From multiple views to textured 3D meshes: a GPU-powered approach

K. Tzevanidis, X. Zabulis, T. Sarmis, P. Koutlemanis,
N. Kyriazis, and A. Argyros

Institute of Computer Science (ICS)
Foundation for Research and Technology - Hellas (FORTH)
N.Plastira 100, Vassilika Vouton, GR 700 13
Heraklion, Crete, GREECE
{ktzevani, zabulis, argyros}@ics.forth.gr

Abstract. We present work on exploiting modern graphics hardware towards the real-time production of a textured 3D mesh representation of a scene observed by a multicamera system. The employed computational infrastructure consists of a network of four PC workstations each of which is connected to a pair of cameras. One of the PCs is equipped with a GPU that is used for parallel computations. The result of the processing is a list of texture mapped triangles representing the reconstructed surfaces. In contrast to previous works, the entire processing pipeline (foreground segmentation, 3D reconstruction, 3D mesh computation, 3D mesh smoothing and texture mapping) has been implemented on the GPU. Experimental results demonstrate that an accurate, high resolution, texture-mapped 3D reconstruction of a scene observed by eight cameras is achievable in real time.

1 Introduction

The goal of this work is the design and the implementation of a multicamera system that captures 4D videos of human grasping and manipulation activities performed on a desktop environment. Thus, the intended output of the target system is a temporal sequence of texture mapped, accurate 3D mesh representations of the observed scene. This constitutes rich perceptual input that may feed higher level modules responsible for scene understanding and human activity interpretation.

From the advent of GPU programmable pipeline, researchers have made great efforts to exploit the computational power provided by the graphics hardware (i.e. GPGPUs). The evolution of GPUs led to the introduction of flexible computing models such as shader model 4.0 and CUDA that support general purpose computations. Various GPU implementations of shape-from-silhouette reconstruction have been presented in the recent literature [1, 2]. Moreover, following past attempts on real-time reconstruction and rendering (e.g. [3, 4]), some recent works introduce full 3D reconstruction systems [5, 6] that incorporate modern graphics hardware for their calculations. The later implementations take as input

segmented object silhouettes and produce as output voxel scene representations. In contrast to these systems, the one proposed in this paper parallelizes the whole processing pipeline that consists of foreground object segmentation, visual hull computation and smoothing, 3D mesh calculation and texture mapping. The algorithms implementing this processing chain are inherently parallel. We capitalize on the enormous computational power of modern GPU hardware through NVIDIA's CUDA framework, in order to exploit this fact and to achieve realtime performance.

The remainder of this paper is organized as follows. Section 2 introduces the system architecture both at hardware and software level. Section 3 details the GPU-based parallel implementation of the 3D reconstruction process. Experiments and performance measurements are presented in Sec. 4. Finally, Sec. 5 provides conclusions and suggestions for future enhancements of the proposed system.

2 Infrastructure

2.1 Hardware Configuration

The developed multicamera system is installed around a $2 \times 1m^2$ bench and consists of 8 *Flea2* PointGrey cameras. Each camera has a maximum framerate of 30 fps at highest (i.e. 1280×960) image resolution. The system employs four computers with quad-core Intel i7 920 CPUs and 6 GBs RAM each, connected by an 1 Gbit ethernet link. Figure 1 shows the overall architecture along with a picture of the developed multicamera system infrastructure.

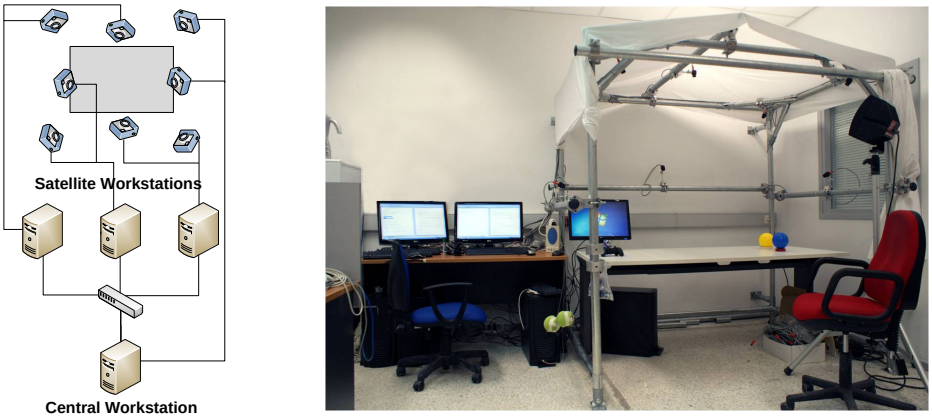
In our *switched-star* network topology, one of the four computers is declared as the *central workstation* and the remaining three as the *satellite workstations*. The central workstation's configuration, includes also a Nvidia GTX 295 dual GPU with 894 $GFlops$ processing power and 896 MBs memory per GPU core. Currently, the developed system utilizes a single GPU core.

