Accelerated Real-Time Reconstruction of 3D Deformable Objects from Multi-View Video Channels

Holger Graf[1], Leon Hazke[1], Svenja Kahn[1], Cornelius Malerczyk[2]

[1] Fraunhofer Institute for Computer Graphics Research IGD,
Fraunhoferstr. 5,
64283 Darmstadt, Germany
{Holger.Graf, Leon.Hazke, Svenja.Kahn}@igd.fraunhofer.de

[2] University of Applied Science Giessen-Friedberg,
Faculty of Mathematics, Natural Sciences and Computer Sciences
Wilhelm-Leuschner-Strasse 13,
61169 Friedberg, Germany
{Cornelius.Malerczyk}@mnd.fh-friedberg.de

Abstract. In this paper we present a new framework for an accelerated 3D reconstruction of deformable objects within a multi-view setup. It is based on a new memory management and an enhanced algorithm pipeline of the well known Image-Based Visual Hull (IBVH) algorithm that enables efficient and fast reconstruction results and opens up new perspectives for the scalability of time consuming computations within larger camera environments. As a result, a significant increase of frame rates for the volumetric reconstruction of deformable objects can be achieved using an optimized CUDA-based implementation on NVIDIA's Fermi-GPUs.

Keywords: Image based 3D reconstruction, GPU, CUDA, Real-Time reconstruction, Image-Based Visual Hull.

1 Introduction

In this paper we present an efficient high resolution Image Based Visual Hull (IBVH) algorithm that entirely runs in real-time on a single consumer graphics card. The target application is a real-time motion capture system with a full body 3D reconstruction. The topic of 3D scene reconstruction based on multiple images has been investigated during the last twenty years and produced numerous results in the area of computer graphics and computer vision. Especially, real-time 3D reconstruction of target objects within a GPU environment has become one of the hot issues nowadays.
Image based visual hull reconstruction (IBVH) is a 3D reconstruction technique from multiple images [1], which is based on the visual hull concept defined by Laurentini [2]. The visual hull of a 3D object is a conservative 3D shell, which contains the 3D object. The IBVH algorithm calculates the visual hull of a 3D object by intersecting

the silhouette cones of the 3D object in several images. Whereas other 3D reconstruction techniques are often computationally intensive or limited to the reconstruction of static 3D scenes [3,4,5], the IBVH algorithm can reconstruct the 3D visual hull of rigid and deformable objects. A common application scenario of 3D visual hull reconstruction from multi-view video channels is the acquisition, analysis and visualization of human movements. Corazza et al. estimate human body poses by fitting an articulated 3D model of a human into the reconstructed visual hull of a person [6]. Graf, Yoon and Malerczyk reconstruct the visual hull to give the user the possibility to inspect the deformable object from an arbitrary viewpoint [7]. While the user moves a virtual camera around the deformable object, new views of the object are synthesized in real-time from the point of view of the virtual camera. Whereas the visual hull based pose estimation proposed in [6] is an offline process, the 3D reconstruction for the synthesis of new views needs to be calculated in real-time [7]. The processing time of the visual hull reconstruction increases with the number of cameras and their resolution. On current state-of-the-art processors, the frame rate of the CPU-based IBVH algorithm on multi-view video channels [1] is not high enough for a smooth real-time visualization.

However, the immense speed-up of the processing power of graphical processing units in the last few years allows for the development of very fast GPU based 3D reconstruction algorithms.

The mapping of computer vision algorithms to the GPU can significantly accelerate their processing time. Here, Owens et al. present a broad survey about general-purpose computations on graphics processors [8], Fung and Mann focus on the mapping of computer vision and image processing tasks to the GPU [9]. Rossi et. al [10] as well as Weinlich et al. [11] show that 3D reconstruction algorithms can be considerably accelerated by adapting reconstruction algorithms such that they can be executed on the GPU. Waizenegger et al. [12] as well as Ladikos, Benhimane and Navab [13] and Kim et al. [14] proposed GPU versions of visual hull 3D reconstruction algorithms. Whereas Ladikos, Benhimane and Navab [13] and Kim et al. [14] accelerate visual hull algorithms which use a voxel representation to explicitly reconstruct the 3D geometry, Waizenegger et al. [12] present a GPU version of image based visual hull reconstruction reducing the number of line segment intersection tests by a caching strategy. In this paper we also propose a GPU accelerated image based visual hull reconstruction algorithm. However, in contrast to [12], we focus on adapting the algorithm pipeline and its memory management such that they are optimized for the parallelized GPU architecture.

