = 1 to *maxIter*
   a) Image difference $\eta(x) = I(x) - J(x + v_k)$
   b) Calculate mismatch vector $b = \sum_{x \in W(p)} \eta(x) \cdot \nabla I(x)$
   c) Calculate updated motion $v_{k+1} = v_k + G^{-1} b$
   d) if $\| v_{k+1} - v_k \| < eps$ then stop (converged)
5. Report final motion vector v

Algorithm 1. Pseudo-code of the calculation of the motion vector v for a given feature point *p*. *W(p)* is a window centred at *p*.

D. GPU Implementation and Results

In the following, we discuss some aspects of the GPU implementation, using the CUDA programming environment by NVIDIA. We compare the GPU implementation with a highly optimized CPU implementation from the OpenCV[2] library.

1) Feature point detection. The cornerness calculation is easy to parallelize, as it is done independently for each pixel. In contrast, the parallel calculation of the maximum cornerness λ_{max} is more involved, as many threads would have to modify the same variable simultaneously, leading to many read-write hazards. This issue can be solved efficiently by using the CUDPP[3] library. In the thresholding and non-maxima suppression steps, we mark features that do not meet the respective conditions as invalid. The last step, the minimum distance enforcement, cannot be implemented efficiently on the GPU as it is inherently serial. So we first transfer all features, which have not been marked as invalid, back to the CPU memory. Once again we use the CUDPP library for filtering out the invalid features before transferring. For the minimum-distance enforcement, we employ an alternative algorithm (using a mask image) which is more efficient than the OpenCV method, if many feature points (> 1000) are to be handled. It adds a point only if its position is not already marked in the mask image. If the point is added, a circular area around it with radius d is marked in the mask image, indicating that area as occupied.

2) Feature point tracking. In contrast to feature point detection, the whole feature point tracking (for a specific pyramid level) is done only by one CUDA kernel to get the most benefit of shared memory. Each feature point corresponds to exactly one GPU thread. To take advantage of the hardware bilinear interpolation, the image pyramids are stored as texture images. A complicated yet important issue for performance is the minimization of shared memory and register usage of one thread. Using less of these memory types allows a better utilization of the multiprocessors. A significant amount of experimentation has also been done to determine the optimum number of threads per thread block.

3) Results. We compare the quality and speed of the GPU implementation with respect to an optimized CPU implementation from the OpenCV library. The runtime measurements were done on a 2.4 GHz Intel Xeon Quad-Core, equipped with a NVIDIA Geforce GX280 GPU. A quality comparison of the feature points delivered by both implementations shows us that most of the detected and tracked feature points are similar. Minor differences are

[2] The OpenCV library (http://opencv.willowgarage.com/) uses SSE and OpenMP internally.

[3] CUDPP (http://code.google.com/p/cudpp/) provides some useful functions for efficient compaction, sorting etc.

likely due to slightly different floating-point implementation in the GPU and CPU. For Full HD resolution material, the GPU implementation is roughly an order of magnitude faster than the CPU implementation (see [7]). This allows us to do feature point tracking with several thousand points on Full HD material in real-time.

III. MOTION TRAJECTORY CLUSTERING

The input for motion trajectory clustering is the result of tracking feature points throughout an image sequence. For each frame in which a point is present, the image coordinates of the point are given. Points may appear and disappear at any time.

The task is to identify subsets of tracked points that are subject to similar motion according to a motion model. The reason for clustering trajectories instead of motion parameters in single frames is to achieve a more stable cluster structure over time. The problem thus leads to a problem of clustering trajectories.