Each workstation is connected to a camera pair. Cameras are synchronized by a timestamp-based software that utilizes a dedicated *FireWire 2* interface (800 $MBits/sec$) which guarantees a maximum of 125 μsec temporal discrepancy in images with the same timestamp. Eight images sharing the same timestamp constitute a *multiframe*.

2.2 Processing Pipeline

Cameras are extrinsically and intrinsically calibrated based on the method and tools reported in [7]. The processing pipeline consists of the CPU workflow, responsible for image acquisition and communication management and the GPU workflow, where the 3D reconstruction pipeline has been implemented. Both processes are detailed in the following.

CPU Workflow and Networking
Each workstation holds in its RAM a buffer of fixed size for every camera that



(a) (b)

Fig. 1. The developed platform (a) schematic diagram (b) actual configuration.

is connected to it. Each buffer stores the captured frames after they have been converted from Bayer Tile to RGB format. Moreover, prior to storing in buffer, each image is transformed so that geometric distortions are cancelled out based on the available calibration information. The rate of storing images into buffers matches the camera's acquisition frame rate. Image data are stored together with their associated timestamps. To avoid buffer overflow as newer frames arrive, older frames are removed.

Each time a new image enters a buffer in a satellite workstation, its timestamp is broadcasted to the central workstation. This way, at every time step the central workstation is aware of which frames are stored in the satellite buffers. The same is also true for central's local buffers. During the creation of a multiframe, the central workstation selects the appropriate timestamps for each buffer, local or remote. Then, it broadcasts timestamp queries to the satellite workstations and acquires as response the queried frames, while for local buffers it just fetches the frames from its main memory. The frame set that is created in this way constitutes the multiframe for the corresponding time step. The process is shown schematically in Fig. 2.

GPU Workflow

After a multiframe has been assembled, it is uploaded on the GPU for further processing. Initially, a pixel-wise parallelized foreground detection procedure is applied to the synchronized frames. The algorithm labels each pixel either as background or foreground, providing binary silhouette images as output. The produced silhouette set is given as input to a shape-from-silhouette 3D reconstruction process which, in turn, outputs voxel occupancy information. The occupancy data are then send to an instance of a parallel marching cubes algorithm for computing the surfaces of reconstructed objects. Optionally, prior to mesh

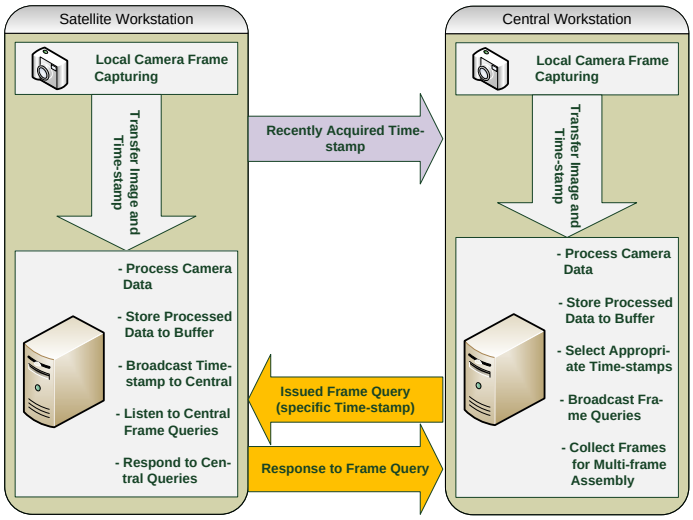


Fig. 2. Multiframe acquisition process.

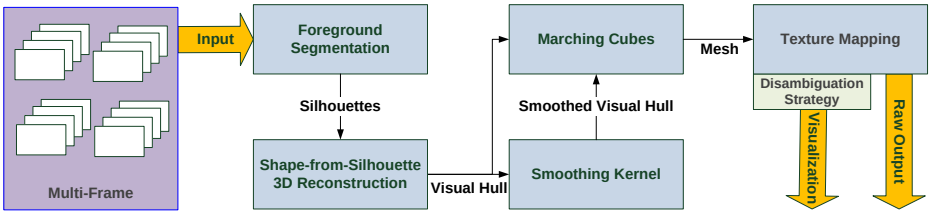


Fig. 3. GPU workflow.

calculation, the voxel representation is convolved with a 3D mean filter kernel to produce a smoothed output. Then, the texture of the original images is mapped onto the triangles of the resulted mesh. During this step multiple texture coordinate pairs are computed for each triangle. Each pair, projects the triangle's vertices at each view the triangle's front face is visible from. A disambiguation strategy is later incorporated to resolve the multi-texturing conflicts. Finally, results are formatted into appropriate data structures and returned to the CPU host program for further processing. In case the execution is intended for visualization, the process keeps the data on the GPU and returns to the host process handles to DirectX or OpenGL data structures (i.e. vertex and texture buffers). These are consequently used with proper graphics API manipulation for onscreen rendering. The overall procedure is presented schematically in Fig. 3.