Castano-Diez et al. evaluate the performance of image processing algorithms on the GPU [15]. They note that the speed-up which can be attained by mapping an algorithm to the GPU strongly depends on the characteristics of the algorithm itself: A high speed-up can be attained for algorithms which exhibit a high level of data parallelism (operations which are independent from one another) and which are characterized by a high number of sequentially and independently applied instructions. Whereas algorithms, which fulfill these requirements can benefit from a high speed-up if they are mapped to the GPU, memory access can significantly slow down the calculations. This is due to the fact that the GPU processing pipeline is not optimized for fast access of large amounts of memory.

In this work we present a 3D reconstruction framework, which accounts for these requirements to attain a maximal speed-up of the reconstruction algorithm on the GPU with CUDA. The algorithm pipeline and the memory management of our 3D reconstruction framework have been carefully adapted to the inherent parallelism of GPU architectures.

2 Concept

We follow a photo-realistic 3D reconstruction methodology from multiple images that exploits camera calibration data and foreground extraction information. We extract the silhouette of a foreground object from an array of multiple static cameras using kernel density estimation based background subtraction. The IBVH algorithm computes a 3D volumetric reconstruction of a target object from its 2D projections captured from a sequence of images in different viewpoints (viewing rays). We consider a scene observed by n calibrated static cameras and we focus on the state of one voxel at position X chosen among the positions of the 3D lattice used to discretize the scene (figure 1).

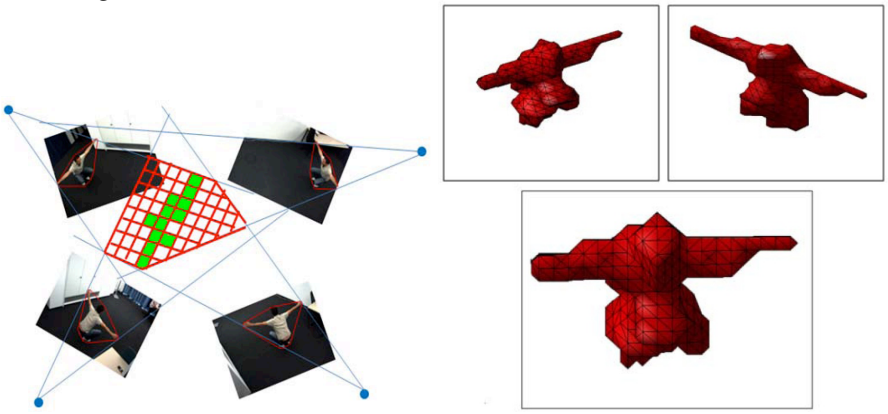

Fig. 1. Basic principle of the IVBH mechanism: on the left, different viewing rays and line intersections reconstructing the 3D lattice, on the right, the resulting target object (untextured).

For the available silhouettes we assume, that each silhouette is represented by a binary array and the positions can be derived by using extrinsic and intrinsic calibration information of each camera. As a result of the IVBH we obtain a sequence of 3D intervals along each viewing ray. Those intervals contain the volumetric area of the target object. The proposed framework exploits the hardware features of recent consumer graphics cards and is deployed on a single graphic board. For our implementation we used an open source CPU implementation of the IBVH (Image Based Visual Hull) algorithm as proposed in [1]. In order to speed up the overall process chain, the algorithm pipeline and its memory management have been adapted to the inherent parallelized GPU architecture. Our concept is based on the subsequent strategies.

2.1 Optimizing Memory Management for the 2D and 3D Intervals

In order to fully exploit the parallelism of the CUDA architecture the computation of several viewing rays is mapped to individual CUDA threads. As a consequence, we are using as many CUDA threads as available viewing rays. If the maximum number of intersection points for each viewing ray is defined to n_{vr}, the allocated memory block M for the computation of the intersection points is

$$M = n_{vr} \cdot I_w \cdot I_h \cdot n_{Ip} \cdot P_{Ip}(x,y),\ \forall (x,y) \in I_p, n_{Ip} = \left| I_p \right|$$

with I_p is the set of 3D intersection points, P_{Ip} the size of an intersection element at pixel value (x,y), I_w , I_h image width, resp. image height. In order to efficiently implement the overall memory consumption, memory access is realized in local areas rather than within global alignment procedures. Here, several threads within a CUDA WARP (smallest unit of threads (#32), capable of being physically executed in parallel by the multiprocessor) could access sequentially ordered data blocks, i.e. neighbored threads access sequentially ordered data sets. Thus, the pattern of memory storage resembles a volumetric lattice with multiple *"image layers"*. Within each *"image layer"* we store the intersection points of each corresponding viewing ray. The number of *image layers* complies with the maximum number of intersection points.