Given is a set of trajectories p_i, $i = 1..N$, and an image sequence of M frames. The point trajectories are defined for $k = 1..M$: $p_{i,k} = 0$ (not present in frame k) or $p_{i,k} = (u_{i,k}, v_{i,k})^T$, where $u_{i,k} = x_{i,k} - x_{i,k\text{-}1}$ and $v_{i,k} = y_{i,k} - y_{i,k\text{-}1}$ are the displacements of the points. There are three main difficulties in clustering feature trajectories:

- Temporal extent of trajectories. Not all trajectories may exist throughout the whole time window used for clustering. Trajectories may end when points exit the scene, cannot be reliably tracked any more or are occluded. New trajectories start when new reliable points are detected.
- Unknown number of clusters. As the number of moving objects in the scene is not known, the number of clusters is also unknown. The number of resulting clusters of the previous time window serves as a hint, but objects may appear or disappear.
- Clustering by hidden feature. The feature trajectories just contain the x- and y-displacement of the moving objects. If clustering is performed using a more complex model than a simple translatory one, the feature used for clustering is hidden. A function must be defined which expresses how well a trajectory matches a set of motion parameters.

An additional requirement is the capability to perform clustering on a time window without having all data for a shot available. This enables the application of the algorithm to incrementally process data and make it applicable in near online scenarios.

To deal with these issues, our algorithm uses an Expectation-Maximization approach to solve the clustering problem, consisting of the following steps:

- Estimate a motion parameter sequence q for a set of trajectories.
- Assign the trajectories to the motion parameter sequence.
- Cluster motion parameter sequences.

After initialization, these steps are performed iteratively until the cluster structure for the time window stabilizes.

A. Approach

The problem of estimating the sequence of parameters is an optimization problem with the following constraints: The sequence of predicted point coordinates (based in the parameters) shall approximate the point measurements as well as possible. This corresponds to the emission probabilities of an HMM, modeled by a normal distribution with the predicted point coordinates as μ and the measurement error as σ. The predicted point measurement depends only on the previous point measurement and on the current estimated parameter tuple (this expresses the Markovian property of the point coordinate sequence).

The parameter tuple depends on the previous parameter tuple, and a smoothness constraint is enforced between subsequent parameters. This corresponds to the transition probabilities of an HMM. If a first order smoothness constraint is enforced, this corresponds to a normal distribution with the previous parameter tuple as μ and the standard deviation σ. If a second order smoothness constraint is enforced, this corresponds to a normal distribution with the linearly predicted parameter tuple as μ and the standard deviation σ.

In the following the steps of our algorithm are described in detail. An overview is shown in Figure 2.

1) Initialization. The clustering step for a time window is initialised from the results of the previous one. The resulting motion parameter sequences are kept and those trajectories, that still exist, are assigned to parameter sets according to the previous assignment. All unassigned trajectories (i.e. those newly appearing in this time window) will be used to create additional sequences of motion parameters. Initial sets of unassigned trajectories are formed by grouping between 3 and 5 spatially adjacent trajectories, as it is likely that they belong to the same moving object. Motion parameter sequences are estimated for the trajectory sets as described below and those with a low total matching error are selected as initial trajectory sets.

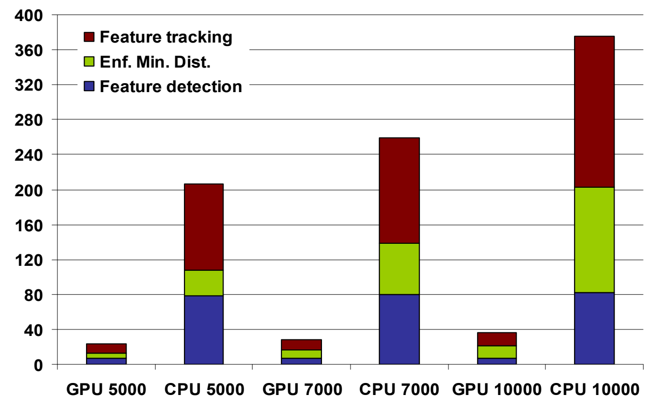

Figure 1. Comparison of the runtime of the GPU and CPU implementation of the KLT algorithm for Full HD material and different maximum numbers of feature points (5000, 7000, 10000). The runtime is given in milliseconds.