3 GPU Implementation

In this section, the algorithms implemented on the GPU are presented in detail.

3.1 Foreground Segmentation

The terms *foreground segmentation* and *background subtraction* refer to methods that detect and segment moving objects in images captured by static cameras. Due to the significance and necessity of such methods a great number of approaches have been proposed. The majority of these approaches define pixel-wise operations [8]. The most straightforward of those subtract the average, median or running average within a certain time window from static views. Others utilize kernel density estimators and mean-shift based estimation [9, 10].

A very popular approach [11] that achieves great performance defines each image pixel's appearance model as a mixture of Gaussian components. This method is able to model complex background variations. Targeted at systems operating in relatively controlled environments (i.e., indoor environments with controlled lighting conditions) this work is based on the parallelization of the background modeling and foreground detection work of [12] which considers the appearance of a background pixel to be modeled by a single Gaussian distribution. This reduces substantially both the memory requirements and the overall computational complexity of the resulting process. Moreover, the assumption that pixels are independent, indicates the inherent parallelism of this algorithm. In addition, our implementation incorporates a technique for shadow detection that is also used in [13] and described thoroughly in [14]. Detected shadows are always labeled as background.

Formally, let $I^{(t)}$ correspond to an image of the multiframe acquired at timestamp t, and let $x^{(t)}$ be a pixel of this image represented in some colorspace. The background model is initialized by the first image of the sequence (i.e. $I^{(0)}$) and is given by

$$\hat{p}(\boldsymbol{x}|x^{(0)}, BG) = N(\boldsymbol{x}; \hat{\boldsymbol{\mu}}, \hat{\sigma}^2 I), \tag{1}$$

with $\hat{\boldsymbol{\mu}}$ and $\hat{\sigma}^2$ being the estimates of mean and variance of the Gaussian, respectively. In order to compensate for gradual global light variation, the estimations of $\boldsymbol{\mu}$ and σ are updated at every time step through the following equations:

$$\hat{\boldsymbol{\mu}}^{(t+1)} \leftarrow \hat{\boldsymbol{\mu}}^{(t)} + o^{(t)} \alpha_\mu \boldsymbol{\delta_\mu}^{(t)} \tag{2}$$

$$\sigma^{(t+1)} \leftarrow \sigma^{(t)} + o^{(t)} \alpha_\sigma \delta_\sigma{}^{(t)}, \tag{3}$$

where $\boldsymbol{\delta_\mu} = \boldsymbol{x}^{(t)} - \boldsymbol{\mu}^{(t)}$, $\delta_\sigma = |\boldsymbol{\mu}^{(t)} - \boldsymbol{x}^{(t)}|^2 - \sigma^{(t)}$ and a_μ, a_σ are the update factors for mean and standard deviation, respectively, and

$$o^{(t)} = \begin{cases} 1 & if\, x^{(t)} \in BG \\ 0 & if\, x^{(t)} \in FG. \end{cases} \tag{4}$$

A newly arrived sample is considered as background if the sample's distance to the background mode is less than four standard deviations. If this does not hold,

an additional condition is examined to determine whether the sample belongs to the foreground or it is a shadow on the background:

$$T_1 \leq \frac{\boldsymbol{\mu} \cdot \boldsymbol{x}^{(t)}}{|\boldsymbol{\mu}|^2} \leq 1 \quad and \quad \left| \left(\frac{\boldsymbol{\mu} \cdot \boldsymbol{x}^{(t)}}{|\boldsymbol{\mu}|^2} \right) \boldsymbol{\mu} - \boldsymbol{x} \right|^2 < \sigma^2 T_2 \left(\frac{\boldsymbol{\mu} \cdot \boldsymbol{x}^{(t)}}{|\boldsymbol{\mu}|^2} \right)^2, \quad (5)$$

where T_1, T_2, are empirically defined thresholds that are set to $T_1 = 0.25$, $T_2 = 150.0$.

The above described foreground detection method has been parallelized in a per pixel basis. In addition, because there is a need to preserve the background model for each view, this is stored and updated on GPU during the entire lifetime of the reconstruction process. In order to keep the memory requirements low and to meet the GPU alignment constrains, the background model of each pixel is stored in a 4-byte structure. This representation leads to a reduction of precision. Nevertheless, it has been verified experimentally that this does not affect noticeably the quality of the produced silhouettes.

3.2 Visual Hull Computation

The idea of *volume intersection* for the computation of a volumetric object description was introduced in the early 80's [15] and has been revisited in several subsequent works [16–18]. The term *visual hull*, is defined as the maximal shape that projects to the same silhouettes as the observed object on all views that lay outside the convex hull of the object [19].