The challenge we have been facing is, that the course of the sampling of the image along its epipolar lines (see figure 2) within each viewpoint is not optimal for a CUDA implementation.

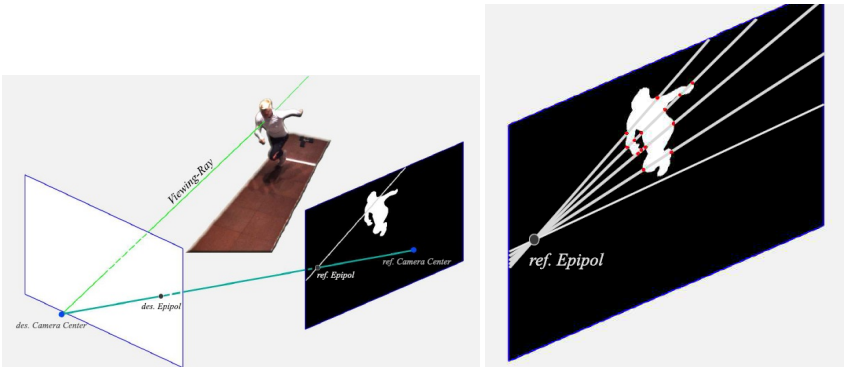

Fig. 2. Projection of viewing ray onto reference image (containing silhouette), left; traversal (sampling) of reference image along epipolar line, right.

Coalesced memory access is not possible as the threads within a WARP typically cannot access the resulting, neighbored (and sequentially aligned) data sets. To reduce the negative impact on the performance, we store the silhouette images as textures. The use of textures is optimized for 2D memory access as the multiprocessor is capable of transferring 2D areas of an image into its texture cache. This strategy allows us to make use of the overall CUDA capability for fast texture cache access

that is as fast and efficient as classical register access[1]. We achieve a further optimization by reducing the number of cache mismatches. This can be achieved by a "packaging" strategy for the silhouette images. Here, we condense the images to an image element (one byte for 8 images or two bytes for 16 images) that contains the information on multiple images (1 bit for 1 image[2]).

2.2 GPU Adapted Silhouette-Line Intersection Algorithm

For further memory optimization we adapted the Wedge-Cache algorithm described by Matusik [1] by adjusting the memory storage patterns for parallel processing on the GPU. The idea of the Wedge Cache algorithm is that the projections of the viewing rays onto the image plane lead to a 2D-cone. As a consequence projections of different rays may coincide and therefore induce the same epipolar lines, which have to be calculated once only and lead to a reduction of memory accesses. The algorithm uses indexed border pixels and therefore, it is possible to assign the epipolar lines, which are used for traversal to a specific wedge cache index, which is calculated as the furthest intersection point of epipolar lien and image border (see fig. 3). Each traversal of an epipolar line is implemented as a single GPU thread and therefore, the number of threads is reduced to 2*image width + 2*image height. Furthermore, all intersection points of epipolar lines with the object (silhouette) border are stored in the corresponding position (wedge cache index).

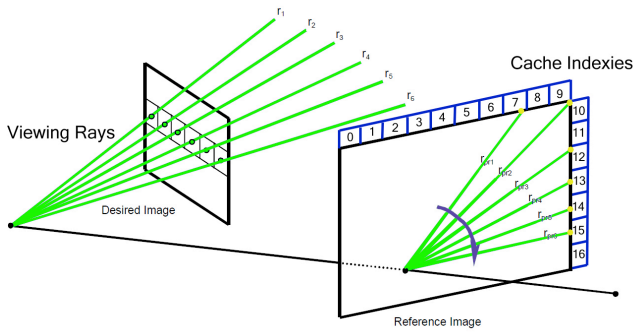

Fig. 3. Projection of viewing rays onto the image plane and building the wedge cache indexes

2.3 Efficient CUDA Alignment

Within this step, we bundle the optimized calculation of silhouette line intersections (bundle of viewing rays) into a dedicated CUDA block. This again leads to an

[1] Note: this is only possible if several threads within a CUDA block actually access the uploaded 2D area, otherwise it results in a cache mismatch.