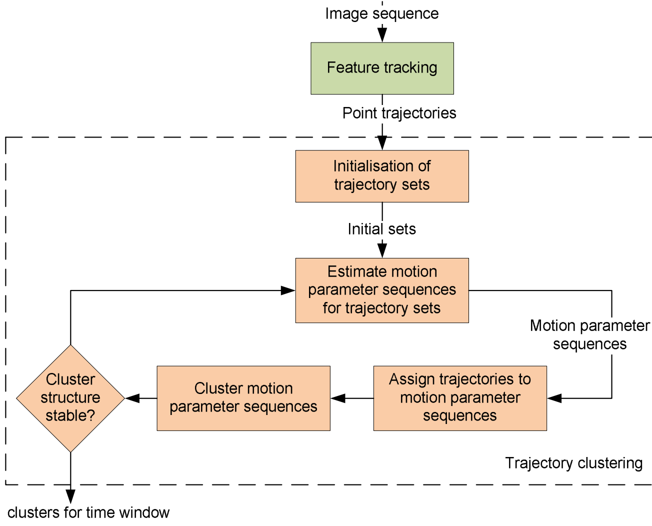

Figure 2. Overview of the feature point trajectory clustering algorithm.

2) Estimation of motion parameters. The estimation of the sequence of motion parameters for a set of trajectories is formulated as an optimisation problem. The function to be minimised is an error function containing the matching error between the measured trajectory and the prediction of the trajectory and a smoothness constraint on the motion parameter sequence. The matching error is calculated as the sum of the Euclidian distances between the measured point positions and those predicted using the current motion parameter sequence estimates over the frames of the time window. In the current implementation, a four parameter motion model with x and y translation, scaling and rotation is used. The smoothness constraint uses a penalty function for differences between subsequent motion parameter tuples (currently the difference of curvature is used), weighted to tolerate small deviations. This models the fact that in natural scenes neither camera nor object motion change abruptly but always gradually.

In detail, the error function to be optimised for a single trajectory is given as:

$$\varepsilon_{traj}(t) = \mathrm{dist}(x(t); q(t)x(t-1)), \quad (3)$$

where dist is a distance function between the two point locations. The error for the time window is given as

$$\varepsilon_{traj}(T) = \sum_{t=1}^{T} \varepsilon_{traj}(t). \quad (4)$$

For a set of trajectories P, the function to be minimized is

$$\varepsilon_{trajset}(T, P) = \sum_{P} \sum_{t=1}^{T} \left(x(t) - q(t)x(t-1)\right)^2. \quad (5)$$

The smoothness constraint for the parameters is given as

$$\varepsilon_s(t) = \omega(q(t); q(t-1)), \quad (6)$$

where ω is a penalty function for differences between the parameter tuples (e.g. difference of curvature), possibly weighted to tolerate small deviations.

The smoothness constraint for the parameter sequence is given as

$$\varepsilon_s(T) = \sum_{t=1}^{T} \varepsilon_s(t), \quad (7)$$

and the total error function for the optimization problem is

$$\varepsilon(T, P) = \varepsilon_s(T) + \varepsilon_{trajset}(T, P). \quad (8)$$

To solve the optimization problem, Newton's method with simple dogleg (using the implementation of [11]) is used. It has been found to perform better than other tested algorithms (e.g. BFGS in the implementation of [20]) on synthetic data, and at least as good as others on noisy data. The solution of the optimisation problem is a motion parameter sequence that approximates the set of trajectories.

3) Assignment of trajectories to motion parameter sequences. After motion parameter sequences have been (re-)estimated, the trajectories are assigned based on the error between the measured trajectory and the predicted trajectory by using the estimated motion parameter sequence. The error for one trajectory is calculated using the matching error function described above. Each trajectory is assigned to the best matching motion parameter sequence in terms of this error function.

4) Clustering of motion parameter sequences. Clustering is performed on the motion parameter sequences, which has two main advantages over directly clustering trajectories: The cluster criterion is now a visible feature, i.e. the motion parameters itself, and the number of data items to be clustered is significantly less than the number of trajectories. Hierarchical clustering using the single linkage algorithm [5] is performed on the motion parameter sequences. A distance function between motion parameter sequences has been used, which penalises differences in scale and rotation higher than differences in translation. The distance values of the most similar clusters are also used as the terminating condition for the trajectory clustering step.

