Real-time online camera synchronization for volume carving on GPU

Torkel Andreas Haufmann, André Rigland Brodtkorb, Asbjørn Berge
Anna Kim
SINTEF, Dept. Appl. Math.
P.O. Box 124, Blindern,
N-0314 Oslo, Norway
{Torkel.Haufmann,Andre.Brodtkorb,Asbjorn.Berge,Anna.Kim}@sintef.no

This is the accepted manuscript to be published in The 10th IEEE International Conference on Advanced Video and Signal-Based Surveillance 2013 (AVSS 2013)[1]

Abstract

Volume carving is a well-known technique for reconstructing a 3D scene from a set of 2D images, using features detected in individual cameras, and camera parameters. Spatial calibration of the cameras is well understood, but the resulting carved volume is very sensitive to temporal offsets between the cameras. Automatic synchronization between the cameras is therefore desirable. In this paper, we present a highly efficient implementation of volume carving and synchronization on a heterogeneous system fitted with commodity GPUs using an improved version of the algorithm in [1].

An online, real-time synchronization system is described and evaluated on surveillance video of an indoor scene. Improvements to the state of the art CPU-based algorithms are described.

1 Background and Related Work

Modern surveillance setups are comprised of networks of IP cameras, which in high security areas are commonly positioned with overlapping fields of view. A natural extension to standard video surveillance analysis (i.e., motion detection and tracking) is to extract 3D information by triangulation. The typical approach for this is backprojection of 2D foreground segmentation in each camera view (e.g., [2]). This is commonly referred to as volume carving, and is a generic tool in many computer vision applications. In the surveillance setting it is used for improving occlusion resistance in tracking, as well as supporting human pose estimation.

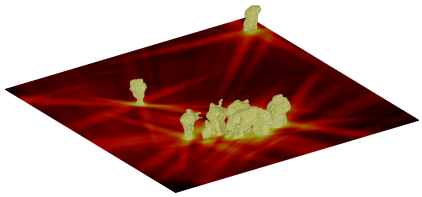

Figure 1: Illustration of voxel carving in a synchronized situation — 3D information has been recovered from the 2D images observed by each camera.

One major problem when performing volume carving over time in such a network is that the cameras fall out of sync. This can happen for a number of reasons, e.g., congested networks leading to dropped frames, varying time spent encoding frames in the IP cameras, et cetera. It is clear that as the video streams lose synchronization, backprojection of detection results becomes worthless. Our scene is a rectangular area, each side being a few meters long, and considering the spe speed of human motion and our video rate (i.e., 20-25 Hz) in relation to the scene, the cameras should stay synchronized to within approximately 2 frames, or artefacts such as projection of different arm positions will likely occur.

1.1 Related work

Many approaches to camera synchronization have been studied. For example, multiple cameras can be synchronized using hardware or audio triggers as proposed in [3] for markerless motion capture. Such approaches generally require installation (e.g., cabling) for each camera in the network, and can be limited by the resolution or the accuracy of the recorded audio signal, or hardware failure. In [4], the authors formalized the general synchronization problem for stationary cameras by tracking moving objects. In [5], cameras are assumed to be stationary, observing the same scene and not connected to any external clock. The proposed approach matches two or more sequences by evaluating the correlation among the temporal features of the sequences. While the possibility of realtime operation was not discussed, synchronization also fails when persons or objects enter or exit the scene.

A more general heterogenous camera network was considered in [6]. However, the complexity of the proposed solution can be a major hindrance for operating online. For our application, heterogeneity of the camera network is not under consideration, and neither is moving cameras. Furthermore, since the video streams are collected through IP networks, they may be affected by different delays (and jitters). Continuous and *online* synchronization of more than two static cameras observing the same scene is therefore our main objective. To this end, we extend the idea presented in [1], which suits our requirement well. There, a camera pair is synchronized using temporal co-occurrences computed from epipolar lines. Details of this algorithm and our improvements are further explained in Section 2.2.

