Fast Rendering of Large Crowds using GPU

HunKi Park, JungHyun Han

Game Research Center, College of Information and Communication, Korea Univeristy
{hunki,jhan}@korea.ac.kr
http://media.korea.ac.kr

Abstract. This paper proposes a fast rendering algorithm for real-time animation of large crowds, which is essential for video games with a large number of non-player characters. The proposed approach leaves the minimal work of rendering to CPU, and makes GPU take all the major work, including LOD assignment and view frustum culling, which have been the typical tasks of CPU. By offloading the rendering overhead from CPU, the approach enables the CPU to perform intensive computations for crowd simulation. The experiments show that tens of thousands of characters can be skin-animated in real time.

Key words: crowd rendering, skinning, instancing, GPU

1 Introduction

In interactive environments, animation of large crowds is an area of research that has been receiving an increased amount of interest. Especially, it is becoming essential in video games that accommodate a number of non-player characters (NPCs). A good example of NPCs is a herd of land animals.

This paper proposes a rendering algorithm which is implemented mostly within GPU. The performance of the algorithm is excellent enough to render tens of thousands of game characters at real time. The experiment results show that the proposed algorithm is appropriate for applications which render extremely large crowds, and require CPU-intensive computation for the crowds simulation.

The organization of the paper is as follows. Section 2 reviews related work. Section 3 describes the rendering algorithm, which is composed of three phases. Each phase is discussed in a subsection in detail. Section 4 presents the implementation and test results. Finally, Section 5 concludes the paper.

2 Related Work

Many works in the area of crowd rendering have adopted the image-based technique based on impostors [1, 2]. Dobbyn et al. [3] proposed a hybrid technique for large crowd rendering, where polygon meshes are used for characters in close proximity and pre-generated impostors are used for distant characters. However, impostors do not support free and fast navigation of the viewpoint.

In OpenGL and DirectX, rendering an object requires invoking a draw-call. Suppose an object is repeatedly rendered with a distinct world matrix. Then, repeated draw-calls are needed. In order to reduce the draw-call overhead, various instancing techniques have been developed. The early work in instancing handles only the rigid objects [4, 5].

In games, the most popular technique for character animation is skinning [6]. The skinning algorithm is based on a hierarchy of bones, and each vertex in the mesh is assigned a set of influencing bones and a blending weight for each influence. A vertex shader implementation of skinning has been reported in [7], where the bone matrices are recorded in the constant registers. Due to the limited number of constant registers, however, the vertex shader-based skinning is not good for rendering large crowds. Wu presented a pixel shader implementation of skinning [8]. As the animation data are stored in a texture, the algorithm shows good performances in rendering large crowds.

The most recent algorithm for skinning animation of large crowds is based on DirectX10, and makes use of the API, DrawIndexedInstanced [9]. The per-instance parameters are encoded into an array of shader constants, and the array is indexed using the system variable, SV_InstanceID. This is often called a constant buffer technique.

3 Crowd Rendering

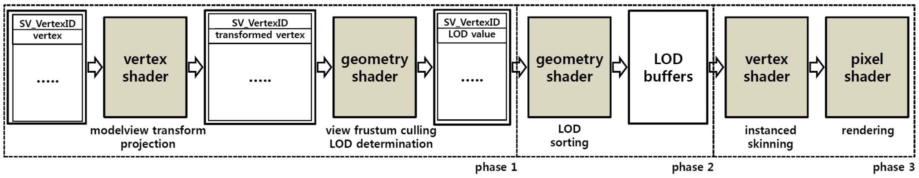


Fig. 1. Flow chart of the rendering algorithm

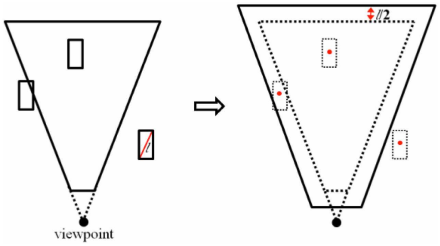


Fig. 2. View frustum culling

The rendering algorithm proposed in this paper leaves the minimal work of rendering to CPU, and makes GPU take all of the remaining work, including level of detail (LOD) assignment and view frustum culling, which have been the typical tasks of CPU. By doing so, CPU can be devoted to expensive computations for crowd simulation. The rendering algorithm consists of 3 phases, as illustrated in Fig. 1.

3.1 1^{st} phase

For a character, an AABB (Axis-aligned bounding box) is pre-computed such that the AABB is tight but large enough to bound all animations of the character. In the current implementation, we have two characters, zebra and gnu, and therefore use two AABBs.