[2] The silhouette images are stored in a binary array.

increase of locally aligned memory access reducing the number of cache mismatches and avoiding longer latencies. The projections of bundles of rays lead to several epipolar lines on the image plane. Due to the fact that they distant against each other towards the image borders, their cache indexes may not be consecutive. Therefore, access to the wedge cache is due to the CUDA coalesced memory access only sequential possible. To avoid this we arrange the CUDA blocks in such a way that viewing rays in x- and y-direction are calculated within one block of the size 16*16 or 32*32 (depending on the used GPU).
Further on, we configure the execution of the CUDA kernels in such a way, that a CUDA block is capable of sampling multiple images along the epipolar line in parallel. Thus we reduce the cache mismatches by increasing the probability of successful memory accesses to the same 2D areas by several threads within one CUDA block. An alignment of the CUDA blocks according to the position of the epipole of the image plane leads to a further optimization in handling the CUDA memory. As to this position, not all areas of the wedge cache alignment have to be computed. We therefore achieve a higher degree of parallelism, which we deploy in the subsequent implementation.

3 Implementation

The GPU architecture is designed for massive parallel computations. It is necessary to use as much streams processors as possible at the same time to achieve a maximal computational power. Therefore, the original IBVH algorithm described by Matusik [1] has been adapted in such a way that all separately computable portions of the algorithm exploit the GPU's parallel capabilities. By executing multiple CUDA kernels in parallel, we fully deploy the GPU capacities and its utilization reducing the overall execution time for the 3D reconstruction by avoiding several time consuming memory access tasks within a local area rather than accessing global memory reducing the overall latency of data transfer. Especially the calculation of the 3D cones (as describes in the previous sections) allows a high degree of parallelism and therefore a vast amount of saving computational time due to the facts that is it possible to calculate all viewing rays independently. Nevertheless, the IBVH algorithm is obviously not parallelizable as a whole and parallelization has to be implemented in several sections that are executed sequentially:

```
ComputeMatrixAndEpipols ();

initializeViewingRayBuffer ();

initializeWedgeCacheBuffer ();

compressSilhouettes ();

compute2DIntervals ();

for each referenceImage r in R
```

```
computeViewingRayAndProjekt2DIntervalsBackIn3D ();

merge3DIntervals ();
```

All methods of the pseudo code above are realized as separated CUDA kernels.

3.1 Hardware Setup

As usual for computer vision based algorithm that have to deal with real-time requirements, it is important to have an appropriate hardware setup at hand. For our system we use TheImagingSource[3] DBK 21BF04 Firewire cameras with a VGA resolution of 640*480 pixels and 4mm lenses. Using a Bayer filter the cameras deliver up to 60 frames per second in 8-bit mode, which are converted to RGB images by de-mosaicing them under CUDA on the GPU. To ensure that all image tuples used for both calibration and reconstruction purposes are acquired exactly at the same time, the cameras are triggered by a 5 volts TTL signal. We use an USB-PIO device by BMC Messsysteme GmbH[4], which features three 8-bit bidirectional ports and therefore allows to trigger up to 127 cameras simultaneously. All cameras are connected with BNC cables to a server PC, which is used to generate the trigger signals. Nevertheless, using multiple calibrated cameras for the reconstruction process leads to the typical bottleneck of vision-based systems dealing with a huge amount of data that have to be transmitted via several different busses. Both Firewire and PCI Express bus allow up to four cameras to be connected to a single PC.
Therefore, we propose two different hardware setup approaches depending on the count of used cameras, one for up to four cameras with a single PC and one for five or more cameras using a network-based client-server architecture.
The first system is able to build the visual hull out of up to four synchronous cameras, which are connected to a single PC, which is used for as well triggering the camera images as al image processing and computer vision tasks (see fig 4, left).