In terms of volume carving, much emphasis has been devoted to fast implementa-

tions in the recent years, which in practice often means GPU implementations. In [7], two GPU based visual hull computation algorithms were proposed, where the resulting 3D reconstruction method is able to operate at 30 fps with 16 cameras and 4 PCs. In [8], a single GPU implementation is used for a realtime 3D video conferencing application, where a new cache strategy was developed for parallel calculation of visual hulls and a factor of five speedup was reported. A CUDA-accelerated real-time 3D modeling system is presented in [9], which offers accurate visual hull reconstruction with texture mapping, at a frame rate of 20fps. The authors of [10] presented both algorithmic improvement in voxel based visual hull reconstruction, and implementation speedup using FPGA and GPU that offered both realtime (30 fps) and high definition. Other existing near realtime implementations using GPUs are also reported therein.

In the field of visual surveillance, research is primarily concerned with the volume carving having sufficient fidelity to enable precise track assignment [2] and rudimentary human pose estimation [11]. The implementation presented in the following delivers synchronized, high resolution voxel spaces at faster than real-time speeds.

1.2 Our Contribution

In the ADABTS project the ADABTS Demonstrator System (ADS) was developed, showcasing state-of-the-art surveillance algorithms on heterogeneous platforms. In this paper we describe a small part of this system, specifically the part pertaining to volume carving, which also requires camera synchronization. Our novel contributions mainly relate to the synchronization algorithm used.

2 3D Reconstruction in a Camera Network

ADS uses a proprietary segmentation algorithm that we will not discuss further in this paper. Here we describe the 3D reconstruction part of the system in detail.

2.1 Volume Carving

Volume carving is a well-known technique for reconstructing a 3D scene from a set of 2D images. For the sake of completeness we give a brief overview of the method in the following. For pseudo-code, see Algorithm 1.

As mentioned we assume that each of n cameras has a currently observed frame with a foreground segmentation available, defined as a function $F_j : \mathbb{R}^2 \to [0, 1]$ with the support of F_j a bounded rectangular area corresponding to the camera image. The value F_j takes in any position corresponds to the confidence that a given pixel is a foreground pixel. We also assume the existence of transformation functions $T_j : \mathbb{R}^3 \to \mathbb{R}^2$ that compute the 2D position in the camera plane for a given 3D position.

The functions T_j are not, in practice, simple perspective projections, as IP cameras used in our setup distort the observed scene. Our voxel carving grid is in actual 3D space and we must correct for this distortion. Our approach is based on OpenCV's, which uses Brown's model (see for example [12]).

Algorithm 1 Computing the voxel grid values

Input: Number of cameras, n. T_j is the transformation into camera plane j. $\mathbf{p}$ is the center of the current voxel in world space. F_j is the frame as seen in camera j.

function CARVE
  conf $\leftarrow 0$
  for $j = 0$ to n **do**
    $\mathbf{r} \leftarrow T_j(\mathbf{p})$
    conf $\leftarrow$ conf $+ F_j(\mathbf{r})$
  conf $\leftarrow$ conf$/n$
  return conf

Output: The confidence for the voxel having center position $\mathbf{p}$.

In effect, each 3D point is projected into the $z = 1$ plane, corrected for distortion, and then projected back into world space. The obtained position in 3D world space is then perspective projected into the camera plane. In our experiments we found that while this approach is fine locally, some points that are far from the observed scene can be massively "undistorted" and erroneously wind up in the visible area after the approach is applied. Computing the proper points to exclude from the undistortion process is a rather complex process, but a simple heuristic that works well is filtering points after the projection into the plane $z = 1$, leaving out all points not contained in $[-1.1, 1.1] \times [-1.1, 1.1]$.