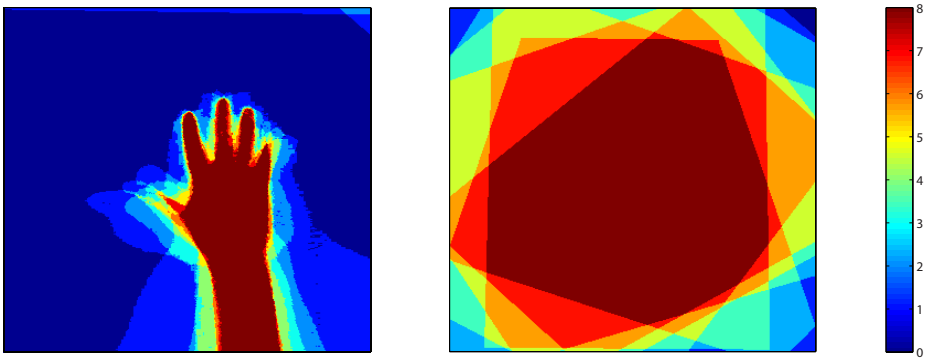
To compute the visual hull, every silhouette image acquired from a given multiframe, is back-projected and intersected into the common 3D space along with all others, resulting to the *inferred visual hull*, i.e. a voxel representation containing occupancy information. In this 3D space, a fixed size volume is defined and sampled to produce a 3D grid, $G = \left\{G^0, G^1, \ldots, G^n\right\}$ where $G^c = (X_c, Y_c, Z_c)$. Let C_i be the calibration matrix of camera i and R_i, T_i the corresponding rotation matrix and translation vector respectively, in relation to the global world-centered coordinate system. The general perspective projection of a point G expressed in homogeneous coordinates (i.e. $(X_c, Y_c, Z_c, 1)$) to the i^{th} view plane is described through the following equation

$$(x_c, y_c, f_c)^T = C_i \left[R_i | T_i\right] (X_c, Y_c, Z_c, 1)^T, \quad (6)$$

where $P_i = C_i \left[R_i | T_i\right]$ is the projection matrix of the corresponding view. Each point can be considered to be the mass center of some voxel on the defined 3D volume. We also define two additional functions. The first, labels projections falling inside the FOV of camera i as

$$L_i(x, y) = \begin{cases} 1 & 1 \leq x \leq w_i \ \wedge 1 \leq y \leq h_i \\ 0 & otherwise, \end{cases} \quad (7)$$

where w_i and h_i denote the width and height of the corresponding view plane, respectively. The second function measures the occupancy scores of each voxel



(a) (b)

Fig. 4. Each figure presents a xy plane slice of the voxel space. (a) The intersection of the projected silhouettes in slice $Z_{slice} = 90cm$. (b) The voxel space defined in this example is much larger than the previous, visibility factor variations are shown with different colors. Dark red denotes an area visible by all views.

via its projected center of mass, as

$$O(X_k, Y_k, Z_k) = \begin{cases} 1 & s = l > \frac{|C|}{2} \\ 0 & otherwise \end{cases}, \quad \forall k \in [0, n], \tag{8}$$

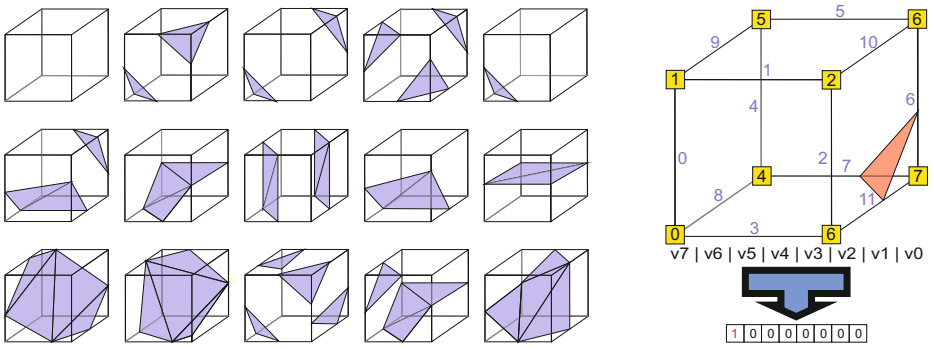
where $|C|$ is the number of views. l is the *visibility factor*, s the *intersection factor* and are defined as

$$l = \sum_{i \in C} L_i \left(\frac{x_k^i}{f_k^i}, \frac{y_k^i}{f_k^i} \right), \quad s = \sum_{i \in C} S_i \left(\frac{x_k^i}{f_k^i}, \frac{y_k^i}{f_k^i} \right), \tag{9}$$

with $\left(x_k^i/f_k^i, y_k^i/f_k^i\right)$ be the projection of (X_k, Y_k, Z_k) at view i and $S_i(x, y)$ is the function that takes value 1 if at view i the pixel (x, y) is a foreground pixel and 0 otherwise (i.e. background pixel). Figure 4 illustrates graphically the notion of l and s.

