Reconstruction of 3D Human Motion in Real-Time Using Particle Swarm Optimization with GPU-Accelerated Fitness Function

Bogdan Kwolek · Boguslaw Rymut

Abstract In this paper, a novel framework for acceleration of 3D model-based, markerless visual tracking in multi-camera videos is proposed. The objective function being the most computationally demanding part of model-based 3D motion reconstruction is calculated on a GPU. The proposed framework effectively utilizes the rendering power of OpenGL to render the 3D models in the predicted poses, whereas the CUDA threads are used to match such rendered models with the image observations and to perform particle swarm optimization-based tracking. We demonstrate effective parallelization of the particle swarm optimization on GPU. Execution of time-consuming parts of the algorithm on GPU using CUDA-OpenGL significantly accelerates the 3D motion reconstruction, making our method capable of tracking full body movements with a maximum speed of 15 fps. Qualitative and quantitative experimental results on various 4-camera benchmark datasets demonstrate the efficiency and accuracy of our method for real-time motion tracking.

Keywords 3D motion tracking · Particle Swarm Optimization · Parallelization of fitness function

1 Introduction

High quality non-intrusive human motion analysis has received much attention in many areas [20], including computer graphics [38], computer vision [29][31] and multimedia [3][46]. Currently, the need for accurate estimation of human motion is not only limited to the field of animation [28], since in order to achieve immersive experience [44], researchers and engineers are looking for low-cost real-time techniques for 3D reconstruction of users [47] as well as their motions. By understanding human motion, clinicians and biomechanics engineers can improve treatment during rehabilitation [49] as well as improve performance, for example, in high-performance sport [6]. Rapid progress in hardware-supported processing [7,4,48] and 3D capturing devices [16][17] makes possible more precise motion analysis, which is required in hand and body gesture recognition.

The aim of 3D motion recovery is to estimate the position and orientation of each limb of the human body undergoing tracking [20]. Lots of methods have been proposed for articulated object tracking on the basis of typical RGB cameras [29][31]. Discriminative models typically consist in learning a mapping from image measurements to 3D pose, whereas generative approaches rely on density estimation to learn a prior distribution over plausible human motions and poses.

Markerless motion capture (moCap) systems use computer vision algorithms to track 3D motion of subjects without the need for any special suits or markers [31]. Because no special suits, markers or equipment are required, subjects can simply step into the capture scene to deliver required motion patterns or movements. By this means the animators and game developers can capture accurate motion data in less time, and for a much lower total cost. Markerless moCap also makes many clinical and research applications much more practical. The difficulty, however, is implementing robust 3D motion tracking algorithms that provide enough accuracy.

M. Kwolek ✉
AGH University of Science and Technology, 30 Mickiewicza Av., 30-059 Krakow, Poland
E-mail: bkw@agh.edu.pl

B. Rymut
Rzeszow University of Technology, Powst. Warszawy 12, 35-959 Rzeszow, Poland

To accomplish this, typically advanced algorithms are required, especially when the motion tracking or 3D reconstruction of users should be done in real-time [45][44].

In the discriminative bottom–up approaches various features identifying the main body parts are typically sought in the images [17]. Such recognition-based methods predict state distributions directly from body features. That means that their recognition performance strongly depends on discriminative power of the utilized body part detectors. The challenge is to make such approaches robust to wide range of poses needed for general 3D pose reconstruction without modeling most of the 3D articulations. This is since the space of typical human articulation is huge. Despite being simple to understand and fast, their recognition performance can decrease in many real-word situations due to self-occlusions, which are hard to model in 2D. A degradation of the performance in real-word scenarios arises because most of the human body parts do not have the sufficient distinctiveness for their unambiguous detection.

Despite being a natural way to model the appearance of complex articulated structures, the top-down methods are not as extensively studied as bottom-up methods because of huge computational demands to infer the distribution on the high-dimensional hidden states [29][31]. In general, 3D pose estimation in a high-dimensional space of body configurations is intrinsically difficult [20,26]. On the other hand, three dimensional model-based methods are generally more accurate in comparison to methods relying on learned mapping between pose exemplars and image primitives. Moreover, projecting the 3D model into multiple views creates a number of opportunities for better dealing with occlusions and self-occlusions arising in typical daily activities [36].

Recent progress in sensor technologies have enabled the acquisition of dense depth images in real-time. In [16], a real-time tracking of full body motion has been achieved using a time-of-flight camera. The system required 100–250 ms per frame to estimate the joint angles of a 48 degree-of-freedom human model. The more recent Kinect sensors [17] have revolutionized the field of human-computer interaction by enabling 3D motion tracking without body-worn inertial devices or markers. The depth information is employed to estimate a skeletal model of any humans in Kinect's view using a Random Forest classifier, which assigns each pixel as being a body part or background. Although the Kinect's algorithm for pose estimation works quickly and relatively robustly, the lack of an underlying kinematic model and the static, frame-by-frame nature of the pose estimation result in some inaccuracies in posture estimation [11]. In addition, as pointed out in [39], due to the use of a single-depth camera, the recognition accuracy drops significantly when the body parts are occluded. Moreover, Kinect and other single camera-based systems require a person to stay within the device's line-of-sight and cannot track him/her across rooms. In this respect, smart cameras and multi-camera networks are becoming popular [45], among others, due to possibility of covering larger field of view and better exploitation of 3D information for dealing with occlusions. In [2], a multi-Kinect 2 system for 3D reconstruction of moving humans as well skeleton-based motion tracking from multiple depth cameras has been proposed. On a CUDA-enabled NVIDIA GTX 560 the mean reconstruction time of Fourier Transform-based (FT-based) reconstruction takes 163 ms. In a recently proposed method [2], as indicated in Tab. III, for the left/right elbow the mean errors between the estimated angle and the ground-truth are from 14.3 deg. to 25.3 deg. depending on complexity of the sequences.

3D model based methods are frequently employed in motion tracking both in single and multiple camera settings. A framework for 3D model-based tracking of an object in monocular videos has been discussed in [8]. The rendering of the projected 3D model has been realized using Direct3D, whereas the NVIDIA CUDA was utilized to harness the computational power of Graphics Processing Unit (GPU) and to achieve the tracking in real-time. A SIR particle filter has been used to perform tracking on monocular image sequences. Some operations like state estimation, particle propagation and particle resampling were executed on CPU. A speed-up of about 9.5 has been achieved on images of size 96×72 using 900 particles and about 14-fold speed-up using 1296 particles and 128×96 images. In a recent work [38], an algorithm for model-based 3D object tracking in cluttered environments has been proposed. The average overall processing time was about 30 ms and it was achieved on an NVIDIA Geforce GPU, which processed images of 640×480 resolution. Recently, in [33] a model-based method to reconstruct human motions captured outdoors in a multi-camera setup has been proposed. A coarse skeletal pose is estimated in the first stage, and non-rigid surface shape and body pose are jointly refined in the second one. The run time of the second stage is about 5–30 minutes on a CPU and 1–4 minutes on NVIDIA GeForce Titan X, per frame.

In another region-based approach to pose tracking using 3D models [36], an optimization of the 2D projection error is achieved through minimization of an energy function depending on the pose parameters. A precise tracking with an average error of 33.4 mm on the walking part of the Sequence S4 (frames 15–350) of the HumanEva-II database [40] has been reported. However, approximately 15 seconds are needed to process a single frame of the discussed four-views sequence. Corazza et al. reports an average tracking error of about

80 mm on the first 150 frames of the mentioned above sequence [12]. However, as indicated in [14] the frequency at which the video sequences were acquired has considerable influence on the tracking accuracy. In the discussed work, an average error of 38 mm has been obtained on 450 frames of LeeWalk sequence [5], which has been captured with 60 Hz by 4 cameras with a resolution of 644×488 pixels. The average error grows to 46 mm when the tracking was on every third frame, i.e. assuming that the frames were recorded with 20 fps. The discussed results were obtained using a soft partitioning PSO, which is a variant of particle swarm optimization [22]. In recent work [9], the 3D human pose reconstruction is done on the basis of Covariance Matrix Adaptation Evolutionary Strategy (CMAES) based optimization. However, despite remarkable speedup on the GPUs as compared with the CPU naive approach, the computation time of the algorithm with 3000 evaluations of the fitness function on single GPU is far from real-time. Moreover, instead of rasterizing the triangle meshes, the algorithm calculates the projection of their vertices and draws rectangular patches around them. The result of such an operation are binary images that are matched with binary silhouettes (extracted by background subtraction) using XOR operation. In consequence, it ignores edge information and does not use edge distance map, which is commonly utilized to estimate the proximity of a pixel to the edge.

In [47] a 3D motion tracking is realized in eight camera setup using a volumetric reconstruction. The tracking has been accelerated by implementing some time-consuming steps using CUDA. The method is capable of tracking the body movements with a maximum speed of 9 fps. An average error of about 31.3 mm has been reported on LeeWalk sequence using edge and silhouette based likelihoods. However, only the first 150 frames were used in the evaluation, as most likely the tracking became less robust in the remaining part of the sequence, see also error plots in [14]. Considerable attention has been devoted in recent years to interactive virtual try-on applications, which track the user's position and pose to augment him or her with garments. In [19] a system consisting of ten synchronized cameras is mounted in a virtual dressing room of size 2×3. A cabin with green walls has been used to simplify background extraction, which was needed in the image-based visual hull in order to render users and clothes from arbitrary viewing angles. Given, the extracted person in the depth maps, the skeleton was extracted using the Open-NI API. The images were recorded at 15 Hz. However, the processing time of the algorithm of about 120 ms is far from real-time.

Currently, real-time reconstruction of 3D human motion is of great importance. However, the existing applications for markerless motion capture are far from real-time or at least they can process the frames at time close to real-time, and only are able to deliver promising accuracies using sequences that were recorded with typical camera rates. In this paper we present an approach for on-line markerless reconstruction of human motion in a four camera system. We demonstrate how to achieve the 3D motion tracking with an average accuracy of 50 mm with a frequency close to 15 Hz using a PC with a graphics card. To achieve this we take advantages of CUDA-OpenGL processing, where the model rendering being the most time consuming part of the tracking is realized by graphics card's hardware. We then demonstrate that owing to the use of CUDA-OpenGL interoperability the tracking time can be shortened more than twice in comparison to GPU-accelerated processing using CUDA only. The contribution of this paper is an advanced framework for real-time markerless human motion tracking on the basis of OpenGL-based 3D model rendering and projecting it to the camera views. To the best of our knowledge, this is a new application of OpenGL in a novel approach to real-time human motion reconstruction. The most relevant work is [8,9]. The difference between our work and [8] is that we employ a multi-camera system (entirely calibrated and synchronized four camera system, which is calibrated and synchronized with moCap). Another difference is that in our system the 3D model rendering can be easily configured and changed through shader programs. Last but not least, it is worth noting that the problem of 3D tracking of full body (in a multi-camera system) is different from monocular hand tracking. In contrast to [9], which projects vertices and then draw patches around them, we rasterize silhouette and edges, as well as we employ the edge distance in the fitness function. Our approach is different since we employ OpenGL and shaders.