Let us denote the diagonal length of the larger AABB by l. See Fig. 2. The six sides of the view frustum are expanded by $l/2$. Then, each character is reduced into a vertex, the position of which is the center of the AABB. The vertices fill the vertex buffer, and the primitive type is defined to be point list. For each vertex, its ID is automatically generated as a system variable, SV_VertexID. The vertex shader takes a pair of a vertex and its ID, one at a time, as shown in Fig. 1.

The vertex shader transforms each vertex into the clip space, and the view frustum into a $2 \times 2 \times 1$ cuboid. The transformed vertex is culled against the view frustum by the geometry shader, i.e. is tested if it is inside the $2 \times 2 \times 1$ cuboid. If so, the LOD of the vertex is determined using its depth-value.

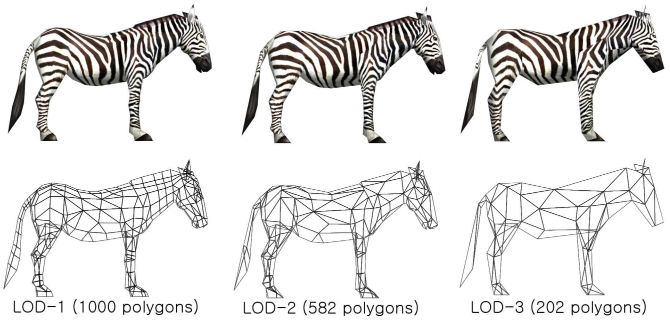
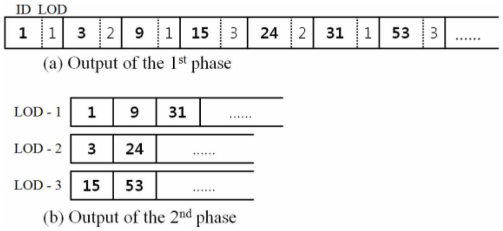


Fig. 3. LOD characters

The current implementation adopts a discrete LOD technique. Fig. 3 shows 3 LODs for zebra. Gnu also has 3 LODs. In fact, zebra and gnu of the same LOD have similar resolutions.

The geometry shaders output is taken as input to the 2^{nd} phase. (No pixel shader is invoked in the 1^{st} phase.) For this purpose, the new feature of DirectX10, stream out, is used. The vertex ID and its LOD value are output, one



(a) Output of the 1st phase

(b) Output of the 2nd phase

Fig. 4. Output streams

at a time, by appending them to an output stream object. Fig. 4-(a) illustrates the output of the 1^{st} phase.

3.2 2^{nd} phase

A number of instances of zebra or gnu will be rendered at the 3^{rd} phase, using an instancing API of DirectX10. Note that instancing requires a single LOD, i.e. a single-resolution mesh. However, the output of the 1^{st} phase is a mixture of different LODs. The 2^{nd} phase sorts the output of the 1^{st} phase into appropriate LOD buffers such that all of the characters in an LOD buffer have the same resolution mesh. As 3 discrete LODs are used in the current implementation, 3 LOD buffers are created.

An LOD buffer is populated by invoking a draw-call, and we need 3 draw-calls in total. For each draw-call, the vertex shader is passed through, and the geometry shader collects the vertices (reduced characters) that share the common LOD value. Fig. 4-(b) illustrates the output of the 2^{nd} phase.

3.3 3^{rd} phase

The last phase of the proposed algorithm performs skinning animation for each instance. For this purpose, the constant buffer technique [9] has been adopted. Both of the geometry data and the animation texture remain fixed for the entire animation. In contrast, the instance buffer implemented in the constant buffer is repeatedly updated. After the instance buffer is updated, the 3-phase rendering starts. Let us return to Fig. 4-(b), the output of the 2^{nd} phase. For each LOD buffer, DrawIndexedInstanced is invoked, i.e. we need 3 draw-calls. Each element in the LOD buffer works as an index into the instance buffer. The appropriate LOD mesh is world-transformed, and then skin-animated using the animation frame extracted from the animation texture.