B. Results

To evaluate the clustering algorithm, we use a subset of the Hopkins155 data set [19]. We use the 25 real-world video sequences in the data set and do not use the derived variants with only one or two of the motions present. To make the results comparable we use the trajectories provided with the data set. In contrast to typical tracking results, all of these trajectories exist throughout the whole sequence.

Our algorithm estimates an appropriate number of clusters and thus creates more clusters than there are in the ground truth in some cases. In some of the sequences objects are stationary in some frames of the video. If this is the case in the beginning of the video, our algorithm merges them and splits them only later.

comparison method	window size	mean MCR	median MCR
max. overlapping	video	0.46	0.41
all overlapping	video	0.05	0.02
group to target nr.	video	0.29	0.13
max. overlapping	20	0.44	0.40
group to target nr.	20	0.29	0.13

Table 1. Mean and median misclassification rates of our trajectory clustering algorithm on 25 videos of the Hopkins155 data set.

As proposed in [19] we use the misclassification rate (MCR), i.e. the fraction of trajectories assigned to the wring cluster, as evaluation measure. Table 1 shows the results of this evaluation. For the first three experiments, we set the clustering time window to the duration of the video. The first one gives the results when we only consider the largest cluster that overlaps with a ground truth cluster as correct. If we count all trajectories of clusters that lie within the ground truth cluster as correct, the results are significantly better. This shows that the main issue here is over-segmentation, and only few trajectories are misclassified. We have also further grouped the result clusters by similarity of the estimated parameter sequences until the target cluster number given by the ground truth is reached. The results lie between the results for the other experiments. In the last two experiments we used a window size of 20 frames, shifted by 2 frames. The reported results are for the frame in the middle of the sequence. All other parameters of the algorithm were identical for all sequences in all experiments.

IV. Workflow Integration

A. Application to Audio Rendering

Surround sound has been for a long time an integral part of modern multimedia experiences, using an ever increasing number of channels (e.g. 22.2 [9]). The approach used in the 2020 3D media project 0[6] is not bound to a fixed number of channels, but uses ambisonic measurements of sound sources and a model of the room geometry to reproduce more realistic spatial sounds. In virtual scenes or presence of computer-generated objects this requires obtaining information about the sound source positions. In order to attach a sound source to a moving object, the trajectory of the respective object has to be determined. Doing this manually is a time consuming process. Thus our tracking and motion clustering algorithm is used for this purpose.

The motion trajectory clustering method has some limitations due to the nature of the problem. This includes separation of (temporarily) stationary objects and distinct objects that move similarly for some time. An example of such a problem is shown in Figure 3. This video frame shows a foreground building in the middle and two buildings behind it. During a camera pan, the front building induces motion trajectories which are different than those induced by the two buildings in the background. Most of the point trajectories belonging to the foreground building are correctly grouped into the same cluster. But point trajectories belonging to the two buildings in the background are inseparable and are assigned to the same cluster. In general, two objects can only be correctly separated, when sufficient motion differences between them are present. Objects that move only slowly or stop for some time will be merged together, even with the static background.

The clustering result gives us a coarse separation of object included in a scene. Often only marginal human intervention leads to satisfying results. We have thus implemented an interactive tool that allows controlling the automatic tracking and clustering process and enables an operator to modify the clustering results. Using the tool, the workflow is as follows:

- After the tracking process, select and define a single trajectory or a bundle of trajectories as sound source path.
- Check whether the path is correct in the following frames by scrolling the video forward.
- Optionally, correct the path by adding or removing point trajectories.
- Restart the clustering process to extend the corrected path.
- Repeat steps 2 to 4 if necessary for the time window in which the object is present.

For further processing of the scene, the tool exports the object bounding boxes and trajectories in COLLADA [2] format. This is an XML based standard for 3D scenes and animations that can be imported into a wide range of 3D modeling and animation tools.

B. Functionality of the Interactive Tool

The interactive tool wraps the point tracking and clustering algorithms into one single application. This includes a user interface which allows us to load a video, launch the tracking and clustering algorithms and save the clustering results.