In our previous work [24], a CPU-based approach to real-time full body motion tracking has been proposed. It has been demonstrated experimentally that an algorithm termed as annealed PSO with resampling (RAPSO) can perform the tracking with 15 fps on two PC nodes, each of them with two multi-core CPUs with hyper-threading, connected by 1 GigE links. In order to utilize the computational power of two XEON, 6-core CPUs, a multi-swarm PSO consisting of 8 swarms has been employed. We demonstrated experimentally that on P1S image sequence, which is used in this work, we can achieve 5-fold speed-up, and full-body tracking with the average error equal to 50.6 mm. In [25] we demonstrated that in 3D-model based human pose tracking a considerable shortening time of calculation of the fitness function can be achieved through the use of OpenGL-based rendering of the model in the requested poses. We measured the execution times of the main components of the fitness function using CPU, CUDA, CPU-OpenGL and CUDA-OpenGL. We demonstrated that the

calculation of the fitness score using CUDA-OpenGL is up to 40 times faster in comparison to calculation it on a multi-core CPU using OpenGL-based model rendering. We demonstrated that good tracking accuracies can be obtained on LeeWalk and Human Eva I datasets using the introduced OpenGL-based rendering module. In the discussed work the PSO was not parallelized and completely integrated with the rendering module. In this work we present how the synchronous particle swarm optimization using OpenGL-accelerated fitness function has been decomposed on GPU. It is worth noting that the problem of parallelization of PSO is not an easy task and considerable research has been devoted to it [43]. We present in detail the motion prediction, memory mapping as well as CUDA-OpenGL interoperability. We explain in detail vertex pipeline, silhouette edge pipeline and rasterizing pipeline. We present comprehensive evaluations on LeeWalk dataset as well as video sequences that were used in [24].

The remainder of this paper is organized as follows. In Section 2 we outline the architecture for the system and discuss its main ingredients. Afterwards, in Section 3 we discuss computing on GPU using CUDA-OpenGL interoperability. In Section 4 we detail parallelization of particle swarm optimization on GPUs. Section 5 details acceleration of the calculation objective function through OpenGL-based rendering of the 3D model in the required pose. Experimental results are discussed in Section 6. Section 7 contains concluding remarks.

2 Architecture and Main Ingredients of the System

In general, the methods based on volumetric reconstruction require larger number of the cameras in comparison to methods relying on matching a projected 3D model with the captured images. Moreover, such methods typically assume that a performer occupies a central part of the scene observed by multiple cameras. Bearing in mind that in applications for walking motion analysis, which are increasingly being used in biomechanics and rehabilitation, typically a larger scene to be covered by the cameras is needed, as well as that the distance between the performer and the cameras changes, we focus on a setup with four perpendicular RGB cameras and motion reconstruction on the basis of matching the projected 3D model with the images. Having on regard that for larger distances between the person and the camera, the person will occupy smaller portions of the image, there is no rationale to construct a detailed wireframe to model the human. As a result, our human model consists of several component objects, which are linked by joints. Such a coarse human models are frequently used in 3D motion reconstruction [5][40][35].

Figure 1 depicts a diagram of the system for real-time human motion tracking on the basis of OpenGL-based 3D model rendering and then projecting it to the camera views. The system consists of three main ingredients, namely, OpenGL-based 3D model rendering, GPU-accelerated tracking, and image processing. The OpenGL module is universal since it can deliver the rendered models for all kinds of predict-evaluate trackers, like particle filters [13][5] and particle swarm optimization [14][35]. The input for this module are transformation matrices, which are then used to obtain the 3D model in the required poses. The output is a collection of the rendered models in the requested poses. The rendered images contain information both on the silhouette and the edges. The number of rendered images is equal the number of the particles times number of the cameras, where each particle represents the model in a hypothetical (predicted) pose. In each frame the tracking algorithm, see also CUDA part on Fig. 1, calculates the state vectors of the potential poses, which are then expressed in the form of a set of transformation matrices for each of the considered poses. The rendered images are matched with the images containing the person's silhouettes and edges. On the basis of an objective function the best state is estimated, which is then used to calculate the potential states for the person's pose in the next frame.

CPUs are still the most often used hardware for image processing, given their versatility and tremendous processing power [32]. On the other hand, image processing algorithms are perfect candidates for computing on GPU, since many image processing operations have high inherent parallelism, which is often accomplished through per-pixel operations [15]. Many researches confirmed this by showing GPU acceleration on several image processing algorithms [10]. A recent study [32] reports a thirty-fold speed-up for low-level algorithms and up to ten-fold speed-up for high-level functions, which bring larger overheads and typically require many steps that are not easy to parallelize.

In our system the CPU is responsible for acquiring images from four calibrated and synchronized cameras as well as processing them. The advantage of such an approach is that the image processing is executed in parallel with the acquisition of the images [34]. The images with the extracted silhouettes and the edges are transferred asynchronously from the host memory to the GPU memory. Data transfers from the host (CPU) to the device (GPU) are done using PCIe 3.0 x16 bus with peek transfer rate of about 16 GB/s in single direction. Thus, the memory transfer overheads for uploading the preprocessed images from four cameras are negligible. By using

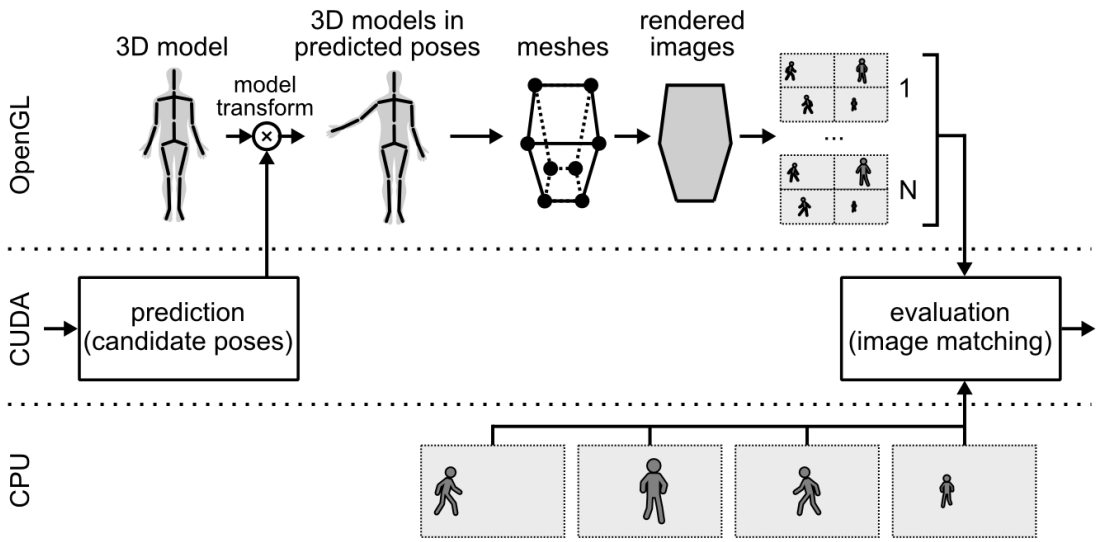

Fig. 1 Diagram of the system for real-time markerless motion reconstruction.

CUDA we turn the GPU into a powerful parallel processor, whereas by using OpenGL we utilize the same GPU hardware to render the required images. Because CUDA and OpenGL both run on GPU and share data through common memory, the CUDA-OpenGL interoperability is very fast [25].

3 Computing on Graphics Processor Units Using CUDA-OpenGL Interoperability

A GPU is a dedicated processor, which offloads 3D graphics rendering workload from the CPU. The image synthesis is implemented as a pipeline of specialized stages, which is usually called graphics pipeline [27]. The input to such a pipeline is a wireframe being composed of a set of primitives, which are defined by a group of one or more vertices. Over the past two decades, this basic graphics pipeline has transitioned steadily from fixed-function graphics architecture into powerful programmable co-processing units capable of performing general purpose computing (GPGPU). The replacement of fixed-function dedicated logic by programmable processors logic while maintaining the basic 3D graphics pipeline organization was one of the key developments in computer graphics. As a result of such evolution the current programmable processors consist of:

- Vertex processor, which goal is to transform each input vertex to data required by the next graphics pipeline stages.
- Geometry processor, which is accountable for processing a mesh at primitive level and producing vertices and attributes to define primitives.
- Fragment processor, which is responsible for determining the color of each fragment.

The three different shaders were merged into one unified shader model with a consistent instruction set across all three processor types. As a result, vertex, geometry, and pixel/fragment shaders became threads, running different programs on the programmable cores. These cores on the NVIDIA platform are called CUDA cores, and a streaming multiprocessor (SM) consists of many CUDA cores. To program the GPU pipeline an application ought to set the configuration of fixed-function stages and supply shader programs, which run within environments associated with each programmable stage. The discussed above GPU pipeline provides the following key benefits:

- Programming simplicity. The high-level, graphics-specific pipeline abstraction makes the programming easy.
- Versatility. Programmable stages allow the GPU pipeline to accommodate a wide variety of rendering techniques.
- High performance. The pipeline abstraction and accompanying shader programming model provide both task parallelism across stages and data-parallelism within the stages.

The parallel programming languages like CUDA or OpenCL allow us to write software that runs as parallel program on GPU's programmable cores. They permits full exploitation of the computational power of GPUs, which are excellent at graphics and visual computing since they were specially designed for these applications.