The output of the above process is the set $O(X_k, Y_k, Z_k)$ of occupancy values that represent the visual hull of the reconstructed objects. It can also be conceived as the estimation of a 3D density function. Optionally, the visual hull can be convolved with a 3D mean filter to smooth out the result. Due to its high computational requirements, this method targets the offline mode of 3D reconstruction.

The above described 3D reconstruction process has been parallelized on a per 3D point basis. More specifically, each grid point is assigned to a single GPU thread responsible for executing the above mentioned calculations. To speed up the process, shared memory is utilized for storing the static per thread block calibration information, silhouette images are preserved in GPU texture memory in a compact bit-per-pixel format and occupancy scores are mapped to single bytes.



(a) (b)

Fig. 5. (a) Marching Cubes fundamental states. (b) Byte representation and indexing.

3.3 Marching Cubes

Marching cubes [20, 21] is a popular algorithm for calculating isosurface descriptions out of density function estimates. Due to its inherent and massive data parallelism it is ideal for GPU implementation. Over the last few years, a lot of isosurface calculation variates that utilize GPUs have been proposed [22–26]. In this work we employ a slightly modified version of the marching cubes implementation found at [27] due to its simplicity and speed. More specifically, the occupancy grid resulting from 3D visual hull estimation is mapped into a CUDA 3D texture. Each voxel is assigned to a GPU thread. During calculations, each thread samples the density function (i.e. CUDA 3D texture) at the vertices of it's corresponding voxel. The normalized (in the range $[0, 1]$), bilinearly interpolated, single precision values returned by this step, represent whether the voxel vertices are located inside or outside a certain volume. We consider the isosurface level to be at 0.5. Values between 0 and 1, also show how close a voxel vertex is to the isosurface level. Using this information, a voxel can be described by a single byte, where each bit corresponds to a vertex and is set to 1 or to 0 if this vertex lays inside or outside a volume, respectively. There are 256 discrete generic states in which a voxel can be intersected by an isosurface fragment, produced from the 15 fundamental cases illustrated in Fig. 5a.

Parallel marching cubes uses two constant lookup tables for its operation. The first lookup table is indexed by the voxel byte representation and is utilized for determining the number of triangles the intersecting surface consists of. The second table is a 2D array, where its first dimension is indexed by the byte descriptor and the second by an additional index $trI \in [0, 3N_{iso} - 1]$ where N_{iso} is the number of triangles returned by the first lookup. Given a byte index, sequential triplets accessed through successive trI values, contain the indices of voxel vertices intersected by a single surface triangle. An example of how the voxel byte descriptor is formed is shown in Fig. 5b. This figure also presents the

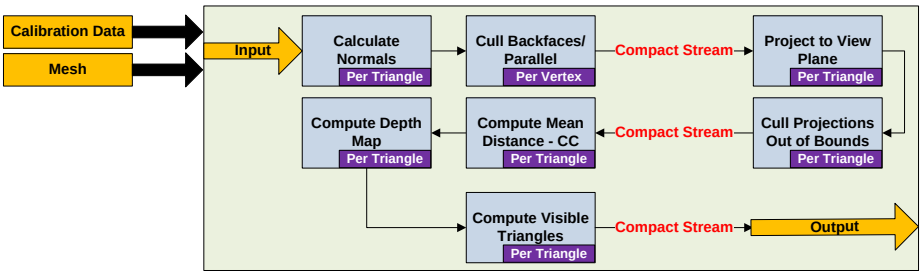


Fig. 6. Computation of texture coordinates.

vertex and edge indexing along with an example of an intersecting isosurface fragment that consists of a single triangle.

To avoid applying this process to all voxels, our implementation determines the voxels that are intersected by the iso-surface and then, using the CUDA data parallel primitives library [28], applies stream compaction through the exclusive sum-scan algorithm [29] to produce the minimal voxel set containing only intersected voxels. Finally, lookup tables are mapped to texture memory for fast access.

3.4 Texture Mapping

Due to the fact that the employed camera setup provides multiple texture sources, texture mapping of a single triangle can be seen as a three step procedure: a) determine the views from which the triangle is visible, b) compute the corresponding texture coordinates and c) apply a policy for resolving multitexturing ambiguities (i.e. texture selection). The current implementation carries out the first two steps in a per view manner i.e.: a) determines the subset of triangles that are visible by a certain view and b) computes their projections on view plane. The third step is applied either on a per pixel basis through a pixel shader during the visualization stage, or is explicitly computed by the consumer of the offline dataset.