With all of the above given, we assume that the cameras are observing the same scene; specifically, that there is a convex volume contained in the intersection of all the cameras' fields of view (disregarding occlusions). In this volume we position a grid of voxels, which for our purposes are simply little cubes containing confidence values. For each frame received, we iterate through the voxels, computing the average foreground confidence of each voxel's center over all cameras, and using this average as the confidence in the voxel. For further processing, a thresholding can be employed so that low-confidence voxels do not cause undue noise in the 3D estimation.

2.2 Synchronization

Synchronized cameras are a necessity for high-quality volume carving. ADS receives approximate timestamps from IP cameras, but these only suffice for a rough synchronization. Relatively few frames of drift can make a significant difference in quality, and so we need an algorithm for performing fine-grained synchronization — which is the hard case for us. In [1] an approach to synchronization is suggested, building on epipolar geometry. The authors of [1] discuss camera pairs, so we designate one camera — camera 0 — as the "base" camera, and synchronize all the other cameras with this camera. We will briefly restate the algorithm of [1] as it applies to our situation, then move on to discussing our alterations. We describe the algorithm as it applies to a single camera pair; the network synchronization built on top should be obvious.

Algorithm 2 Frame-wise update for pair $(0, j)$

Input: The segmentations F_0 and F_j, the transformation T_i from world space into the 2D camera plane of camera i, and the sample point sets.

for each epipolar plane **do**
  $c_{\text{base}} \leftarrow 0$
  for each sample point $\mathbf{v}$ in camera 0 **do**
    $\mathbf{p} \leftarrow T_0(\mathbf{v})$.
    $c \leftarrow F_0(\mathbf{p})$.
    $c_{\text{base}} \leftarrow c_{\text{base}} + c$.
  Store c_{base}.
  $c_{\text{other}} \leftarrow 0$
  for each sample point $\mathbf{v}$ in camera j **do**
    $\mathbf{p} \leftarrow T_j(\mathbf{v})$.
    $c \leftarrow F_j(\mathbf{p})$.
    $c_{\text{other}} \leftarrow c_{\text{other}} + c$.
  Store c_{other}.

For our purposes an *epipolar plane* is the plane determined by two camera positions and a third point, which point we refer to as the *epipolar focus*. An *epipolar line* is the intersection of an epipolar plane with the image plane for one of its two defining cameras. By construction there is a one-to-one correspondence between epipolar lines in the two cameras, each of which correspond to the same plane. The basic assumption in [1] is that by this construction motion somewhere along the line in one camera is likely accompanied by motion along the corresponding line in the other camera.

Hence, for each line we define the *line signal* at a given time to be the summed deviations from the background over a set of sample points along the line. We have a foreground model from our segmentation algorithm, so we reuse this as our measure of deviation from the background. We aggregate the line signals of each line over time obtaining the temporal signal matrices $\mathbb{S}, \mathbb{S}'$ for the base and "other" camera, respectively, where

$$\mathbb{S}_{r,t} = \sum_{p \in l_r} F_j(p). \tag{1}$$

The frame-wise algorithm is summarized in Algorithm 2.

Next, after a set interval, the most likely drift between the two cameras is computed using the temporal signals. The set of lines is first modified by subtracting the average of each line from each signal stored for it, then filtering out certain lines (We will return to the line filtering below). The formula for the most likely drift Δt^* is then

$$\begin{aligned} \Delta t^* &= \arg \max_{-c \le \Delta t \le c} P(\Delta t) g(\Delta t), \\ g(\Delta t) &= e^{-\sum_{r \in \mathcal{L}(t)} \sum_{t=t_0-k}^{t_0} \frac{\left(\mathbb{S}_{r,t} - \mathbb{S}'_{r,t+\Delta t}\right)^2}{2\sigma^2}}. \end{aligned} \tag{2}$$

In this equation c is the maximal drift we even consider in each direction, σ is a parameter expressing the expected variance in line signal intensity, and k is a parameter

constraining how many frames' line signals we include in the computation. We are abusing notation and denoting by $\mathbb{S}$ and $\mathbb{S}'$ the temporal signals after subtraction of averages. $\mathcal{L}(t)$ is the set of lines that are active, i.e., non-filtered. $P(t)$ is a (Gaussian) prior used to express our pre-existing assumptions about drift. See Algorithm 3 for pseudo-code. The different algorithm parameters are discussed in Section 4.