Thanks to the use of the GPU as both a graphics processor and a computing processor at the same time, we achieved a highly effective computing framework for 3D motion reconstruction in real-time. More specifically, owing to programming the GPU by the graphics API and utilizing graphics pipeline to perform nongraphics tasks, like particle swarm optimization, and thanks to OpenGL permitting the use of the same GPU hardware to render the images needed for the calculating of the objective function, we achieved considerable shortening of the tracking time. That means that we effectively utilize the rendering power of OpenGL to render the 3D models in the requested poses, whereas the CUDA threads match such rendered models with the image features and perform the tracking. Because CUDA and OpenGL both run on GPU and share data through common memory, the CUDA-OpenGL interoperability is very fast in practice [41,25]. In the motion tracking methods relying on matching the projected models with the camera images the most significant computational overheads are associated with the rendering of the 3D models [23,25]. Thus, the OpenGL-based 3D model rendering can shorten the whole motion tracking time considerably.

4 Parallel Particle Swarm Optimization for 3D Motion Reconstruction

At the beginning of this Section we discuss how the particle swarm optimization algorithm is executed in parallel on GPU. Afterwards, in two subsections we discuss the motion and the evaluation steps, which involve mapping OpenGL resources to the CUDA. In the final part of Subsection 4.1, 4.2 and 4.3 we outline organization of threads and blocks.

4.1 Parallel Particle Swarm Optimization for Object Tracking

Particle swarm optimization (PSO) [22] is a global optimization method and also an heuristic optimization algorithm, which is based on swarm intelligence. It is derivative–free, stochastic and population–based computational method, which demonstrated a high optimization potential in unfriendly non–convex, non–continuous spaces. The swarm consists of a set of particles, and each swarm member represents a potential solution of an optimization task. The particles are placed in the search space and move through such a space according to rules, which take into account each particle's personal knowledge and the global knowledge of the swarm. Each particle is attracted to some extent to the best location it has discovered so far, and also to the best location any member of the swarm has found. Every individual moves with its own velocity in the multidimensional search space, determines its own position and calculates its fitness using an objective function $f(x)$. On the basis of the fitness function the particles determine the best locations as well as the global best location. Any member of the swarm follows simple position and velocity update equations; yet as particles interact, the collective behavior arises, and the interactions between the particles lead to the emergence of global and collective search capabilities, which allow the particles to gravitate towards the global extremum.

The ordinary PSO algorithm begins by creating particles at initial locations, and assigning them initial velocities [22]. Afterwards, it determines the value of the objective function at each particle location, as well as determines the best function value and the corresponding best location. It determines new velocities, based on the current velocity, the particles's individual best locations, and the best location of the entire swarm. The best location can be:

- a global best that is immediately updated when a new best position is found by any particle in the swarm
- neighborhood best where only a specific number of particles is affected if a new best position is found by any particle in the sub-population

It then iteratively updates the particle locations and velocities, and optionally it updates the neighbors if the neighborhood best topology has been chosen. However, such a topology slows down the convergence speed. Taking into account the faster convergence the topology with the global best has been selected for parallel implementation and it is discussed below in more detail.

At the beginning of the optimization, each particle is initialized with a random position and velocity. While seeking the best fitness every individual i is attracted towards a position, which is affected by the best position $p^{(i)}$ found so far by itself and the global best position p_{g} found by the whole swarm. In every iteration k, each particle's velocity is first updated based on the particle's current velocity, the particle's local information and global information discovered by the entire population. Then, each particle's position is updated using the velocity. In the ordinary PSO, the position and velocity of particle are calculated as follows:

$$v_j^{(i)} \leftarrow w v_j^{(i)} + c_1 r_{1,j}^{(i)} (p_j^{(i)} - x_j^{(i)}) + c_2 r_{2,j}^{(i)} (p_{\mathrm{g},j} - x_j^{(i)}) \tag{1}$$

$$x_j^{(i)} \leftarrow x_j^{(i)} + v_j^{(i)} \tag{2}$$

where w is the positive inertia weight, $v_j^{(i)}$ is the velocity of particle i in dimension j, $r_{1,j}^{(i)}$ and $r_{2,j}^{(i)}$ are uniquely generated random numbers with the uniform distribution in the interval [0.0, 1.0], c_1, c_2 are positive constants, $p^{(i)}$ is the best position that the particle i has found so far, p_{g} denotes the best position that was found by any member of the swarm.

Equation (1), which updates the particle velocity has three main components. The first component that is frequently referred to as inertia models the particle's tendency to keep it moving in the same direction it was moving previously. In fact it controls the exploration of the search space. The second component, called cognitive, attracts the particle towards the best position $p^{(i)}$ that was found formerly. The last component is referred to as social and it pulls the particle towards the best position p_{g} found by any particle. The fitness value that corresponds to $p^{(i)}$ is called local best $p_{\mathrm{best}}^{(i)}$, whereas the fitness value corresponding to p_{g} is defined as g_{best}.

Algorithm 1 shows the pseudo-code of the ordinary PSO algorithm. The discussed algorithm is synchronous

Algorithm 1 Synchronous PSO

```
Initialize randomly x^(i), v^(i)
repeat
    for each particle i do
        Evaluate the fitness function f(x^(i))
        if f(x^(i)) < p_best^(i) then
            p_best^(i) = f(x^(i))
            p^(i) = x^(i)
        end if
        if f(x^(i)) < g_best then
            g_best = f(x^(i))
            p_g = x^(i)
        end if
    end for
    for each particle i do do
        Update position using (1) and (2)
    end for
until stopping criterion is not satisfied (e.g. maximum number of iterations or a sufficiently good fitness value is not attained)
```

since all particle velocities and positions are updated at the end of each optimization iteration. The algorithm presented above can be used for solving static optimization problems. The motion tracking can be attained by dynamic optimization and incorporating the temporal continuity information into the ordinary PSO. Consequently, it can be achieved by a sequence of static PSO-based optimizations, followed by re-diversification of the particles to cover the potential poses that can arise in the next time step. The re-diversification of the particle i can be obtained on the basis of normal distribution concentrated around the best particle location p_g in time $t-1$, which can be expressed as: $x^{(i)} \leftarrow \mathcal{N}(p_g, \Sigma)$, where $x^{(i)}$ stands for particle's location in time t, Σ denotes the covariance matrix of the Gaussian distribution, whose diagonal elements are proportional to the expected movement.

Figure 2 shows how the synchronous PSO has been decomposed for a parallel execution on the GPU. The algorithm consists of eight steps: RNG (calling random number generator), initialization, motion, rendering, evaluation, compute p_{best}, and compute g_{best}. The motion and rendering steps are specific for 3D motion tracking and are discussed in more detail in Subsection 4.2 and 4.3. The PSO algorithm is now widely used in 3D motion tracking [24,3,35,42]. A survey of GPU-based implementations of PSO, including parallel implementations for 3D motion tracking can be found in [43]. As demonstrated in [23], a considerable shortening of the motion reconstruction time can be obtained through implementing both the PSO and rendering of the 3D model cuboids on the GPU. The most time consuming part of the motion tracking is rendering of the 3D model and matching of such a projected model with image observations.

At the beginning of each frame the algorithm generates uniformly distributed pseudorandom numbers. It generates $2D(K+1)N$ pseudorandom values, where D denotes dimension, K is the number of iterations in PSO, whereas N stands for the number of the particles. The random numbers are used in the initialization step

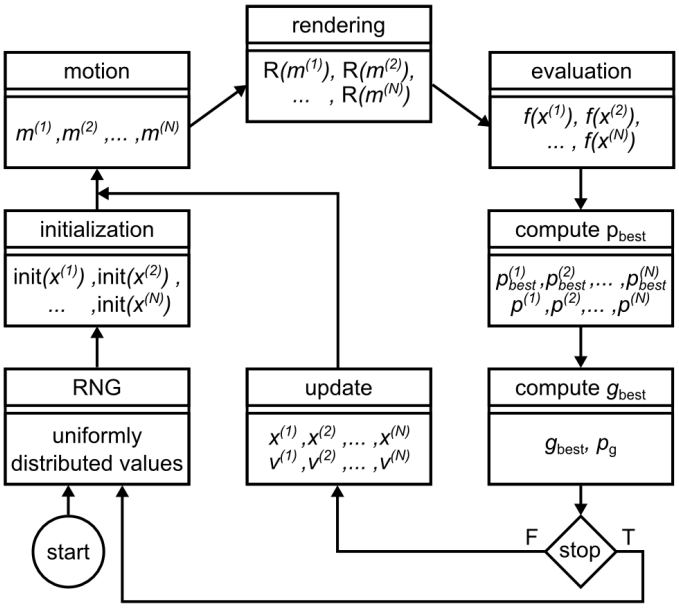

Fig. 2 Decomposition of synchronous particle swarm optimization algorithm on GPU.

to generate normally distributed pseudorandom numbers needed for the re-diversification of the particles as well as to set the r_1 and r_2 numbers, which are needed in each iteration of the optimization, see also the update step. In the p_{best} step the algorithm executes ND threads in $B = \lceil ND/W \rceil$ blocks, where W denotes the number of the GPU cores. This means that $N_B = \lceil N/B \rceil$ threads determine the values p_{best} of N_B particles, and then calculates N_B values of the best positions $p^{(i)}$. In the g_{best} step the algorithm executes $N_B = \lceil N/2 \rceil$ threads in a single block, and they determine the g_{best} and p_{g} values. In the update step the ND threads are executed in $\lceil ND/W \rceil$ blocks, and each thread updates a single element of the state vector of a single particle.

4.2 Motion Prediction

Human body models are often used to describe both the kinematic properties of the body and the body shape. Figure 3 depicts the utilized human skeleton and the corresponding hierarchical structure of joints. The pelvis is the root node in the kinematic chain and it determines the global orientation and position of the model. The pose is encoded on the basis of the position and orientation of the root segment in the world, and a set of relative joint angles, which represent the orientations of body parts with respect to their parents along the tree. The model has 31 degrees of freedom (DOFs) and the body segments are linked to the parent segments by either 1-DOF or 3-DOF rotational joints. The relative rotation between neighboring parts is described by Euler angles.

Part-based shape models represent each body part as a rigid shape attached to a joint of the kinematic tree. Such shape models do not deform during motion and they rigidly move jointly with the skeleton segment. These models are frequently used to achieve articulated human body pose estimation and tracking [13][5][40]. Our skeleton-driven 3D body model consists of ten rigid body segments that are represented by truncated cylinders. The state of each truncated cylinder includes base radius, top radius, length and a transform matrix. The sizes of each cylinder's length, base radius and top radius were attuned manually. The 4×4 transform matrices are determined by the number of degrees of freedom of the joints and they establish the rotation, translation and scale of the corresponding truncated cylinder.