Figure 3. An example of the clustering results. Points of the foreground building in the middle marked with purple bounding rectangle are grouped into one cluster, while the points of the two background buildings are grouped into other cluster and marked with pink bounding rectangle.

The main focus of the tool is on enabling convenient user interaction for changing and correcting the cluster structure. In particular, the user shall be able to move points between clusters, split or merge clusters. The ability to scroll the videos forward and backward frame by frame simplifies the visual verification of the results. The video frames are overlaid with tracked points and point trajectories. The tool supports the user in sorting out points such as the crossings of two object contours that disturb the performance of the motion trajectory clustering. Once the cluster structure has been modified, the automatic clustering algorithm can be resumed in subsequent frames using the modifications made by the user as additional input.

A screen shot of the user interface is shown in Figure 4. The application provides the following workflow support functionalities.

1) *Segment wise analysis of the video.* The tracking process can be started and stopped at any video time point. Clustering analysis can be started once the segment is tracked. The analysis can be continued from last stop.

2) *Changing tracking and clustering parameters.* The number of points being tracked, size of time window for motion parameter estimation, etc. can be changed while the analysis is paused. The analysis will be resumed with the new set of parameters. This allows a progressive tuning of algorithm parameters by human intervention.

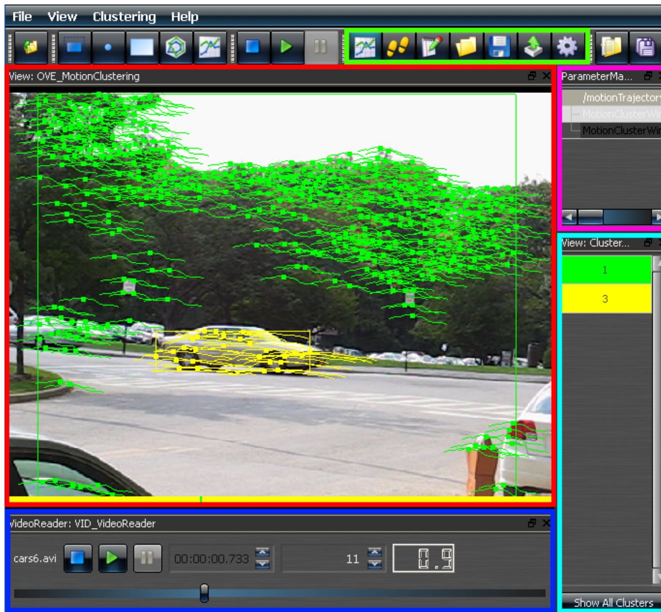

Figure 4: A screen shot of the graphical user interface of the interactive tool. The area for showing video, the playing control buttons, the tracking and clustering buttons, the parameter modification control and the cluster labels are marked with red, blue, green, magenta and cyan rectangles, respectively.

3) *Implicit object tracking.* Although point tracking has been applied, assignment of trajectories to an object, that does not have different motion than other object or the background, is not done by the algorithm. This feature is particularly interesting when tracking of object parts is necessary. For example, the audio rendering may need to attach different sound sources to distinctive parts of a large object.

4) *Export of the cluster centers.* Exporting the paths of the cluster centers in the COLLADA file format allows the data exchange between our interactive tool and other third party applications, e.g. the open source 3D modelling tool Blender[4].

V. Conclusion and Future Work

In this work we have implemented a state of the art feature point tracking algorithm on the GPU using CUDA, achieving a speedup factor that enables real-time tracking of thousands of feature point on Full HD resolution input video. We have proposed an algorithm for clustering the obtained feature point trajectories by a 4-parameter motion model, capable of handling incomplete trajectories. An interactive tool is provided to enable user intervention for cases in which the motion information does not allow to discriminate objects as needed. We have demonstrated the use of the information to support a novel audio rendering method, but each of the components has a number of other applications in media production. The clustering step is currently still implemented on the CPU. Porting it to the GPU will further increase the interactivity of the user interface.