The algorithm implemented in [1] tracks drift in the system. Our system is intended to operate online, so instead of merely tracking the accumulated drift in the cameras, we fix it whenever we have enough confidence in a result. This is accomplished by determining which cameras are the farthest ahead of the pack and having them wait for the others. Based on our experience in applying the algorithm to our use cases, we have also made some algorithmic changes. They are described in the following sections.

Line filtering: In [1], line filtering is implemented by considering the motion gradient of a line at a single time step, applying an assumption of Gaussian white noise and then using a threshold for probability of actual motion. ADS is heterogeneous and the motion information exists on the GPU, so we would neet to store frame-wise samples and download them to the host in order to do the same. This would complicate the code quite a bit, and other options are preferrable.

Hence we have opted for a simpler approach, in which we only consider motion information in the base camera; by assumption, the other camera should have similar results, and we ignore it for the sake of simplicity. Assuming we are comparing the set $\mathcal{T}$ of timesteps we compute for each line l_r the value

$$d_r = \max_{t \in \mathcal{T}} \mathbb{S}_{r,t} - \min_{t \in \mathcal{T}} \mathbb{S}_{r,t}. \tag{3}$$

We can then filter all lines using a threshold τ_{motion}. A line l_r is assumed to be active in the current synchronization only if $d_r > \tau_{\text{motion}}$. We only perform periodic synchronization computations if at least half the line set is active.

Epipolar foci: In [1] the existence of a set of epipolar lines is assumed. For our system to be truly adaptive it needs to create a useful set on initialization, based on camera calibration data. A bad strategy for doing this automatically would carry the risk of bunching the epipolar lines together, in effect filtering out information from our synchronization. We have implemented the following simple heuristic for distributing the lines.

We assume that the setup is observing the $z = 0$ plane in world coordinates, with a point $\mathbf{p}$ in the center of the observed space. Given the locations $\mathbf{b}$ of the base camera and $\mathbf{o}$ of the other camera we wish to generate a set of epipolar lines. We compute the two vectors $\mathbf{v}_b = \mathbf{b} - \mathbf{p}$ and $\mathbf{v}_o = \mathbf{o} - \mathbf{p}$ and their projections into the plane $z = 0$, $\hat{\mathbf{v}}_b$ and $\hat{\mathbf{v}}_o$, and their average $\mathbf{d} = (1/2)(\hat{\mathbf{v}}_b + \hat{\mathbf{v}}_o)$.

If $\hat{\mathbf{v}}_b$ and $\hat{\mathbf{v}}_o$ are almost parallel, $\|\mathbf{d}\|$ will be very small. In such cases, we instead rotate $\hat{\mathbf{v}}_b$ by $\pi/2$ to obtain (after normalization) $\hat{\mathbf{d}}$. Otherwise we set $\hat{\mathbf{d}} = \mathbf{d}/\|\mathbf{d}\|$. In either case, we now distribute our epipolar foci uniformly along the segment of the line $\mathbf{o} + t\hat{\mathbf{d}}, t \in \mathbb{R}$ which is contained in the observed space. We distribute the epipolar foci in standard world space, so after distortion correction some may not be visible — these are removed from the working set of epipolar foci upon software initialization.