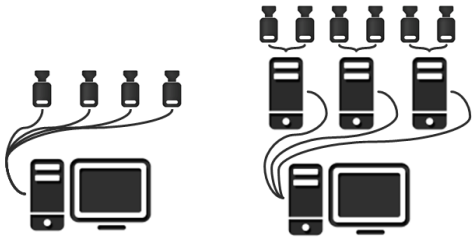

Fig. 4. Hardware system setup: Up to four cameras connected to one single PC (left) and scalable setup for n cameras using a client/server architecture (right)

Obviously, using only four cameras the reconstruction is limited to a restricted volume due to the fact that the target object has to be fully visible in all camera

[3] http://www.theimagingsource.com/
[4] http://www.bmc-messsysteme.de/us/

images at all time. To overcome this limitation more cameras surveying the interaction volume are used, which are connected to several PCs in a network. While two to four cameras each are connected to one client, one PC is used a the server triggering the cameras, performing the visual hull reconstruction on the GPU and rendering the output images. All other PCs are used as clients preprocessing the camera images with de-mosaicing the incoming images and performing the foreground extraction. Due to the fact that after silhouette extraction not all full images but run length encoded images of the silhouettes only are transmitted to the server PC, the system is able to perform with more than 40 frames per second using e.g. eight synchronous cameras.

4 Evaluation

We tested the CUDA-based implementation of the algorithm described in the previous sections on different setups. Due to the fact that only of-the-shelf hardware is used for the system setup, GPU tests were done with consumer graphics cards NVIDIA GeForce only. Parameters for the evaluation tests are the overall processing times in milliseconds running through the algorithm from the first to the last step of the algorithm (including visualization and data transfer from and to GPU) and the necessary memory used on the graphics board. The following table shows the results using five different graphic cards using setups with four, eight and sixteen cameras at three different levels of image resolutions.

Table 1. Framework benchmarks on different GPUs at different resolutions.

GPU	16 Cameras	8 Cameras	4 Cameras	Resolution	Memory Usage (16 cameras)
GeForce GTX 470	4.1 ms	2.7 ms	2.0 ms	320 x 240	429 Mb
GeForce GTX 460	6.9 ms	3.8 ms	2.4 ms	320 x 240	
GeForce GTX 260	6.4 ms	3.7 ms	2.4 ms	320 x 240	
GeForce GTX 9800+	10.4 ms	6.0 ms	3.8 ms	320 x 240	
GeForce GTX 8800	12.1 ms	7.0 ms	4.6 ms	320 x 240	
GeForce GTX 470	12.7 ms	7.5 ms	5.0 ms	640 x 480	651 Mb
GeForce GTX 460	16.9 ms	9.1 ms	6.4 ms	640 x 480	
GeForce GTX 260	21.0 ms	12.0 ms	7.4 ms	640 x 480	
GeForce GTX 9800+	38.7 ms	21.6 ms	13.6 ms	640 x 480	
GeForce GTX 8800	45.8 ms	26.6 ms	17.3 ms	640 x 480	
GeForce GTX 470	48.3 ms	28.5 ms	18.5 ms	1280 x 960	980 Mb
GeForce GTX 460	64.0 ms	36.7 ms	23.4 ms	1280 x 960	
GeForce GTX 260	76.0 ms	43.7 ms	26.4 ms	1280 x 960	
GeForce GTX 9800+	128 ms	78.3 ms	48.9 ms	1280 x 960	
GeForce GTX 8800	170 ms	98.0 ms	64.0 ms	1280 x 960	

5 Conclusion

In this paper we presented a new framework for an accelerated 3D reconstruction of deformable objects within a multi-view setup. It is based on a new memory management and an enhanced algorithm pipeline that enables efficient and fast reconstruction results and opens up new perspectives for the scalability of time consuming computations within larger camera environments. As a result, a significant increase of frame rates for the volumetric reconstruction of deformable objects can be achieved. We tried to restrict several time consuming memory access tasks within a local area rather than accessing global memory reducing the overall latency of data transfer. Therefore, we adapted the proposed Wedge-Cache algorithm to the parallel nature of a GPU, leading to a reduction of inefficient memory access. As the projection of several viewing rays into a 2D plane is computational expensive resulting in a 2D cone, the use of the Wedge Cache algorithm avoids multiple calculations, as correspondent projections of multiple viewing rays into the image plane do not have to be recalculated again. Further on, we bundle the optimized calculation of silhouette-line intersections into a dedicated CUDA block. This again leads to an increase of locally aligned memory access reducing the number of cache mismatches and avoids longer latencies. Finally, the proposed strategies lead to a higher degree of parallelism, which we can exploit, as the framework has been designed to execute several kernels in parallel (which is only possible on the new generation of graphic boards, e.g. NVIDIA Fermi-GPUs.

