# Fast Animation Crowds Using GPU Shaders and Motion Capture Data

Mankyu Sung(✉)

Department of Game and Mobile, College of Engineering, Keimyung University, 1095 Dalgubeol-Daero, Dalseo-Gu, Daegu 42601, Republic of Korea
mksung@kmu.ac.kr

**Abstract.** Animating and rendering of more than 1000 characters in real-time has been a big problem in computer animation research community. In this paper, we introduce the GPU-based technique that are able to animate and render more than thousand of characters at real-time rate where each individual character's movement is animated using motion capture data. The core part of the proposed technique is to split the whole processes into CPU and GPU bound jobs and separate static data with dynamic data of simulation. All static data are sent to GPU memory once and frequently updated data such as transformation matrices are applied on the GPU using SSBO (Shader Storage Buffer Object). High level path planning, on the other hand, are performed in CPU side so that many complicated algorithms can be easily implemented in the application. Experiments shows that the proposed techniques have real-time performance.

**Keywords:** Crowd simulation · GPU shaders

## 1 Introduction

Animation and rendering of a big crowd at real-time rate has been a big huddle in computer animation research community for the last decade. Given rapid growth of performance of graphics card, we must utilize the power of GPU as much as possible to increase animation and rendering speed. However, since most of crowd simulation techniques rely on heavy mathematical computation, it is not easy to design an algorithm optimized for GPU characteristics. For exploiting the GPU as much as possible, we have to think about what need to be computed on CPU and what need to be done on GPU and how to update the data efficiently. In generally, CPU is composed of just a few cores with lots of cache memory that can process a few threads at a time whereas a GPU is made of hundreds of cores that can handle thousands of threads at the same time. The ability of a GPU with more than 100 cores to process thousands of threads can accelerate some software by 100x over a CPU alone. Therefore, when we design an algorithm, we need to consider this characteristics of GPU and try to run the algorithm as parallel as possible. Ever since current GPU has allowed users to program on the GPU specifically, there has been several programming frameworks to support. The OpenGL or DirectX 3D API (Application Programming Interface) provides the *Shaders* that replace some stages of 3D graphics pipeline. The Shaders are small programs running in the graphics pipeline and tells the computer how to render each pixel.


J. J. Park et al. (eds.), *Advances in Computer Science and Ubiquitous Computing*, Lecture Notes in Electrical Engineering 474,
https://doi.org/10.1007/978-981-10-7605-3_193

Second, for using GPU for general computation, the general programming model such as Nividia's CUDA or OpenCL has been developed.

In this paper, we propose shader-based techniques for animating and rendering more than 1000 characters simultaneously at real-time rate where each individual motions are highly plausible. That is, each character must be represented as a full humanoid skeleton that consisted of more than 25 joints and movement of characters must be performed by applying motion capture data. Our algorithm splits the entire simulation into CPU and GPU bound jobs and perform a series of processes optimized for CPU and GPU.

## 2 Related Works

Crowd simulation has been drawing a lot of attentions from computer animation researchers for the last decades. Most of algorithms are targeting on generating the realistic crowd movement without collision or synthesizing the group motion interactively. [1–3] Not many works has been done on how to use GPU for rendering and simulating crowds. Benjamin et al. proposed a method that used a set of GPU clusters for visualizing crowds in which around 4000 characters are simulated at 60 framerate [7]. Jeremy et al. presented GPU-friendly path planning framework for large scale crowd simulation [5]. In this case, by representing the each individual as a simple cylinder, they are able to simulate 65,000 agents at real-time framerate. For simulation, they applied the Continuum Crowds technique proposed by Treuille et al. [8]. Kim et al. used the GPU for LOD assignment and view frustum culling for fast rendering of crowds [6]. All techniques mentioned above are able to generate the crowds at real-time rate, but do not guaranteed the quality of individual motions since they simply ignore individual motion. In our approach, on the other hand, we applies the motion capture to all characters and focus on how to manage those data in on GPU efficiently. As a result, we are able to simulate and render around 1000 characters in real-time whose individual is realistically animated with motion capture data.

## 3 Data Management on GPU

### 3.1 CPU Bound Job

We are going to explain all steps that are done in the CPU side first. The CPU bound jobs take care of parsing the motion capture data and produce a set of transformation matrices to represent the character joints from a simple basic 3D model. Also, it keep updating the positions of characters using path planning algorithm. All required data must be arranged as a GPU-friendly format such as textures so that they can be read and updated on GPU in a parallel manner. For animating individual character, we apply a cyclic locomotion motion capture data formatted as BVH (BioVision Hierarchy).

The motion is a walking motions. A particular $i$ frame of a motion consisting of $m$ frames and $k$ joints can be represented as a vector as follows:

$$F^i = \left\{p_0^i, q_0^i, q_1^i \ldots, q_k^i\right\}, p_i \in R^3, q_i \in S^3, 0 \leq i \leq m$$

where $p_0$ is the root joint position at frame $i$, which corresponds to the global position of the whole character, $q_0$ is its global orientation, and $q_j$ is the local orientation of joint $j$. In most cases, the skeleton of motion is constructed hierarchically and the root joint is the pelvis joint located at the center of the body. The left of Fig. 1 shows an example of hierarchy of BVH.