Drift estimate smoothing: In [1], a drift estimate is tracked over time. We would like to correct this drift in an online manner, however, and acting on a single time point's estimate makes the algorithm susceptible to an unfortunate choice of periodic synchronization time. To counter this problem, we compute a moving weighted mean d_{avg} of drift estimates, and only take action if $|d_{\text{avg}}| \geq 1$. We admit a computed drift to the "window set" of drifts over which the average is computed if it has a confidence above some threshold τ_{conf}. In order to maintain responsiveness this approach requires a more frequent periodic synchronization than is used in [1], but since most of our work is performed on the GPU we have an abundance of computing power available. See Algorithm 3 for a pseudo-code description of the periodic synchronization algorithm as we have implemented it.

Algorithm 3 Periodic synchronization for pair $(0, j)$

Input: The temporal signals $\mathbb{S}$ and $\mathbb{S}'$ and the parameters $\sigma, \sigma_{\text{prior}}, \tau_{\text{motion}}, \tau_{\text{conf}}$.

Subtract line averages.
Filter inactive lines using τ_{motion}.
Compute Δt^* using Eq. 2.
if confidence of Δt^* exceeds τ_{conf} **then**
    Add Δt^* to window set, discarding oldest.

3 Implementation on the GPU

ADS is a heterogeneous system performing the bulk of work on the GPU. Both algorithms described in the previous section are suitable for GPU parallelization. ADS is written in C++ on the CPU side and uses CUDA for the GPU code.

Our test results are obtained running the ADS software on a system with an Intel i7 and an Nvidia GeForce GTX580. The volume carving is entirely implemented on the GPU, while only the frame-wise synchronization is implemented in CUDA; Algorithm 3 is better suited for the CPU.

3.1 Volume Carving

Volume carving on the GPU is straight-forward (and as mentioned, previously studied, see for instance [7, 8, 9, 10]). The multiple texture fetches involved in processing each voxel means that the algorithm is completely I/O-bound, so there is no significant overhead in correcting for distortion. In our system we launch a block grid consisting of blocks with dimension 16×16 and let each thread iterate through an entire column in the voxel grid's z direction. Our experience is that this setup provides a high-quality volume carving given synchronized video.

Parameter	Description
σ	Expected variation in line signals
σ_{prior}	Standard deviation of prior
τ_{motion}	Motion threshold
τ_{conf}	Confidence threshold

Table 1: The synchronization parameters.

3.2 Synchronization

Algorithm 2 computations amount to running block reductions, iterating over the lines in each camera. We have implemented this using a single-block kernel with 512 sample points, to facilitate concurrent kernel execution with other parts of ADS. The algorithm is entirely I/O-bound. The temporal signals are copied to the host before running Algorithm 3.

All experiments have been run with a total frame history of 250 frames, 50 frames between each periodic synchronization and a sliding average window consisting of 5 computed drifts. In each periodic synchronization we consider a maximal drift of 10 (corresponding to c in Eq. 2) in each direction, and compare a total of 221 frames (corresponding to k in Eq. 2). We note that this means we consider relatively small drifts, time-wise; as described before correcting small drifts is the main purpose of our algorithm in ADS. We distribute up to 100 lines in the visible area.

Beyond these settings, we have experimented with various settings for the parameters in Table 1. Each captures a slightly different aspect of the synchronizer's behaviour, and they all need to be tuned in relation to each other.

4 Experiments and Results

For obtaining good results in our system, the cameras must be correctly calibrated. The previously published data sets we considered lacked the features we desired, but in the ADABTS project a video sequence was recorded for which the cameras were correctly calibrated and the setup known, and we used this data set to collect our results. The data set has 4 video streams, which are known to be synchronized to within 3 frames or so. We test using roughly 10000 frames of video. The first two thousand frames are relatively empty, involving perhaps one moving person at a time, while between frames 2000 and 5000 smaller groups of people congregate. From roughly frame 5000 there is a great number of people moving around in the scene.