4 Implementation and Test Result

The proposed algorithm has been implemented in C++, DirectX10 and HLSL on a PC with Intel Core2 Duo E6400, 2GB memory, and ATI Radeon HD 2900

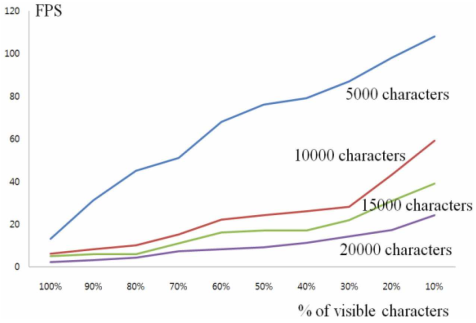


Fig. 5. Performance visualization

XT 512MB. For experiments, two characters, zebra and gnu, are used, each with 44 bones. For both, the finest resolution mesh consists of 1,000 polygons.

Fig. 5 shows the frame rates of the proposed rendering algorithm, where the percentage indicates how many characters are inside the view frustum. For example, a scene of 5,000 characters is rendered at 31 fps when 4,500 characters (90% of 5,000 characters) are inside the view frustum. For analysis, the characters are uniformly spread within the view frustum such that about 1/3 of the visible characters is sorted to an LOD buffer, i.e. each of the 3 LOD buffers contains about 1,500 characters. Obviously, the frame rates get higher when more characters are discarded through view frustum culling.

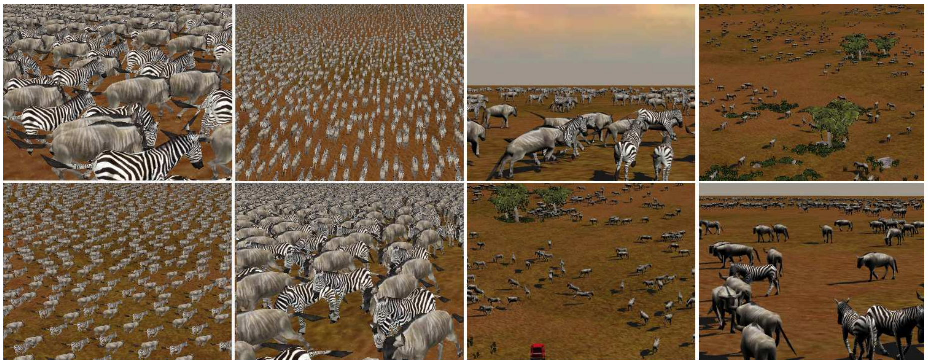


Fig. 6. Rendering results

Fig. 6 shows snapshots of rendering 20,000 characters. The average percentage of the characters inside the view frustum is 27%, and the average FPS is 15. For crowd simulation, the flocking algorithm of Reynolds [10] is used, but any simulation technique can be integrated with the rendering module.

5 Conclusion

This paper presented a rendering algorithm for real-time animation of large crowds. The rendered scene shows tens of thousands of game characters, each animating in different poses, rendered using just a few draw-calls. In the current implementation, 7 draw-calls are needed in total. The rendering algorithm is implemented mostly within GPU. By offloading the rendering overhead from CPU, it enables the CPU to perform intensive computations for the crowds simulation.

Acknowledgments. This research was supported by MKE, Korea under ITRC IITA-2008-(C1090-0801-0046).

References

1. Aubel, A., Boulic, R., Thalmann, D.: Real-time display of virtual humans: levels of details and impostors. IEEE Transactions on Circuits and Systems for Video Technology 10(2) 207-217. 2000.
2. Tecchia, F., Loscos, C., Chrysanthou, Y.: Image based crowd rendering. IEEE Computer Graphics and Applications 22(2) 36-43. 2002.
3. Dobbyn, S., Hamill, J., OConor, K., OSullivan, C.: Geopostors: A real-time geometry/impostor crowd rendering system. ACM Transactions on Graphics 24(3) 933. 2005.
4. Instancing Sample. DirectX SDK. February 2006.
5. Zelsnack, J.: GLSL Pseudo-Instancing. NVIDIA Technical Report. November 2004.
6. Lewis, J.P., Cordner, M., Fong, N.: Pose space deformations: A unified approach to shape interpoalation and skeleton-driven deformation. Proc. SIGGRAPH 165-172. 2000.
7. Gosselin, D. R., Sander, P. V., Mitchell, J. L.: Drawing a Crowd. ShaderX3 (CHARLES RIVER MEDIA) 505-517. 2005.
8. Wu, O.: Animation with R2VB. ATI SDK. 2006.
9. Dudash, B.: Skinned Instancing. NVIDIA Technical Report. February 2007.
10. Reynolds, C.W.: Flocks, herds and schools: A distributed behavioral model. Proc. SIGGRAPH 25-34. 1987.