Specifically, given the calibration data for a view and the reconstructed mesh, a first step is the calculation of the triangle normals. Then, the direction of each camera's principal axis vector is used to cull triangles back-facing the camera or having an orientation (near-)parallel to the camera's view plane. The triangle stream is compacted excluding culled polygons and the process continues by computing the view plane projections of the remaining triangles. Projections falling outside the plane's bounds are also removed through culling and stream compaction. Subsequently, the mean vertex distance from the camera center is computed for each remaining triangle and a depth testing procedure (Z-buffering) is applied to determine the final triangle set. The procedure is shown schematically in Fig. 6. This figure also shows the granularity of the decomposition in independent GPU threads. During depth testing, CUDA atomics are used for issuing writes on the depth map. The reason for the multiple culling

Image resolution	Multiframe acquisition	Foreground segmentation
320×240	30 *mfps*	$22.566,3$ *fps* / $2.820,8$ *mfps*
640×480	13 *mfps*	$6.773,9$ *fps* / $846,4$ *mfps*
800×600	9 *mfps*	$4.282,6$ *fps* / $535,3$ *mfps*
1280×960	$3,3$ *mfps*	$1.809,9$ *fps* / $226,2$ *mfps*

Table 1. Performance of acquisition and segmentation for various image resolutions.

iterations prior to depth testing is for keeping the thread execution queues length minimal during serialization of depth map writes.

There is a number of approaches that one can use to resolve multitexturing conflicts. Two different strategies have been implemented in this work. The first assigns to each triangle the texture source at which the projection area is maximal among all projections. The second blends all textures according to a weighting factor, proportional to the size of the projected area. A special case is the one where all weights are equal. This last approach is used during online experiments to avoid the additional overhead of computing and comparing the projection areas, while the others are used in offline mode for producing better quality results. In online mode the process is applied through a pixel shader implemented using HLSL and shader model 3.0. Visualizations of a resulted mesh are shown in Fig. 7. The supplemental material attached to this paper shows representative videos obtained from both online and offline experiments.

4 Performance

Given a fixed number of cameras, the overall performance is determined by the network bandwidth, the size of transferred data, the GPU execution time and the quality of the reconstruction. In online experiments, camera images are preprocessed, transferred through network and finally collected at the central workstation to construct a synchronized multiframe. This is performed at a rate of 30 multiframes per second (mfps). To achieve this performance, original images (i.e. 1280×960) are resized during the CPU preprocessing stage to a size of 320×240. Further reduction of image resolution increases the framerate beyond real-time (i.e. $\geq 30\, mfps$) at the cost of reducing the 3D reconstruction quality. Table 1 shows the achieved multiframe acquisition speed.

Table 1 also shows that, as expected, foreground segmentation speed is linearly proportional to image size. These last reported measurements do not include CPU/GPU memory transfers.

The number of voxels that constitute the voxel space is the primary factor that affects the quality of the reconstruction and overall performance. Given a bounded voxel space (i.e., observed volume), smaller voxel sizes, produce more accurate estimates of the 3D density function leading to a reconstruction output of higher accuracy. Moreover, higher voxel space resolutions issue greater numbers of GPU threads and produce more triangles for the isosurface that, in turn, leads to an increased overhead during texture mapping. The performance

(a) (b) (c)

Fig. 7. 3D reconstruction of a single multiframe: (a) no smoothing, (b) smoothed reconstruction and (c) smoothed and textured output.

graph of Fig. 8a shows the overall performance impact of voxel space resolution increment in the cases of a) no smoothing of the visual hull, b) smoothed hull utilizing a 3^3 kernel and c) smoothed hull utilizing a 5^3 kernel. The graph in Fig. 8b presents computational performance as a function of smoothing kernel size. In both graphs, multiframe processing rate corresponds at the processing rate of the entire GPU pipeline including the CPU/GPU memory transfer times. It is worth mentioning that although image resolution affects the quality of the reconstruction and introduces additional computational costs due to the increased memory transfer and foreground segmentation overheads, it does not have a significant impact on the performance of the rest of the GPU reconstruction pipeline.

Table 2 presents quantitative performance results obtained from executed experiments. In the 3rd and 4th columns, the performance of 3D reconstruction and texture mapping are shown independently. The 3D reconstruction column corresponds to the processes of computing the visual hull, smoothing the occupancy volume and creating the mesh, while texture mapping column corresponds to the

Voxels	**Smoothing**	**3D reconst.**	**Text. mapping**	**Output**
Online Experiments				
$120 \times 140 \times 70$	No	$136,8$ *mfps*	$178,0$ *mfps*	$64,0$ *mfps*
$100 \times 116 \times 58$	No	$220,5$ *mfps*	$209,9$ *mfps*	$84,5$ *mfps*
Offline Experiments				
$277 \times 244 \times 222$	Kernel: 3^3	$7,7$ *mfps*	$27,5$ *mfps*	$5,0$ *mfps*
$277 \times 244 \times 222$	Kernel : 5^3	$4,7$ *mfps*	$28,9$ *mfps*	$3,5$ *mfps*
$277 \times 244 \times 222$	No	$11,4$ *mfps*	$25,3$ *mfps*	$6,2$ *mfps*

Table 2. Quantitative performance results obtained from representative experiments. Image resolution is set to 320×240 for online and 1280×960 for offline experiments.