We measure system performance in two situations: One in which we simply run the recorded data set as-is, and one in which we artificially let camera 1 skip 1 frame every 1000 frames. The first case is intended to verify that the synchronizer does no harm, while the second is intended to verify that it is able to correct the type of drift for which it is made. Since we are synchronizing all cameras against camera 0 we will plot their frame count (in terms of the video streams on disk) versus the frame count in

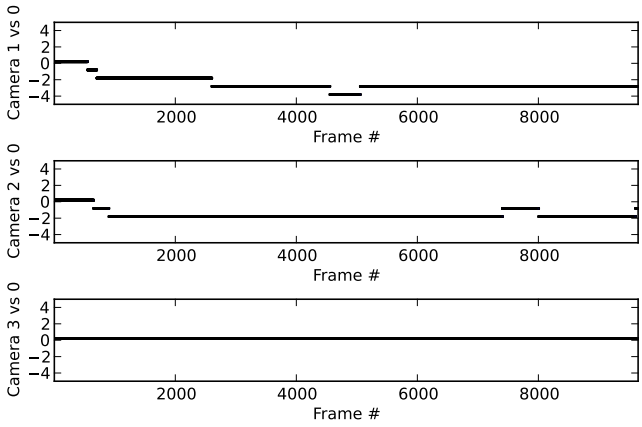

Figure 2: Frame-count difference per camera when video streams are run with no changes A few frames are to be expected given our video quality, and we see that the synchronizer appears to stabilize on an estimate of drift.

camera 0.

We run both test cases with the same parameter set we found to be the best fit for our test data. We have set $\sigma = 0.3$, which governs the confidence intensification on the best found drift, and $\sigma_{\text{prior}} = 1.75$, corresponding to our base assumption that a drift much greater than 1 or 2 frames should be less likely to come under consideration. We found that for our setup, $\tau_{\text{conf}} = 10^{-40}$ worked well, which essentially means turning off the threshold for window set admittance. $\tau_{\text{motion}} = 0.001$ filters out only those lines that detect no motion at all, which in our data set turned out to happen quite a bit.

Results: The results of our two tests can be seen in Figures 2-3. In the latter we have left out the cameras 2 and 3, as the plots are the same as in Figure 2. We see that in both instances the synchronizer keeps the cameras within the approximate accuracy known for our dataset (in fact Figure 2 indicates that our dataset is slightly out of sync). The unmodified version appears to converge to an estimate for the drift, while the other struggles slightly more due to being interrupted regularly. After 10000 frames, however, the version with frame drops is still within 1 frame of the version without, indicating that the synchronization algorithm makes the system resilient to this kind of drift.

Performance: In Figure 4 we show system performance with increasing number of videos — a voxel grid size of $256 \times 256 \times 64$ is used. At this size we see that we can handle up to about 5 or 6 streams at greater-than-real-time speeds. In none of our experiments did the synchronization algorithm appreciably impact runtime; all the significant work involves video processing and volume carving.

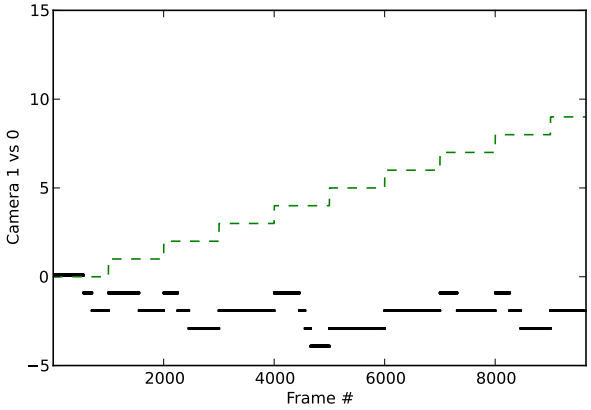

Figure 3: Frame-count difference between camera 1 and 0 when camera 1 is dropping 1 frame every 1000 frames – the dashed line shows what would happen without synchronization. Here there is no stabilization, but given the continual frame-dropping this is to be expected.