Acknowledgment

The authors would like to thank their partners in the “2020 3D Media” for the collaboration, especially Toni Mateos and Pau Arumí. The research described in this paper has been partially supported by the European Commission’s 7th Framework Programme under the contract FP7-215475, “2020 3D Media” (http://www.20203dmedia.eu/). The work of Jakub Rosner has been partially supported by the European Social Fund.

References

[1] P. Arumí, D. Garcia, T. Mateos, A. Garriga and J. Durany. Real-time 3D audio for digital cinema. ASA Conference ACOUSTICS'08, Paris, 2008.

[2] M. Barnes and E. Levy Finch (eds.). COLLADA – Digital Asset Schema Specification. Release 1.5.0, April 2008.

[3] D. Buzan, S. Sclaroff, and G. Kollios. Extraction and Clustering of Motion Trajectories in Video. In Proceedings of the 17th International Conference on Pattern Recognition, pp. 521-524, 2004.

[4] J. Costeira and T. Kanade. A multi-body factorization method for motion analysis. CMU-CS-TR-94-220, 1994.

[5] R. O. Duda, P. E. Hart and D. G. Stork. Pattern Classification. 2nd ed. Wiley-Interscience, 2001.

[4] http://www.blender.org/

[6] J. Durany, A. Garriga and T. Mateos. Towards a realistic ray tracing for room acoustics. ASA Conference ACOUSTICS'08, Paris, 2008.

[7] H. Fassold, J. Rosner, P. Schallauer, W. Bailer. Realtime KLT Feature Point Tracking for High Definition Video, GravisMa workshop, Plzen, 2009.

[8] M. Fradet, P. Robert, and P. Pérez. Clustering Point Trajectories with Various Life-Spans, 6th European Conf. on Visual Media Production, London, UK, Nov. 2009.

[9] K. Hamasaki, S. Komiyama, K. Hiyama, H. Okubo. 5.1 and 22.2 Multichannel Sound Productions Using an Integrated Surround Sound Panning System, AES Convention, 2004.

[10] K. Kanatani. Motion Segmentation by Subspace Separation and Model Selection. In Proc. 8th International Conference on Computer Vision, Vancouver, CA, pp. 586–591, 2001.

[11] C. T. Kelley. Iterative Methods for Optimization. URL: http://www4.ncsu.edu/eos/users/c/ctkelley/www/matlab_darts.html

[12] B.D. Lucas, T. Kanade. An Iterative Image Registration Technique with an Application to Stereo Vision. Joint Conference on Artificial Intelligence, pp. 674-679, 1981.

[13] J. Ohmer, N. Redding. GPU-Accelerated KLT Tracking with Monte-Carlo-Based Feature Reselection. Digital Image Computing: Techniques and Applications, 2008.

[14] S. Pachoud, E. Maggio, and A. Cavallaro. Grouping motion trajectories. Proc. IEEE International Conference on Acoustics, Speech and Signal Processing, pp. 1477-1480, 2009.

[15] S. Sinha, J. Frahm, M. Pollefeys, Y. Genc. GPU-Based Video Feature Tracking and Matching. Technical Report 06-012, Department of Computer Science, UNC Chapel Hill, 2006.

[16] P. Smyth. Clustering Sequences with Hidden Markov Models. In Advances in Neural Information Processing 9, MIT Press, 1997.

[17] C. Tomasi and T. Kanade. Shape and motion from image streams under orthography: a factorization method. Int. J. Comput. Vision, 9(2):137–154, 1992.

[18] C. Tomasi and T. Kanade. Detection and Tracking of Point Features. Technical Report CMU-CS-91-132, Carnegie Mellon University, 1991.

[19] R. Tron and R. Vidal. A Benchmark for the Comparison of 3-D Motion Segmentation Algorithms," IEEE Conference on Computer Vision and Pattern Recognition, Jun. 2007.

[20] S. Ulbrich. Optimieriung 3. MATLAB Routinen und Beispiele. URL: http://www-m1.ma.tum.de/m1/ personen/sulbrich/opt3/codes.html

[21] C. Zach, D. Gallup, J.M. Frahm. Fast gain-adaptive KLT tracking on the GPU. CVGPU Workshop on Computer Vision on GPUs, 2008.