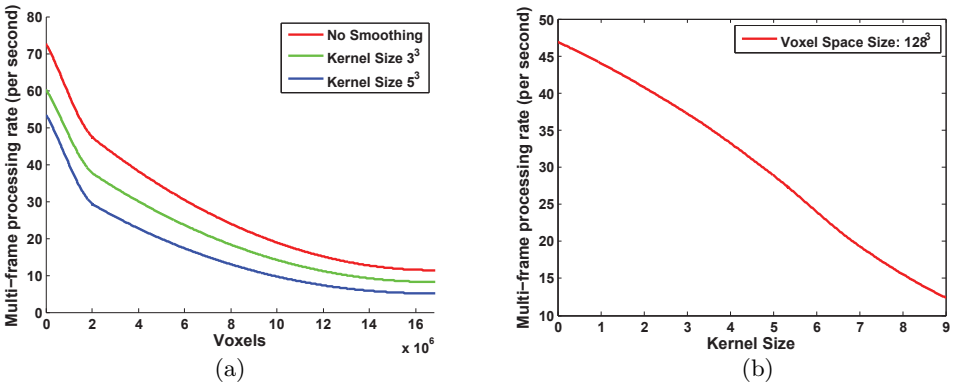


Fig. 8. Performance graphs. Image resolution is set to 640 × 480 in all experiments. (a) Performance impact of voxel space descretization resolution. (b) The performance effect of 3D smoothing kernel size.

performance of the process depicted in Fig. 6. Finally, in the output column, as in the previous experiments, the performance of the entire reconstruction pipeline is measured including foreground segmentation and memory transfers. It can be seen that keeping the voxel space resolution at a fixed size, the multiframe processing rate of 3D reconstruction drops significantly when the smoothing process is activated. On the contrary, texture mapping is actually accelerated due to the fact that the smoothed surface is described by less triangles than the original one. Online experiments present clearly the effect of the voxel space resolution in overall performance.

5 Conclusions - Future Work

In this paper, we presented the design and implementation of an integrated, GPU-powered, multicamera vision system that is capable of performing foreground image segmentation, silhouette-based 3D reconstruction, 3D mesh computation and texture mapping in real-time. In online mode, the developed system can support higher level processes that are responsible for activity monitoring and interpretation. In offline mode, it enables the acquisition of high quality 3D datasets. Experimental results provide a quantitative assessment of the system's performance. Additionally, the supplementary material provides qualitative evidence regarding the quality of the obtained results.

The current implementation utilizes a single GPU. A future work direction is the incorporation of more GPUs either on central or satellite workstations, to increase the system's overall raw computational power in terms of GFlops. In this case, an intelligent method for distributing the computations over the entire GPU set must be adopted, while various difficult concurrency and synchronization issues that this approach raises must be addressed. Furthermore,

performance gains could be achieved by transferring the image post-acquisition CPU processes of Bayer Tile-to-RGB conversion and distortion correction to GPUs as they also encompass a high degree of parallelism. Finally, mesh deformation techniques instead of density function smoothing and advanced texture source disambiguation/blending strategies that incorporate additional information (e.g. edges) can be utilized in order to further augment the quality of the results.

Acknowledgments

This work was partially supported by the IST-FP7-IP-215821 project GRASP and by the FORTH-ICS internal RTD Programme "Ambient Intelligence and Smart Environments".