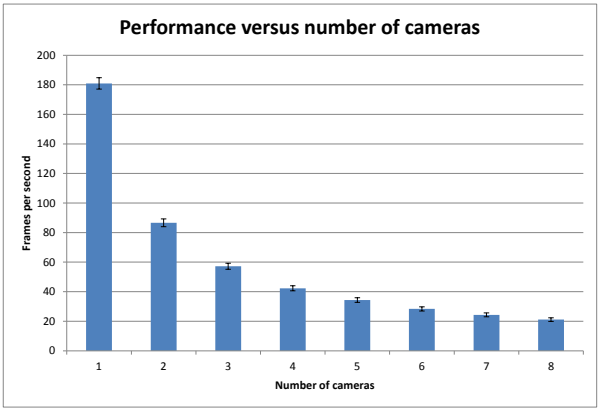

Figure 4: Scaling of system performance with number of cameras.

5 Conclusions and Future Work

The GPU approach in ADS allows us to perform volume carving and synchronization online. The algorithm as presented here, however, is quite sensitive to parameter tuning, and future work in obtaining a more robust set-up, or a better means of automatic tuning, would be of great interest for any practical use.

Acknowledgments

This research has received funding from the European Community Seventh Framework Programme under grant agreement number 218197, the ADABTS project.

References

[1] Pundik, D., Moses, Y.: Video synchronization using temporal signals from epipolar lines. Computer Vision–ECCV 2010 (2010) 15–28

[2] Liem, M., Gavrila, D.: Multi-person localization and track assignment in overlapping camera views. Pattern Recognition (2011) 173–183

[3] Hasler, N., Rosenhahn, B., Thormahlen, T., Wand, M., Gall, J., Seidel, H.P.: Markerless motion capture with unsynchronized moving cameras. In: Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on, IEEE (2009) 224–231

[4] Whitehead, A., Laganiere, R., Bose, P.: Temporal synchronization of video sequences in theory and in practice. In: Application of Computer Vision, 2005. WACV/MOTIONS'05 Volume 1. Seventh IEEE Workshops on. Volume 2., IEEE (2005) 132–137

[5] Ushizaki, M., Okatani, T., Deguchi, K.: Video synchronization based on co-occurrence of appearance changes in video sequences. In: Pattern Recognition, 2006. ICPR 2006. 18th International Conference on. Volume 3., IEEE 71–74

[6] Imre, E., Guillemaut, J.Y., Hilton, A.: Through-the-lens multi-camera synchronisation and frame-drop detection for 3d reconstruction. In: 3D Imaging, Modeling, Processing, Visualization and Transmission (3DIMPVT), 2012 Second International Conference on, IEEE (2012) 395–402

[7] Ladikos, A., Benhimane, S., Navab, N.: Efficient visual hull computation for real-time 3d reconstruction using cuda. In: Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on, IEEE (2008) 1–8

[8] Waizenegger, W., Feldmann, I., Eisert, P., Kauff, P.: Parallel high resolution real-time visual hull on gpu. In: Image Processing (ICIP), 2009 16th IEEE International Conference on, IEEE (2009) 4301–4304

[9] Shujun, Z., Cong, W., Xuqiang, S., Wei, W.: Dreamworld: Cuda-accelerated real-time 3d modeling system. In: Virtual Environments, Human-Computer Interfaces and Measurements Systems, 2009. VECIMS'09. IEEE International Conference on, IEEE (2009) 168–173

[10] Perez, J.M., Aledo, P.G., Sanchez, P.P.: Real-time voxel-based visual hull reconstruction. Microprocessors and Microsystems (2012)

[11] Hofmann, M., Gavrila, D.: Multi-view 3d human pose estimation in complex environment. International journal of computer vision **96**(1) (2012) 103–124

[12] Brown, D.C.: Close-range camera calibration. PHOTOGRAMMETRIC ENGINEERING **37**(8) (1971) 855–866