Each bone has associated with it a local coordinate system. On the basis of the predicted state vector the algorithm calculates local transformation matrices, see Fig. 4. A local transformation matrix expresses the displacement/rotation of the body segment in its local frame of reference. The world transformation is the transformation of the bone in world-space taking into account parent transformations.

As we can see on Fig. 4, the number of the threads is equal the number of the particles. The threads are grouped into blocks. The number of the blocks is equal to the number of the multiprocessors available on the

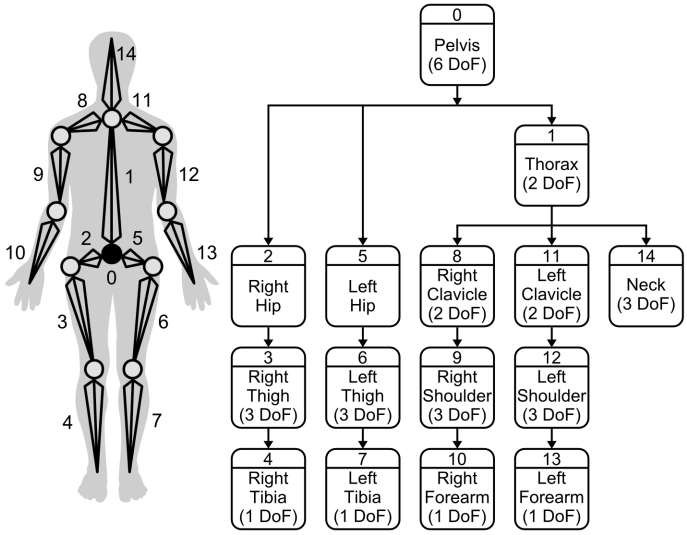

Fig. 3 Human skeleton and the corresponding hierarchical structure.

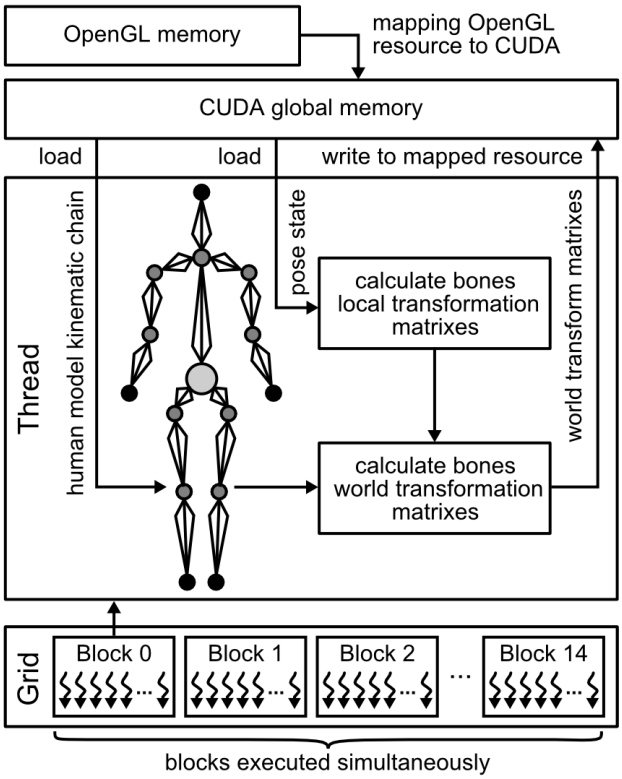

Fig. 4 Calculation of the world transformation matrices on the GPU.

utilized GPU. The kinematic chain is stored in the shared memory. Given the local transformation matrices, the algorithm traverses the kinematic chain and calculates the world transformation matrices of the bones. The OpenGL memory is mapped to CUDA memory, which stores the resulting world transformation matrices, see Fig. 4. In Section 5 we show how the OpenGL makes use of such matrices to pose the 3D models, to render them, and then to generate the rasterized images. In the subsequent subsection we discuss how these rasterized images are employed in the matching, which is performed by CUDA threads.

4.3 Calculation of the Objective Function

The majority of algorithms for 3D motion tracking are based on minimizing an objective function that measures how well the 3D model fits the image observations. The matching function is often based on degree of overlap between the observed image silhouette and the silhouette of the projected model. The edges are also used frequently because they allow more precise localizing of individual body parts [13][5][40]. They also provide some invariance to viewpoint and lighting conditions as well as complement the silhouette cues by providing internal edges of the body areas. The silhouettes and the edges are extracted on the CPU. The thresholded distance transform is calculated to guide the model fitting to the image edges. The images with the extracted silhouettes and the edges are transferred from the host memory to the GPU memory, see Fig. 1. The GPU memory is then mapped to the texture memory.

The image containing the models rendered by the OpenGL is also mapped to the texture memory, see Fig. 5. This way both the reference images, which are extracted on the basis of image observations, and the images with the predicted poses, which are rendered by OpenGL are stored in the texture memory. The rendering result of each camera is stored in single byte, where information about the silhouette is stored in first seven bits, whereas the edge is coded in the eighth bit. We reserved seven bits for the silhouette for future extensions of the algorithm, which will use information about human limbs. The rendering results of four cameras are stored in RGBA pixels with 8 bit per channel, i.e. per camera.

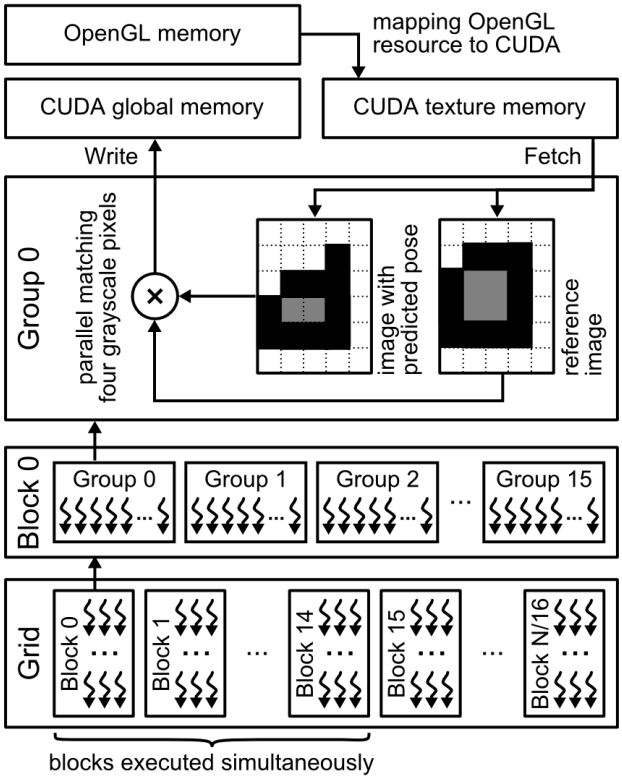

Fig. 5 Calculation of the objective function on GPU using 3D models rendered by OpenGL.

The matching kernel executes $N/16$ blocks and each block is split into 16 groups. Each group processes two regions of interest, which contain the reference and predicted silhouettes (silhouettes, edges and distance transform values). The value of each pixel is stored on four bytes, which hold the values from four cameras.

4.4 Objective Function

The overlap degree between the extracted silhouette in the camera images and the silhouette of the projected model into the camera images $i = 1, \ldots, 4$ is calculated as follows:

$$f_1(x) = 0.5 \frac{\sum_{i=1}^{4} o(i)}{\sum_{i=1}^{4} r^{(i)}} + 0.5 \frac{\sum_{i=1}^{4} o(i)}{\sum_{i=1}^{4} c^{(i)}} \tag{3}$$

where o denotes the number of pixels on the rendered image, which are shared by the observed silhouette and the rendered silhouette, c is the number of pixels in the rendered silhouette, whereas r stands for the number of pixels in the observed silhouette. The edge fitness between the extracted edges in the camera images and the edges of the projected model into the camera images has been calculated in the following manner:

$$f_2(x) = \frac{\sum_{i=1}^{4} e^{(i)}}{\sum_{i=1}^{4} d^{(i)}} \tag{4}$$

where e denotes number of pixels in the projected edges, whereas d denotes the sum of the values of the distance function at positions, where the model edges are projected. Such locations express the proximity of projected edge pixel to the nearest edge in the observed image. The fitness function has been calculated on the basis of the smoothed product:

$$f(x) = 1.0 - f_1(x)^{w_1} f_2(x)^{w_2} \tag{5}$$

where w_1 and w_2 are smoothing factors, which were set to $w_1 = 0.7$ and $w_2 = 1.0 - w_1$. Their values were determined experimentally and set once for datasets utilized in this work. A decrease of the w_1 value about 10% and then update the w_2 value accordingly, does not change the tracking error noticeably.

5 OpenGL-accelerated Rendering of 3D Human Models in the Requested Poses

At the beginning of this Section we overview CUDA-OpenGL interoperability. Afterwards, we explain how we effectively utilized the OpenGL to accelerate the rendering of 3D human models in the requested poses.

5.1 CUDA-OpenGL Interoperability

Open Graphics Library (OpenGL) is an application programming interface for graphics hardware that allows generating high-quality images of 3D objects [37]. It defines a logical graphics processing pipeline, which is mapped onto the GPU hardware and processors, together with programming models and languages for the programmable pipeline stages. The API consists of several hundred procedures and functions that are syntactically similar to C language. In addition to being language-independent, OpenGL is also platform-independent and mainly deals with rendering and managing context to external libraries such as GLUT and GLFW.

OpenGL is only concerned with processing data stored in GPU memory. It does not support typical C-like pointer type memory allocation. Instead, OpenGL stores data in abstract generic buffers that are called buffer objects. These buffers objects can have various types, like texture objects containing texture data, vertex array objects holding a set of vertices, and shader objects containing shader programs. Each object type has a corresponding set of commands that manage objects of particular type. In the process of rasterization and clipping the 3D primitives are projected onto a 2D plane according to a projection matrix that is defined by the camera. The graphics pipeline or rendering pipeline refers to a sequence of steps that are needed to create a 2D raster representation of a 3D object. Some of these stages are programmable through shader programs and some are fixed-function stages.

CUDA functions for mapping resources from OpenGL into the address space of the CUDA permit an interoperability between both APIs. Owing to the CUDA-OpenGL interoperability we can avoid back and forth data transfer, and carry out all processing required for 3D model rendering and matching without additional transfer overheads [41]. Because CUDA and OpenGL both run on GPU and share data through common memory, the CUDA-OpenGL interoperability is very fast in practice. As indicated in [25], time needed for mapping the rendering result is equal to 0.4 ms, and the overall execution time required for determining the objective function values in the single iteration of PSO is equal to 4.4 ms, for a swarm consisting of 96 particles and measurements done for LeeWalk sequence.