References

1. Matusik, W., Buehler, C., Raskar, R., Gortler, S.J., McMillan, L.: Image-Based Visual Hulls. In: SIGGRAPH 2000. Proceedings of the 27th annual conference on Computer graphics and interactive techniques, 369--374. ACM Press/Addison-Wesley Publishing Co, New York (2000)
2. Laurentini, A.: The Visual Hull Concept for Silhouette-Based Image Understanding. IEEE Transactions on Pattern Analysis and Machine Intelligence 16(2), pp. 150--162 (1994)
3. Seitz, S. M., Curless, B., Diebel, J., Scharstein, D., Szeliski, R.: A Comparison and Evaluation of Multi-View Stereo Reconstruction Algorithms. In: CVPR 2006. Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, pp. 519--526 (2006).
4. Hornung, A., Kobbelt, L.: Robust and Efficient Photo-Consistency Estimation for Volumetric 3D Reconstruction. In: Leonardis, A., Bischof, H., Pinz, A. (eds.) ECCV 2006. LNCS, vol. 3952, pp. 179--190. Springer, Heidelberg (2006)
5. Furukawa, Y., Ponce, J.: Accurate, Dense, and Robust Multiview Stereopsis. IEEE Transactions on Pattern Analysis and Machine Intelligence 32, pp. 1362--1376 (2010)
6. Corazza, S., Mündermann, L., Gambaretto, E., Ferrigno, G., Andriacchi, T.: Markerless Motion Capture through Visual Hull, Articulated ICP and Subject Specific Model Generation. International Journal of Computer Vision 87, pp. 156--169 (2010)
7. Graf, H., Yoon, S. M., Malerczyk, C.: Real-time 3D Reconstruction and Pose Estimation for Human Motion Analysis. In: ICIP 2010. Proceedings of the 2010 17th IEEE International Conference on Image Processing, pp. 3981--3984 (2010)

8. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A., Purcell, T. J.: A Survey of General-Purpose Computation on Graphics Hardware. Computer Graphics Forum 26(1), pp. 80--113 (2007)
9. Fung, J., Mann, S.: Using Graphics Devices in Reverse: GPU based Image Processing and Computer Vision. In: ICME 2008. Proceedings of the 2008 IEEE International Conference on Multimedia and Expo, pp. 9--12 (2008)
10. Rossi, R., Savatier, X., Ertaud, J.-Y., Mazari, B.: Real-time 3D Reconstruction for Mobile Robot Using Catadioptric Cameras. In: ROSE 2009. Proceedings of the 2009 IEEE International Workshop on Robotic and Sensors Environments, pp. 104--109 (2009)
11. Weinlich, A., Keck, B., Scherl, H., Kowarschik, M., Hornegger, J.: Comparison of High-Speed Ray Casting on GPU using CUDA and OpenGL. In: Proceedings of the First International Workshop on New Frontiers in High-performance and Hardware-aware Computing, pp. 25--30 (2008)
12. Waizenegger, W., Feldmann, I., Eisert, P., Kauff, P.: Parallel High Resolution Real-time Visual Hull on GPU. In: ICIP 2009. Proceedings of the 2009 16th IEEE International Conference on Image Processing, pp. 4301--4304 (2009)
13. Ladikos, A., Benhimane, S., Navab, N.: Efficient Visual Hull Computation for Real-time 3D Reconstruction using CUDA. In: CVPRW 2008. Proceedings of the 2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, pp. 1--8 (2008)
14. Kim, H., Sakamoto, R., Kitahara, I., Orman, N., Toriyama, T., Kogure, K.: Compensated Visual Hull for Defective Segmentation and Occlusion. In: ICAT 2007. Proceedings of the 17th International Conference on Artificial Reality and Telexistence, pp. 210--217 (2007)
15. Castano-Diez, D., Moser, D., Schoenegger, A., Pruggnaller, S., Frangakis, A. S.: Performance Evaluation of Image Processing Algorithms on the GPU. Journal of Structural Biology 164, pp. 153--160 (2008)