References

1. Kim, H., Sakamoto, R., Kitahara, I., Toriyama, T., Kogure, K.: Compensated visual hull with gpu-based optimization. Advances in Multimedia Information Processing-PCM 2008 (2008) 573–582
2. Schick, A., Stiefelhagen, R.: Real-time gpu-based voxel carving with systematic occlusion handling. In: Pattern Recognition: 31st DAGM Symposium, Jena, Germany, September 9-11, 2009, Proceedings, Springer-Verlag New York Inc (2009) 372
3. Matusik, W., Buehler, C., Raskar, R., Gortler, S.J., McMillan, L.: Image-based visual hulls. In: SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques, New York, NY, USA, ACM Press/Addison-Wesley Publishing Co. (2000) 369–374
4. Matsuyama, T., Wu, X., Takai, T., Nobuhara, S.: Real-time 3d shape reconstruction, dynamic 3d mesh deformation, and high fidelity visualization for 3d video. Computer Vision and Image Understanding **96** (2004) 393–434
5. Ladikos, A., Benhimane, S., Navab, N.: Efficient visual hull computation for real-time 3d reconstruction using cuda. In: IEEE Conference on Computer Vision and Pattern Recognition, Workshops 2008. (2008) 1–8
6. Waizenegger, W., Feldmann, I., Eisert, P., Kauff, P.: Parallel high resolution real-time visual hull on gpu. In: IEEE International Conference on Image Processing. (2009) 4301–4304
7. Sarmis, T., Zabulis, X., Argyros, A.A.: A checkerboard detection utility for intrinsic and extrinsic camera cluster calibration. Technical Report TR-397, FORTH-ICS (2009)
8. Piccardi, M.: Background subtraction techniques: a review. In: IEEE International Conference on Systems, Man and Cybernetics. Volume 4. (2004) 3099–3104
9. Elgammal, A., Harwod, D., Davis, L.: Non-parametric model for background subtraction. In: IEEE International Conference on Computer Vision, Frame-rate Workshop. (1999)
10. Han, B., Comaniciu, D., Davis, L.: Sequential kernel density approximation through mode propagation: applications to background modeling. In: Asian Conference on Computer Vision. (2004)

11. Stauffer, C., Grimson, W.: Adaptive background mixture models for real-time tracking. In: IEEE Conference on Computer Vision and Pattern Recognition. (1999) 246–252
12. Wren, C., Azarbayejani, A., Darrell, T., Pentland, A.: Pfinder: Real-time tracking of the human body. IEEE Transactions on Pattern Analysis and Machine Intelligence **19** (1997) 780–785
13. Zivkovic, Z.: Improved adaptive gaussian mixture model for background subtraction. In: International Conference on Pattern Recognition. (2004)
14. Prati, A., Mikic, I., Trivedi, M.M., Cucchiara, R.: Detecting moving shadows: Algorithms and evaluation. IEEE Transactions on Pattern Analysis and Machine Intelligence **25** (2003) 918–923
15. Martin, W., Aggrawal, J.: Volumetric descriptions of objects from multiple views. IEEE Transactions on Pattern Analysis and Machine Intelligence (1983)
16. Srinivasan, P., Liang, P., Hackwood, S.: Computational geometric methods in volumetric intersection for 3d reconstruction. Pattern Recognition **23** (1990) 843 – 857
17. Greg, F.P., Slabaugh, G., Culbertson, B., Schafer, R., Malzbender, T.: A survey of methods for volumetric scene reconstruction. In: International Workshop on Volume Graphics. (2001)
18. Potmesil, M.: Generating octree models of 3d objects from their silhouettes in a sequence of images. Computer Vision, Graphics, and Image Processing **40** (1987) 1–29
19. Laurentini, A.: The visual hull concept for silhouette-based image understanding. IEEE Transactions on Pattern Analysis and Machine Intelligence **16** (1994) 150–162
20. Lorensen, W.E., Cline, H.E.: Marching cubes: A high resolution 3d surface construction algorithm. Computer Graphics **21** (1987) 163–169
21. Newman, T.S., Yi, H.: A survey of the marching cubes algorithm. Computers and Graphics **30** (2006) 854– 879
22. Klein, T., Stegmaier, S., Ertl, T.: Hardware-accelerated reconstruction of polygonal isosurface representations on unstructured grids. In: PG '04: Proceedings of the Computer Graphics and Applications, 12th Pacific Conference, Washington, DC, USA, IEEE Computer Society (2004) 186–195
23. Pascucci, V.: Isosurface computation made simple: Hardware acceleration, adaptive refinement and tetrahedral stripping. In: In Joint Eurographics - IEEE TVCG Symposium on Visualization (VisSym. (2004) 293–300
24. Reck, F., Dachsbacher, C., Grosso, R., Greiner, G., Stamminger, M.: Realtime isosurface extraction with graphics hardware. In: Proceedings of Eurographics. (2004)
25. Goetz, F., Junklewitz, T., Domik, G.: Real-time marching cubes on the vertex shader. In: Proceedings of Eurographics. Volume 2005. (2005)
26. Johansson, G., Carr, H.: Accelerating marching cubes with graphics hardware. In: In CASCON 06: Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research, ACM, Press (2006) 378
27. NVIDIA. GPU Computing SDK (2009) http://developer.nvidia.com/object/gpu computing.html.
28. Harris, M., Sengupta, S., Owens, J. CUDA Data Parallel Primitives Library (2007) http://code.google.com/p/cudpp/.
29. Sengupta, S., Harris, M., Zhang, Y., Owens, J.D.: Scan primitives for gpu computing. In: Graphics Hardware 2007, ACM (2007) 97–106