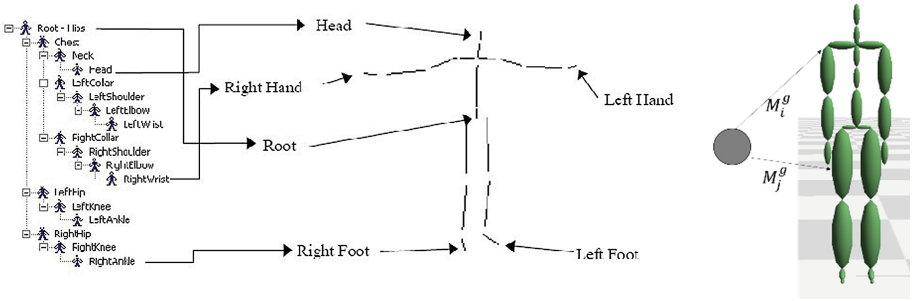


**Fig. 1. Left**: Joint Hierarchy of BVH motion (http://www.mindfiresolutions.com/BVH-biovision-hierarchy.htm) **Right**: Joint representation of joint from a sphere using transformation matrices

The local transformation of a joint represent its orientation in its local coordinate system. Then, it is in turn subject to its parent joint's local orientations. To calculate a global $4 \times 4$ matrix transform for a joint, the local transform needs to be pre-multiplied by its parent's global transform, which itself is derived my multiplying its local transform with its parent's global transform and so on.

If $k$ is the current joint number where the parent joint is $k - 1$, then global transformation of joint $k$, $M_k^i$, at frame $i$ can be calculated as following:

$$M_k^i = \prod_{j=0}^{k} M_j^i$$

where $M_j^i$ is the local transformation matrix of joint k and the local transformation matrix $M_k^l$ can be calculated from $q_k^l$.

The global transformation of joint $i$ is used for transforming a basic 3D object for representing the joint. In our approach, we use a simple sphere to represent the joint. The sphere are rotated, translated and then scaled based on the global transformation matrix. Right of Fig. 1 shows the representation of joints.

For entire frames of motion data, our algorithm calculates $4 \times 4$ transformation matrices for all joints. Those data then passed to the GPU using SSBO (Shader Storage Buffer Object). The SSBO is newly added feature of 3D graphics API for storing and updating a large chunk of data on GPU. The specification guarantees that SSBOs can be up to 128 MB. But, most implementations will let you allocate a size up to the limit of GPU memory and can read and write using incoherent memory accesses.

The second most important job of CPU bounds is to update the global position of characters. Note that transformation matrices of $M_k^i$ are used the determine the particular pose of character. They do not set the global position and orientation of characters. In order to put the character into a particular position with a particular orientation, we use another matrix that can be updated by path planning algorithm. Since original motion data has $p_0^i$ at each frame to represent the global position of root joint, we use the offset distance of two adjacent frames $p_0^i$ and $p_0^{i-1}$. Suppose the initial positon and orientation matrix are $C_i \in R^3$ and $M_o$ is $4 \times 4$ orientation matrix respectively. The initial $C_i$ is set randomly on the environment and $M_o$ is same for entire characters for simulating the group movement. The update formula is following:

$$C_{i+1} = C_i + M_o(p_0^i - p_0^{i-1})$$

Once we determine the $C_{i+1}$, then it is also converted $4 \times 4$ translation matrix. All translation matrices for entire characters are then packed into a single array and send to GPU through SSBO.

In addition, given motion data composed of several frames, it is better to give different starting frame to the characters so that they do not show exactly same poses when we animate them. The current frame number also must be updated through the similar way of updating the character position. Those data is also packed into a single array and then send to the GPU as well.

Finally, the efficient drawing of crowds are done using instancing technique. Since each character is consisted of $k$ number of joints, the number of total instance of $k * n$ where $n$ is the number of characters.

### 3.2 GPU Bound Job

On the GPU side, beside the vertex positions of sphere and their normal vectors, CPU send three big chunk of data through SSBO. Those data are joint transformation matrices, character position matrices and current fame number data. All of them are passed as 1D array. Because the joint transformation matrices packed to cover the entire frames and character position matrices and current fame data is also packed for entire characters. We need a convenient way of obtaining a particular joint transformation matrix index from the joint transformation matrix SSBO and character index from instancing. Current 3D graphics API such as OpenGL provides the in-built

instancing number variable. Given the number of joint, $k$, and *instanceID* which is the instancing number, the character number $m$ can be calculated easily.

$$m = int(\left\lfloor \frac{instanceID}{k} \right\rfloor)$$

The particular joint index $j$ from joint transformation SSB also can be calculated as following:

$$j = k * f + (instancID \,\%\, k)$$

where '%' denotes the modular operator and $f$ is the current frame number obtained from the current frame number SSBO and $k$ is the total number of frames.