5.2 OpenGL-accelerated Rendering of 3D Human Models for Subsequent Use by CUDA

Recent versions of OpenGL support a programmable graphics pipeline designed to exploit the parallel architectures available in modern GPUs [37]. Figure 6 illustrates the OpenGL graphics pipeline. This programmable pipeline permits data to be stored into the graphics device memory, and instructs (through shader programs) the GPU to operate on the data in order to perform the geometric transforms, lighting transforms, and any other desired calculations on data in device memory before it is placed into video RAM (VRAM) memory for display or put back in the off-screen frame-buffer.

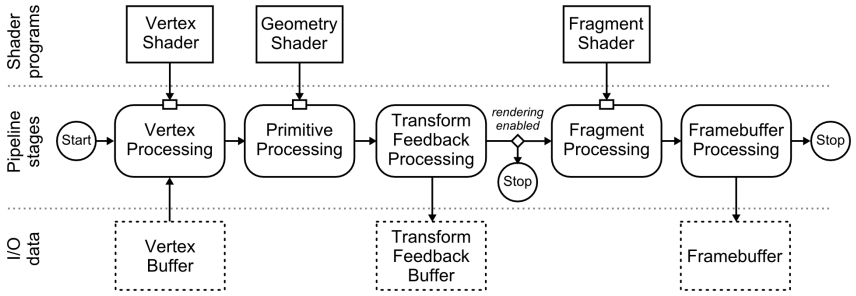

Fig. 6 Block diagram of the OpenGL pipeline.

A shader program is a user-defined program, which purpose is to execute one of the programmable stages of the rendering pipeline. Shaders for modern GPUs in OpenGL programs can be written using GLSL syntax, which is a C like language. The data parallel programming is done by creating vertex, geometry and fragment shader programs. A vertex shader program is executed on all vertex elements in an array containing vertex data, and can thus be executed as kernels in parallel across multiple SIMD engines, where each stream processing unit (SPU) operates on an individual vertex. Each geometry shader processes each incoming primitive, returning zero or more output primitives. Similarly, each fragment shader is executed on all pixels in an array containing pixel data, and they are also executed as kernels in parallel across multiple SIMD engines, where each SPU operates on an individual pixel. The transform feedback, which is located after all of the vertex-processing stages and directly before primitive assembly and rasterization, see Figure 6, captures vertices as they are assembled into primitives and allows some or all of their attributes to be recorded into buffer objects for use as input to future commands.

The rendering of the 3D model has been realized in three steps, in which the following pipelines were executed:

- vertex pipeline
- silhouette edge pipeline
- rasterizing pipeline

The vertex pipeline is responsible for projection and transformation of the model vertices into vertices in image coordinates using the world transformation matrices. Owing to such approach we avoided redundant transformation of the vertices for extracting the edges of the silhouettes and rasterizing the silhouettes. This means that the vertices in the image coordinates are extracted once, and then used in two pipelines responsible for extracting edges of the silhouettes as well as rasterizing the silhouettes.

The vertex pipeline for rendering the 3D model built on the truncated cones is depicted in Fig. 7. Each conical frustrum is represented by the center and radii of the circular bottom and center and radii of the circular top. On the basis of such a representation as well as calibration parameters of the Tsai camera model, the shader program generates four vertices of the trapezium perpendicular to the camera axis. The trapezium vertices were generated using the billboard method [1].

The OpenGL pipeline depicted on Fig. 8 has been executed twice to extract the edges of the 3D model as well as to rasterize the silhouette of the model. As one can notice, both pipelines consist of vertex shader and fragment shader. The silhouette edge pipeline operates on line primitives, whereas the rasterizing pipeline operates on triangle primitives.

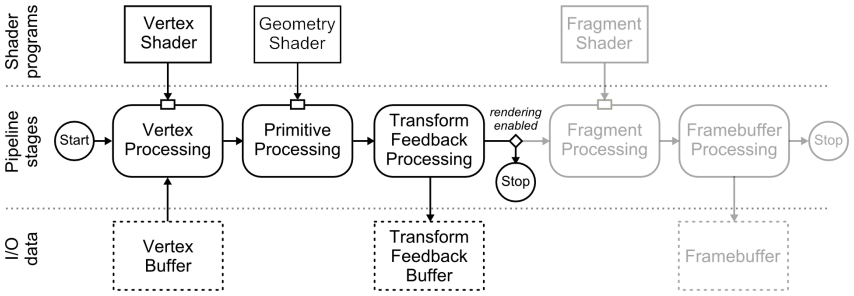

Fig. 7 Vertex pipeline.

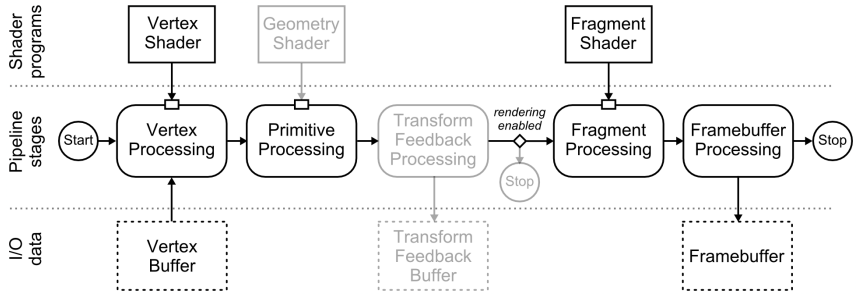

Fig. 8 Pipeline responsible for extracting the edges of silhouette and rasterizing the silhouette.

6 Experimental Results

The algorithm has been implemented in C++ with its core parts implemented in CUDA C language. The system has been evaluated experimentally on a PC with Intel Xeon X5690 3.46 GHz CPU, 8 GB RAM, and NVidia GeForce GTX 780 Ti graphics card. The GeForce GTX 780 Ti is based on Titan that has GK110 Kepler GPU architecture. The new key feature of the Kepler generation is the new streaming multiprocessor architecture called SMX [30]. The GK110 is constructed from up to 15 SMXs. Each SMX consists of 192 CUDA cores, 64 double-precision units, 32 SFUs (special function units) and 32 load and store units. The context of OpenGL 4.2 has been created by GLFW library, whereas the CUDA context has been created using CUDA Driver API.

The performance of algorithm has been evaluated on two datasets with walking subjects. The 3D motion tracking has been initialized on the basis of moCap data. We run the experiments on portion of a circular walking sequence [5] that contains a full 180 turn of the performer. The images in the discussed LeeWalk sequence are 480 pixels high and 640 pixels wide. In addition to images acquired by the cameras, the sequence contains binary images with extracted silhouettes, calibration data and moCap data [40]. In the quantitative evaluation we utilized an average distance between 15 markers corresponding to joints and limb endpoints. The algorithm has also been evaluated on P1S and P2S straight walking sequences [24]. The color images have size 480×270 and were acquired by four synchronized and calibrated cameras. Each pair of the cameras was approximately perpendicular to the other camera pair. The silhouettes were extracted from the input images by statistical background subtraction with a Gaussian mixture model [50,32]. The image edges were detected by the gradient operator and they were masked by the detected silhouettes. A Vicon moCap system using reflective markers and consisting of sixteen cameras has been employed to recover the 3D location of the markers. In the quantitative evaluation of the algorithm we utilized an average distance between 39 markers. The LeeWalk sequence was recorded with 60 Hz, whereas the P1S and P2S sequences were recorded with 25 Hz.

Table 1 presents the average times of 3D motion reconstruction, which were obtained on CPU and GPU. The presented times do not include the time needed for the image processing. This means that the edges, silhouettes and the distance maps were extracted off-line and then they were utilized in the performance evaluations discussed below. The computation times on the GPU include time for the data transfer from the CPU to the GPU. The tracking times were obtained on LeeWalk sequence [5] as well as on the P1S and P2S image sequences. A software rendering algorithm, which has been used in our previous work [24,34] has been utilized in CPU and CUDA implementations. At the beginning, the rendering algorithm performs the projection and transformation of the model vertices into vertices in image coordinates using the world

transformation matrices. Each trapezoid of the model undergoes a projection onto 2D image of each camera using parameters of Tsai camera model. The image of the trapezoid is obtained by projecting the corners and then rasterizing the triangles composing the trapezoid [34]. When rendering, back-face culling is executed, which searches for all normals that point away from the viewpoint and skips the associated faces. Triangle filling and edge determining is achieved using Bresenham algorithm. Through rasterizing all truncated cones we attain image representing the 3D model in a given configuration. The tracking times on the CPU were achieved by a multithread program written in C/C++, which has been executed on the available CPU cores. As we can notice, the tracking time of the program implemented in CUDA and executed on the GPU is far shorter. In general, the CUDA/CPU speed-up is bigger for larger number of the particles. For a PSO-based tracker consisting of 100 particles and executing 10 iterations the CUDA/CPU speed-up is slightly less than two times. In the subsequent paragraph we show that a PSO consisting of one hundred particles and executing ten iterations allows us to achieve tracking with good enough accuracy. As we can observe, the time needed for 3D reconstruction of the motion on the LeeWalk sequence is longer. The larger processing overhead is due to different aspect ratio of the cameras in LeeWalk and P1S/P2S sequences, i.e. different sizes of the silhouette on the images. As a result, larger number of pixels is processed in the rendering and matching of LeeWalk images.

Table 1 Average tracking time [ms] for single frame.

	# part. (10 it.)	CPU [ms]	CUDA [ms]	$\frac{CUDA}{CPU}$ speedup	CGL [ms]	$\frac{CGL}{CUDA}$ speedup
P1S	100	138.7	109.6	1.3	62	1.8
	300	368.0	122.5	3.0	67.5	1.8
	500	607.4	175.2	3.5	72.7	2.4
P2S	100	138.0	106.2	1.3	61.1	1.7
	300	398.5	132.8	3.0	63.7	2.1
	500	644.1	174.1	3.7	71.8	2.4
LeeWalk	100	271.6	168.8	1.6	75.7	2.2
	300	965.3	251.9	3.8	90.9	2.8
	500	1852.0	430.1	4.3	121.7	3.5

As we can notice in Tab. 1, the speed-up of the OpenGL accelerated algorithm (CGL) over algorithm running on GPU without hardware rendering (CUDA) is considerable. Moreover, as we can observe, for five times larger number of the particles the execution time for LeeWalk sequence is only about 1.7 times longer, see also CGL computation times for 100 and 500 particles of P1S and P2S sequences. The tracker running on GPU without OpenGL support needs about 2.8 more time for PSO consisting of five times larger population of the particles. This contrasts with algorithm running on CPU, which execution time is roughly proportional to the number of the particles, see Tab. 2, which presents the ratio of the tracking times for 500 and 100 particles. This means, that the CUDA-OpenGL interoperability allows us to obtain better efficiency of parallel computations for larger number of the particles.

Table 2 Ratio of tracking times for 500 and 100 particles.

	CPU	CUDA	CGL
P1S	4.4	1.6	1.2
P2S	4.4	1.6	1.2
LeeWalk	6.8	2.6	1.6

Figure 9 illustrates some qualitative results, which were obtained on LeeWalk sequence. The better is the overlap of the projected model on the person undergoing tracking, the better is the motion tracking. The discussed results were obtained using CUDA-OpenGL interoperability and PSO consisting of 100 particles and executing 10 iterations. As we can observe, the PSO with 1000 calls of the objective function permits quite

precise tracking, which is reflected by good overlap between the projected models and the silhouettes in all camera views.

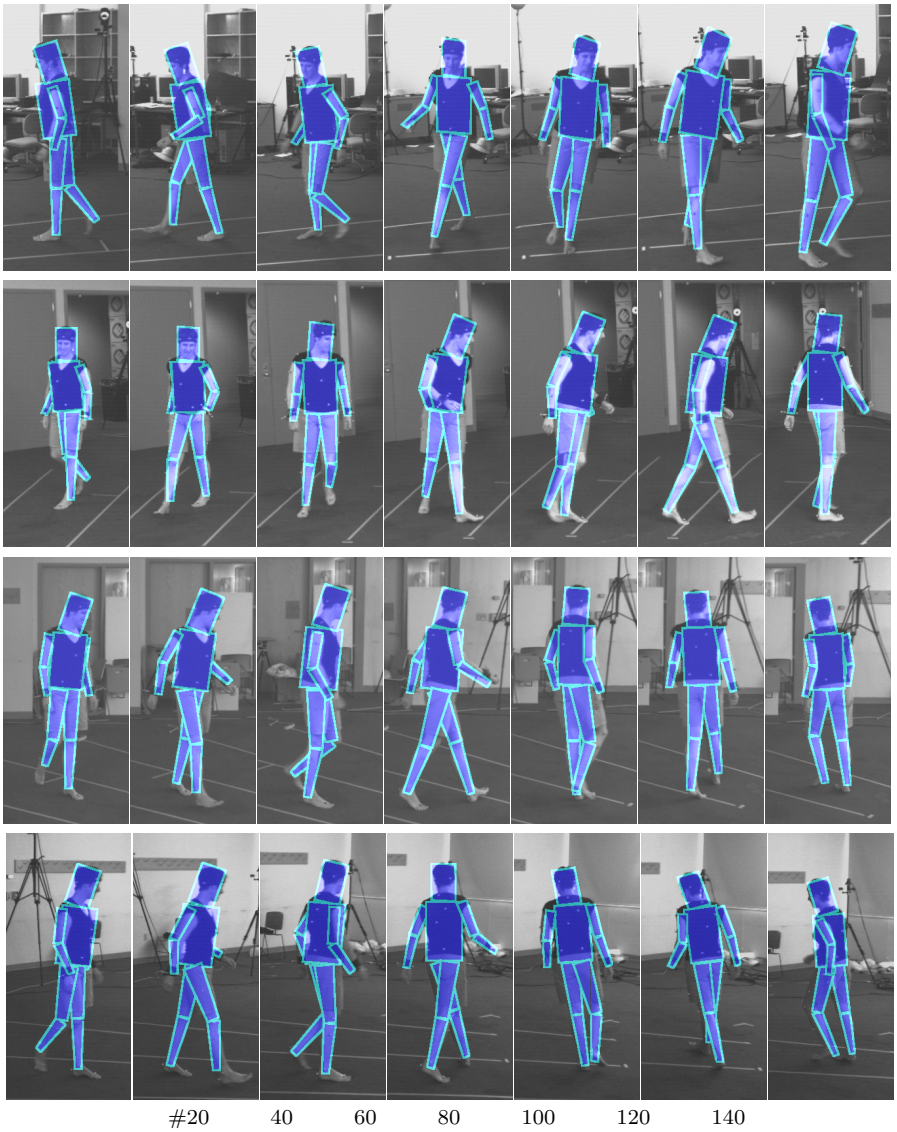

Fig. 9 Articulated 3D human body tracking on LeeWalk sequence.

Figure 10 depicts plots of tracking errors versus frame number for the LeeWalk sequence. The plots have been obtained by an ordinary PSO, executing 10 iterations and consisting of 100 and 300 particles. The average Euclidean error has been calculated on the basis of 15 virtual markers, which were placed on the body: one for pelvis, neck and head, and two for shoulders, elbows, wrists, hips, knees and ankles. As we can observe, the average tracking error for the first configuration of the PSO is about 46.4 mm, whereas for the second one configuration it is about 43.2 mm. Similar tracking accuracy has been reported in [47], where a hierarchical PSO (HPSO) [21] has been used to diminish the search space. Moreover, the discussed tracking accuracy was obtained using 3000 calls of the objective function.

On Fig. 11 there are depicted sample tracking errors that were obtained on the P1S sequence. The tracking errors on this sequence are slightly larger since the walking person occupies far smaller areas in the images acquired by the side cameras. Moreover, the size of the person's silhouette in the frontal/back images changes during the walk, see also tracking results in [24], whereas in the LeeWalk sequence the area occupied by the person in all camera views does not change so much. The errors were calculated using the locations of the markers

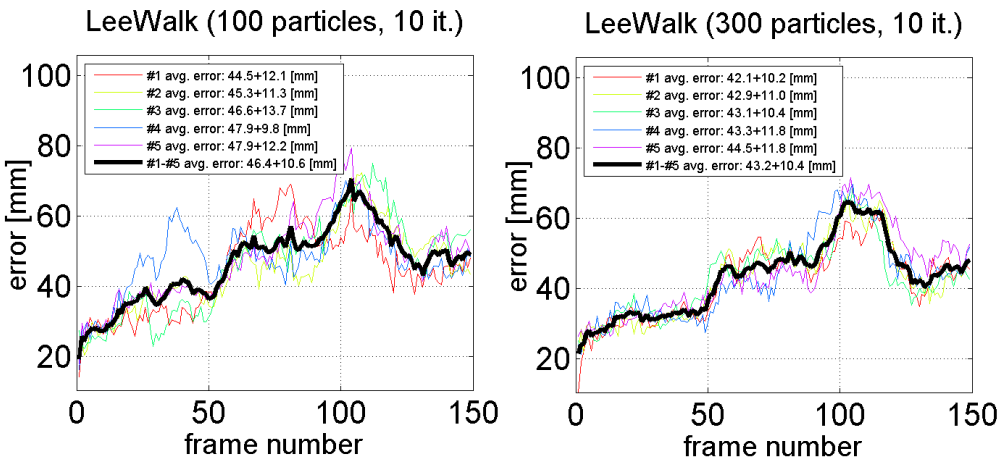

Fig. 10 Tracking errors [mm] versus frame number on LeeWalk sequence. The average error for 100 and 300 particles is 46.4 mm and 43.2 mm, respectively.

recovered by marker-based moCap system and locations of the virtual markers estimated by our marker-less moCap system. For each frame they were computed as average Euclidean distance between individual (39) markers and the recovered 3D joint locations. From the above set of markers, 4 markers were placed on the head, 7 markers were attached on each arm, 12 on the legs, 5 on the torso and 4 markers were attached to the pelvis.

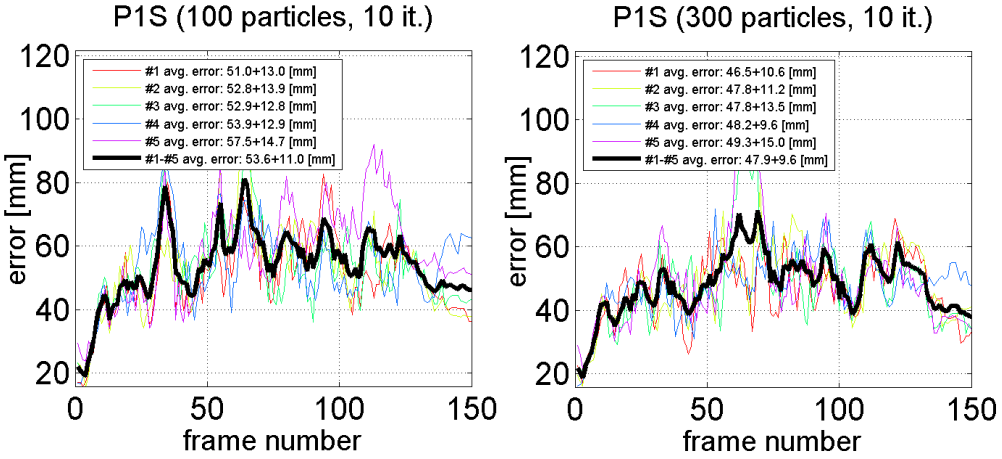

Fig. 11 Tracking errors [mm] versus frame number on sequence P1S. The average error for 100 and 300 particles is 53.6 mm and 47.9 mm, respectively.

The plots shown on Fig. 12 illustrate the tracking errors versus number of the particles. As we can observe, the PSO configuration with ten iterations allows us to obtain better tracking accuracy in comparison to PSO executing only five iterations. As we can notice, with the increased number of iterations to fifteen or even twenty, the decrease in tracking accuracy is not so high as in the case of change of the number of the iterations from five to ten. Thus, a PSO configuration with one hundred particles and ten iterations can be assumed as a base configuration for real-time tracking. For such a configuration the motion on LeeWalk sequence can be reconstructed with frequency of 13 Hz and with frequency of about 16 Hz on P1D and P2S sequences.

Figure 13 contains sample plots of tracking accuracy versus number of the particles, which has been obtained on the P1S sequence. As we can notice, the increase of the number of iterations from 10 to 20 leads to far better accuracy. The reason for this is smaller frequency at which the P1S data were recorded. This means that for the sequences recorded with 25 Hz more iterations are needed to estimate the pose of the person, which changes more between the successive frames.