By using $m$ and $j$ *as* index of SSBO, we are able to get a complete model matrix for positioning a particular joint in the global coordinate system. Then, by multiplying viewing matrix and projection matrix, we can show the joint on the screen. The joint color is also calculated by a simple Phong illumination technique.

## 4 Experiments

The algorithms were verified by building a crowd simulation system. The testing system was built on Windows 10 operating system. The hardware specifications were Intel Xeon E5-1607 with 8 G memory and the graphics card was Nvidia GTX 1070. Figure 2 shows the screen shots of crowd simulation system. In this case, we put the 2000 characters. The simulation and rendering speed was around 24 frame/sec.

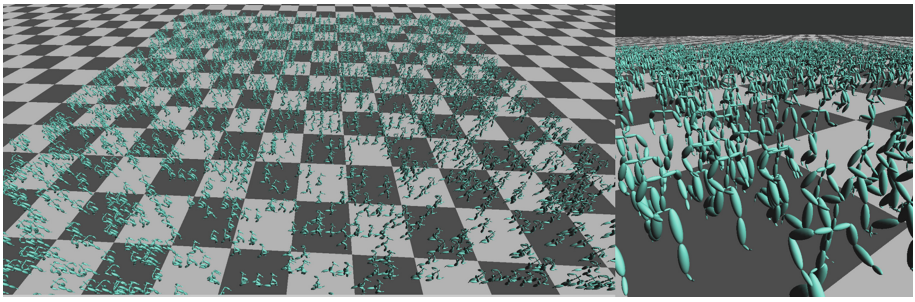

**Fig. 2.** Screenshot of crowd of 2000 characters.

Figure 3 shows a performance graph as we increase the number of characters. Experiments verified that our algorithm can simulate up to 2000 characters in real time rate.

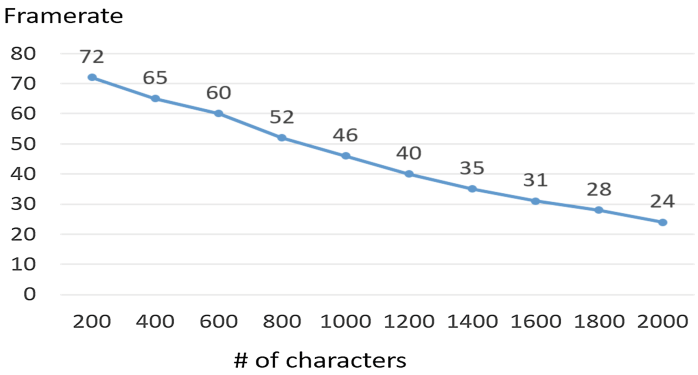


**Fig. 3.** Performance graph

## 5 Conclusion

In this paper, we introduce the GPU-based crowds simulation technique. In this technique, we split the whole rendering and simulation into CPU-bound jobs and GPU-bound jobs. The CPU bound jobs are responsible for maintaining the update of position of whole character and the preprocessing of transformation matrix to represent the joints based on motion capture data. Also, it must keep tracking the frame number that each character animating on. GPU bound jobs apply the transformation matrices to the basic 3D model to represent the character and then calculating the lighting. In our approach, we exploit the instancing technique that can render massive number of same 3D model but different location efficiently. For the future works, we would like to extend our work to represent the individual as a full 3D skinned characters. Also, we want to synthesize continuously changing motions on GPU from the discrete set of motion data.

**Acknowledgment.** This work was supported by a grant from the National Research Foundation of Korea (NRF) funded by the Korean government (No. 2015R1D1A1A010590).

## References

1. Wolinksi, D., Lin, M.C.: WarpDriver: context-aware probabilistic motion prediction for crowd simulation. ACM. Trans. Graph. **35**(6) (2016)
2. Narang, S., Best, A., Manocha, D.: Interactive simulation of local interactions in dense crowds using elliptical agents. J. Stat. Mech. Theory Exp. **3** (2017)
3. Best, A., Narang, S., Curtis, S., Manocha, D.: DenseSense: interactive crowd simulation using density-dependent filters. In: Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation, pp. 97–102. Eurographics Association (2014)
4. Barnettm, A., Shum, H., Komura, T.: Coordinated crowd simulation with topological scene analysis. Comput. Graph. Forum **35**(6), 120–132 (2016)
5. Shopf, J., Oat, C., Barczak J.: GPU crowd simulation. ACM Siggraph-Asia, Technical Briefs (2008)

6. Park, H., Han, J.: Fast rendering of large crowds using GPU. In: International Conference on Entertainment Computing, pp. 197–202 (2008)
7. Hernandez, B., Perez, H., Rudomin, I., Ruiz, S., Toledo, L.: Simulating and visualizing real-time crowds on GPU clusters. Computación y Sistemas **18**(4), 651–664 (2014)
8. Cooper, A., Cooper, S., Popovic, Z.: Continuum crowds. ACM. Trans. Graph. **25**(3), 1160–1168 (2006)