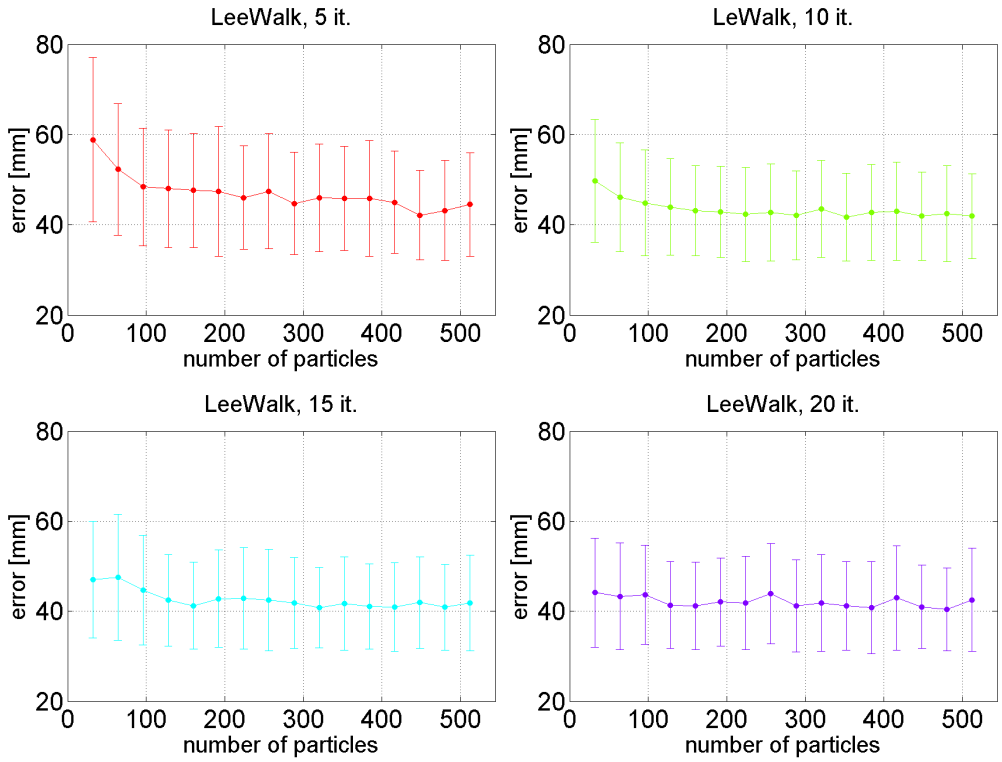

Fig. 12 Tracking errors [mm] versus number of particles on LeeWalk sequence using 100 particles.

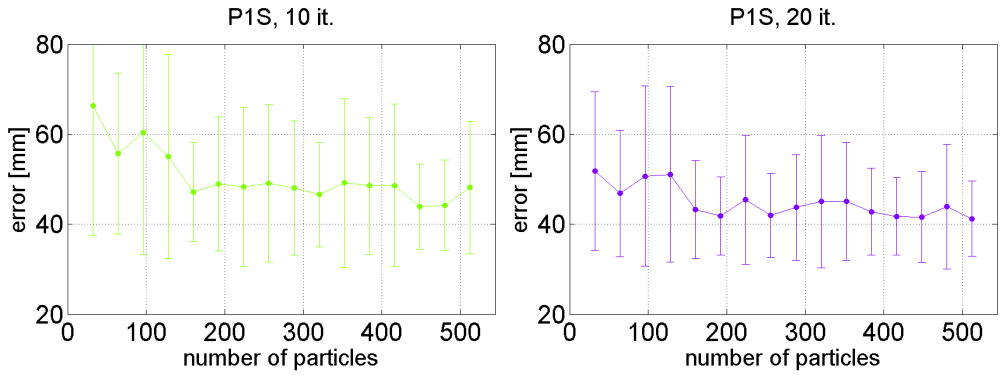

Fig. 13 Tracking errors [mm] versus number of particles on P1S sequence using 100 particles.

In Tab. 3 there are shown some quantitative results illustrating the tracking accuracy, which has been obtained on image sequences from two different datasets, for various number of particles and iterations. The discussed results have been obtained on the basis of ten runs of PSO with unlike initializations as well as distinct random numbers. As we can notice, using ordinary PSO it is possible to obtain on the LeeWalk sequence a tracking error close to 40 mm. We can also notice, that the variance decreases with the number of the calls of the objective function. One of the reason of different tracking accuracy is the frequency at which the considered image sequences were recorded, different image size as well as the aspect ratio of the cameras.

Figure 14 depicts tracking error for LeeWalk sequence, assuming that it was acquired at 12 Hz, i.e. with frequency at which the presented system is able to perform tracking in real-time, see tracking time in Table 1 for PSO consisting of 100 particles and executing 10 iterations. Owing to capability of the system to evaluate the objective function 12000 times in one second, the full-body motion tracking can be done in real-time with average accuracy of about 70 mm.

Although the main purpose of this work was to develop effective mechanisms to marker-less 3D human motion capture in real-time on the basis of two pairs of perpendicular cameras, i.e. camera setup that is used

Table 3 Average errors.

			Seq. P1S	Seq. P2S	Seq. LeeWalk
	#particles	it.	error [mm]	error [mm]	error [mm]
CUDA	10	100	63.4±43.6	64.7±45.9	47.0±12.8
		300	61.0±42.6	55.9±35.6	44.9±12.0
		500	50.4±27.8	52.9±30.7	43.6±11.6
	20	100	58.0±36.1	62.5±46.9	45.7±12.6
		300	55.2±35.4	51.3±28.7	44.5±12.3
		500	47.4±23.9	54.3±31.2	44.9±13.3
CUGL	10	100	60.3±40.6	58.2±36.2	47.9±11.7
		300	48.1±27.6	53.9±27.9	44.1±11.8
		500	44.2±22.6	45.0±30.3	42.5±10.6
	20	100	50.7±32.0	46.7±33.8	43.6±11.1
		300	43.8±23.5	42.6±27.0	41.2±10.2
		500	41.2±20.4	43.6±27.5	41.5±11.5

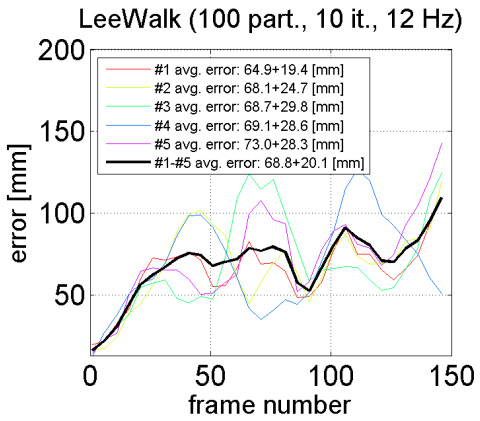

Fig. 14 Tracking errors [mm] versus frame number on images from LeeWalk sequence (100 particles, 10 it.) acquired with 12 Hz.

in Human Motion Laboratory of PJAIT[1], we also evaluated the accuracy of the system on HumanEva I image sequence [40], which is frequently used in research on marker-less 3D human motion capture. The dataset contains seven calibrated video sequences (4 grayscale and 3 color) that are synchronized with 3D ground-truth obtained from Vicon moCap. The grayscale images have size 644 × 448 pixels and color images are of size 659 × 494 pixels. The videos were acquired at 60 Hz. On first 210 frames from walking sequence with S1 subject the error was equal to 55.8±21.3 mm, 53.5±18.4 mm, 52.3±19.1 mm for 96, 304 and 512 particles, respectively. The discussed errors were obtained by PSO executing ten iterations for each frames. Each result is the average of ten runs with unlike initializations of the PSO.

In real-time motion tracking all operations are time critical, and all main computations need to be optimized for time. Video frames arrive at a steady rate and need to be sampled for the tracking at a frequency resulting from the total time needed for acquisition, image processing, transfer time between CPU and GPU, and time for human pose estimation. Using OpenMP, the image processing (silhouette extraction, edge extraction, edge distance map) of images of size 480 × 270 pixels can be done in 12 ms. By overlapping data transfers with computation on the host, and computation on the device [18], we achieved stream and data-parallel computations. On the CPU, every second frame was acquired from the camera, processed and then the results were delivered to the GPU, whereas on the GPU, in parallel, the results from previous frame have been acquired and then used in the 3D pose estimation.

[1] `http://bytom.pja.edu.pl/`

7 Conclusions

In this paper, we presented a framework for 3D model based, markerless full-body motion tracking using CUDA-OpenGL interoperability. The objective function, which is the most computationally demanding part of model based 3D motion reconstruction, has been calculated on a GPU. We proposed a computational framework, which allows us to effectively utilize the rendering power of OpenGL to render the 3D models in the predicted poses, whereas the CUDA threads are used to match such rendered models with the image features and to perform the tracking. We demonstrated experimentally that the OpenGL-based 3D model rendering can shorten the whole motion tracking time considerably. The 3D motion reconstruction has been achieved by a parallel particle swarm optimization. We demonstrated that for larger number of the particles the CUDA-OpenGL interoperability allows us to obtain better efficiency of computations in comparison to CUDA-based implementation of the objective function. In particular, we showed experimentally that thanks to executing computationally-demanding parts of the algorithm on GPU, CUDA-OpenGL speedups up to 15 times over a CPU-only implementation were achieved, in contrast to 4.3 CUDA speedups over the CPU. In a four camera-setup the full-body motion tracking can be realized with frequency close to 15 Hz.

In future work, we intend to reduce the tracking error as well as to extend our system such that 3D motion tracking on the basis of twice larger images will be possible in real-time. By using larger images it will be justified to use a more precise 3D mesh model. Initial results, which were obtained on NVIDIA Titan V GPU demonstrate that it will be possible to achieve 3D tracking in real-time on the basis of a 3D mesh model consisting of thousand(s) vertices.

Acknowledgements This work was supported by Polish National Science Center (NCN) under research grants 2014/15/B/ST6/02808 and 2017/27/B/ST6/01743. The authors would thank Professor Konrad Wojciechowski and his PJAIT team for providing datasets for investigations.

References

1. Akenine-Möller, T., Haines, E., Hoffman, N.: Real-Time Rendering. A. Peters, Ltd., MA, USA (2008)
2. Alexiadis, D., Chatzitofis, A., Zioulis, N., Zoidi, O., Louizis, G., Zarpalas, D., Daras, P.: An integrated platform for live 3D human reconstruction and motion capturing. IEEE Trans. Cir. and Syst. for Video Technol. **27**(4), 798–813 (2017)
3. Alexiadis, D., Daras, P.: Quaternionic signal processing techniques for automatic evaluation of dance performances from mocap data. IEEE Trans. on Multimedia **16**(5), 1391–1406 (2014)
4. Ayuso, F., Botella, G., Garcia, C., Prieto Matias, M., Tirado, F.: GPU-based acceleration of bio-inspired motion estimation model. Concurrency and Comp.: Pract. and Experience **25**, 1037–1056 (2013)
5. Balan, A., Sigal, L., Black, M.: A quantitative evaluation of video-based 3D person tracking. In: Int. Conf. on Comp. Comm. and Networks, pp. 349–356. IEEE Comp. Soc. (2005)
6. Barris, S., Button, C.: A review of vision-based motion analysis in sport. Sports Medicine **38**(12), 1025–1043 (2008)
7. Botella, G., Martìn H., J.A., Santos, M., Meyer-Baese, U.: FPGA-based multimodal embedded sensor system integrating low- and mid-level vision. Sensors **11**(8), 8164–8179 (2011)
8. Brown, J., Capson, D.: A framework for 3D model-based visual tracking using a GPU-accelerated particle filter. IEEE Trans. on Vis. and Comp. Graphics **18**(1), 68–80 (2012)
9. Cano, A., Yeguas-Bolivar, E., Muñoz-Salinas, R., Medina-Carnicer, R., Ventura, S.: Parallelization strategies for markerless human motion capture. J. of Real-Time Image Processing **14**(2), 453–467 (2018)
10. Castano-Diez, D., Moser, D., Schoenegger, A., Pruggnaller, S., Frangakis, A.: Performance evaluation of image processing algorithms on the GPU. J. of Structural Biology **164**(1), 153 – 160 (2008)
11. Clark, R., Pua, Y.H., Fortin, K., Ritchie, C., Webster, K., Denehy, L., Bryant, A.: Validity of the Microsoft Kinect for assessment of postural control. Gait & Posture **36**(3), 372 – 377 (2012)
12. Corazza, S., Mundermann, L., Gambaretto, E., Ferrigno, G., Andriacchi, T.P.: Markerless motion capture through visual hull, articulated ICP and subject specific model generation. Int. J. of Computer Vision **87**(1-2), 156–169 (2010)
13. Deutscher, J., Blake, A., Reid, I.: Articulated body motion capture by annealed particle filtering. In: IEEE Int. Conf. on Pattern Recognition, pp. 126–133 (2000)
14. Fleischmann, P., Austvoll, I., Kwolek, B.: Particle Swarm Optimization with soft search space partitioning for video-based markerless pose tracking. In: Proc. of the 14th Int. Conf. on Advanced Concepts for Intelligent Vision Systems, Lecture Notes in Computer Science, vol. 7517, pp. 479–490. Springer-Verlag, Berlin, Heidelberg (2012)
15. Fung, J., Mann, S.: Using graphics devices in reverse: GPU-based image processing and computer vision. In: IEEE Int. Conf. on Multimedia and Expo, pp. 9–12 (2008)
16. Ganapathi, V., Plagemann, C., Koller, D., Thrun, S.: Real time motion capture using a single time-of-flight camera. In: IEEE Conf. on Comp. Vis. and Patt. Rec. (CVPR), pp. 755–762 (2010)
17. Han, J., Shao, L., Xu, D., Shotton, J.: Enhanced computer vision with Microsoft Kinect sensor: A review. IEEE Trans. Cybernetics **43**(5) (2013)
18. Harris, M.: How to overlap data transfers in cuda c/c++ (2012). URL `https://devblogs.nvidia.com/how-overlap-data-transfers-cuda-cc/`

19. Hauswiesner, S., Straka, M., Reitmayr, G.: Virtual try-on through image-based rendering. IEEE Trans. on Visualization and Computer Graphics **19**(9), 1552–1565 (2013)
20. Ji, X., Liu, H.: Advances in view-invariant human motion analysis: A review. IEEE Trans. on Systems, Man, and Cyb., Part C **40**(1), 13–24 (2010)
21. John, V., Trucco, E., Ivekovic, S.: Markerless human articulated tracking using hierarchical particle swarm optimisation. Image and Vision Computing **28**(11), 1530 – 1547 (2010)
22. Kennedy, J., Eberhart, R.: Particle swarm optimization. In: Proc. of IEEE Int. Conf. on Neural Networks, pp. 1942–1948. IEEE Press, Piscataway, NJ (1995)
23. Krzeszowski, T., Kwolek, B., Wojciechowski, K.: GPU-accelerated tracking of the motion of 3D articulated figure. In: Proc. of Int. Conf. on Computer Vision and Graphics, Lecture Notes in Computer Science, vol. 6374, pp. 155–162. Springer-Verlag (2010)
24. Kwolek, B., Krzeszowski, T., Gagalowicz, A., Wojciechowski, K., Josinski, H.: Real-time multi-view human motion tracking using particle swarm optimization with resampling. In: Proc. of Int. Conf. on Articulated Motion and Deformable Objects, Lecture Notes in Computer Science, vol. 7378, pp. 92–101. Springer (2012)
25. Kwolek, B., Rymut, B.: Marker-less 3D human motion capture in real-time using Particle Swarm Optimization with GPU-accelerated fitness function. In: 9th Int. Conf. on Image and Graphics, Revised Selected Papers, Part III, pp. 423–435 (2017)
26. Li, Z., Ji, Y., Yang, W., Ye, J., Yu, J.: Robust 3D human motion reconstruction via dynamic template construction. In: Int. Conf. on 3D Vision (2017)
27. Luebke, D., Humphreys, G.: How GPUs work. Computer **40**(2), 96–100 (2007)
28. Menache, A.: Understanding Motion Capture for Computer Animation, 2nd edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2010)
29. Moeslund, T.B., Hilton, A., Krüger, V.: A survey of advances in vision-based human motion capture and analysis. Comput. Vis. Image Underst. **104**(2), 90–126 (2006)
30. NVIDIA Corporation: NVIDIA's Next Generation CUDA Compute Architecture: Kepler^{TM} GK110 (2012). URL `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`
31. Poppe, R.: Vision-based human motion analysis: an overview. Computer Vision and Image Understanding **108**(1–2), 4–18 (2007)
32. Pulli, K., Baksheev, A., Kornyakov, K., Eruhimov, V.: Real-time computer vision with OpenCV. Comm. ACM **55**(6), 61–69 (2012)
33. Robertini, N., Casas, D., Rhodin, H., Seidel, H.P., Theobalt, C.: Model-based outdoor performance capture. In: Int. Conf. on 3D Vision (2016)
34. Rymut, B., Kwolek, B.: Real-time multiview human pose tracking using graphics processing unit-accelerated particle swarm optimization. Concurr. Comput. : Pract. Exper. **27**(6), 1551–1563 (2015)
35. Saini, S., Rambli, D.R., Zakaria, M., Sulaiman, S.: A review on particle swarm optimization algorithm and its variants to human motion tracking. Mathematical Problems in Engineering (2014)
36. Schmaltz, C., Rosenhahn, B., Brox, T., Weickert, J.: Region-based pose tracking with occlusions using 3D models. Machine Vision and Appl. **23**(3), 557–577 (2012)
37. Segal, M., Akeley, K.: The OpenGL® Graphics System: A Specification (Version 4.3 (Core Profile) - February 14) (2013). URL `https://www.opengl.org/registry/doc/glspec43.core.20130214.pdf`
38. Seo, B.K., Park, H., Park, J.I., Hinterstoisser, S., Ilic, S.: Optimal local searching for fast and robust textureless 3D object tracking in highly cluttered backgrounds. IEEE Trans. on Visualization and Computer Graphics **20**(1), 99–110 (2014)
39. Shum, H., Ho, E., Jiang, Y., Takagi, S.: Real-time posture reconstruction for Microsoft Kinect. IEEE Trans. on Cybernetics **43**(5), 1357–1369 (2013)
40. Sigal, L., Balan, A.O., Black, M.J.: HumanEva: Synchronized video and motion capture dataset and baseline algorithm for evaluation of articulated human motion. Int. J. of Computer Vision **87**(1-2), 4–27 (2010)
41. Stam, J.: What every CUDA programmer should know about OpenGL. In: GPU Technology Conf. (2009)
42. Tan, Y.: GPU-based Parallel Implementation of Swarm Intelligence Algorithms, 1st edn. Morgan Kaufmann Publ. Inc., San Francisco, CA, USA (2016)
43. Tan, Y., Ding, K.: A survey on GPU-based implementation of swarm intelligence algorithms. IEEE Trans. on Cybernetics **46**(9), 2028–2041 (2016)
44. Vasudevan, R., Kurillo, G., Lobaton, E., Bernardin, T., Kreylos, O., Bajcsy, R., Nahrstedt, K.: High-quality visualization for geographically distributed 3-D teleimmersive applications. IEEE Trans. on Multimedia **13**(3), 573–584 (2011)
45. Wu, C., Aghajan, H.: Real-time human pose estimation: A case study in algorithm design for smart camera networks. Proc. of the IEEE **96**(10), 1715–1732 (2008)
46. Yang, Z., Metallinou, A., Narayanan, S.: Analysis and predictive modeling of body language behavior in dyadic interactions from multimodal interlocutor cues. IEEE Trans. on Multimedia **16**(6), 1766–1778 (2014)
47. Zhang, Z., Seah, H.S., Quah, C.K., Sun, J.: GPU-accelerated real-time tracking of full-body motion with multi-layer search. IEEE Trans. on Multimedia **15**(1), 106–119 (2013)
48. Zhao, P., Zhu, H., Li, H., Shibata, T.: A directional-edge-based real-time object tracking system employing multiple candidate-location generation. IEEE Trans. on Circuits and Systems for Video Technology **23**, 503–517 (2013)
49. Zhou, H., Hu, H.: Human motion tracking for rehabilitation–A survey. Biomedical Signal Proc. and Control **3**(1), 1 – 18 (2008)
50. Zivkovic, Z., van der Heijden, F.: Efficient adaptive density estimation per image pixel for the task of background subtraction. Pattern Rec. Letters **27**(7), 773 – 780 (2006)