A Real-time System of Crowd Rendering: Parallel LOD and Texture-Preserving Approach on GPU

Chao Peng, Seung In Park, Yong Cao

Computer Science Department, Virginia Tech, USA
{chaopeng,spark80,yongcao}@vt.edu

Abstract. In modern games, rendering a massive scene with a large number of animated character is imminent and a very challenging task. In this paper, we present a real-time crowd rendering system on GPUs with a special focus on how to preserve texture appearance in progressive LOD-based mesh simplification algorithms. Our results show that the proposed parallel LOD approach can get up to 5.33 times of speedup compared with the standard pseudo-instancing approach.

Keywords: Level of detail, Crowd animation, Texture-preserving, GPGPU

1 Introduction

Rendering large crowds of animated characters has become a common requirement in massively multi-player online games and interactive virtual environments. In many video games, human characters are the major 3D content composed of deformable meshes, where each mesh is represented with a set of triangles. In a typical massive crowd scene, one of the main challenge is how to increase rendering performance of a large number of articulated characters so that the users can have a real-time gaming experience. To do this, many Level of Detail (LOD) approaches, such as *Progressive Mesh* [7] and *Quadric Error Metrics(QEM)* [5], have been used to reduce the complexity of meshes. A LOD is a geometrically simplified representation of an original mesh without losing visual fidelity at a certain distance. In most of games, the rendering attributes, especially surface texture, are as important as geometry shapes for the realistic rendering of animated characters. And it is essential to preserve the correctness of texture appearance on different LODs.

During the past years, Graphics Processing Units (GPUs) have been significantly improved to perform general-purpose computation. By utilizing their highly parallel architecture, the algorithms, traditionally implemented on CPUs, can be re-designed and implemented on GPUs to enhance the performance. Researchers have proposed parallel LOD algorithms on GPU to support real-time rendering [8, 13]. In this paper, we extend our previous work [13], and present a GPU-based system to render crowds of animated characters while preserving

their per-vertex attributes (e.g. texture coordinates). Our contribution emphasizes on a set of criteria that provide texture-preserving constraints for edge-collapsing in data preprocess. We also propose a processing pipeline that combines the NVIDIA's parallel computing architecture and OpenGL shaders to efficiently render the LOD meshes which are generated in runtime.

We organize the rest of the paper as follows. In Section 2, we review some previous works in LOD techniques and GPGPU computing for rendering. In Section 3, we provide a brief overview of our rendering system. In Section 4, we describe our approach of texture-preserving criteria applied in the preprocess. In Section 5, we present the pipeline for rendering animated characters. We describe our experiments and results in Section 6. Finally, we conclude our work and discuss the future works in Section 7.

2 Related works

Mesh simplification has been well studied in the past. In this section, we review the some previous approaches. We also review the work on general propose GPU computing in this area.

2.1 Mesh Simplification and LOD

LOD is a common representation for simplified meshes. It aims to reduce the complexity of 3D meshes based on some criteria, such as the distance to a camera. There are two typical types of LODs: *Discrete LOD* and *Continuous LOD*.

The concept of discrete LOD is to offline create a limited number of meshes to represent the original object. During runtime, the renderer choose a proper LOD from the already generated meshes to be the alternative for rendering. The major limitation of discrete LOD is that it can not provide smooth transitions between two LODs, and cause the "popping" artifacts. Markus Giegl and Michael Wimmer [6] presented a blending-based algorithm to avoid artifacts in image space, which is further improved by [15]. However, it requires high similarities between LOD meshes in order to achieve a smooth LOD change. In addition, the noisy artifacts may occur on the silhouettes of the LODs.

Continuous LOD is supported by a data structure encoding a continuous spectrum of mesh details. A well-known algorithm of continuous LOD is progressive meshes [7], where a original mesh is simplified by collapsing edges iteratively. Then, the mesh is represented as a base mesh with a sequence of vertex splits. At runtime, a LOD mesh can be recovered by applying a prefix of splits on the base mesh. Other simplification approaches for continuous LOD include region-merging measurement [14], quadric error metrics [5], appearance-preserving method [1] and image-driven simplification [9].

2.2 GPU Computing for Mesh Simplification

With the GPU's parallel architecture, we can design and implement efficient parallel algorithm and achieve high performance. GPU-based mesh simplifica-

tion has been studied in the past. In [2], the authors designed a GPU-friendly octree structure for cluster-based LOD generation. Hu et al. [8] proposed a parallel algorithm for view-dependent LOD rendering. Feng et al. [3] presented a parallel LOD approach for skinned meshes using geometry images. Peng et al. [13] presented a method of parallel progressive LOD for rendering massive and complex models interactively. Although these parallel approaches increase the performance of the LOD-based rendering, none of them are well studied on how the textures should be preserved while appearing on different LODs. Since texture is an important rendering feature for gaming-related applications, in this paper, we contribute a set of texture-preserving criteria for the simplification of deformable mesh.

3 Overview

Our system leverages the advantages of a previous work [13], which provides an efficient, parallel LOD algorithm, and interactively renders massive and static models on GPUs. Our system includes two major stages: *Data Preprocess* and *Rendering Pipeline*. We illustrate the system overview in Fig. 1.

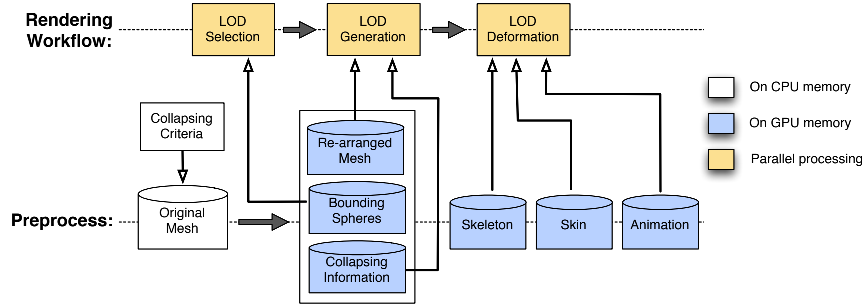


Fig. 1. The overview of preprocess and rendering pipeline.

3.1 Preprocess

In preprocess stage, the vertices and triangles of the original mesh is simplified iteratively. At each iteration, one edge is chosen to be collapsed according to a collapsing criterion. We record the collapsing information into an array structure, where each element corresponds to a vertex, and the value of the element indicates the target vertex that it merges to. Then, we re-arrange the vertices and triangles of the mesh based on the order of edge-collapsing (referring to the details in [13]). we also create a Bounding Sphere(BS) for each original mesh. Note that the BS is large enough to bound the deformed mesh for any animated pose. A BS serves two purposes at runtime. First, we determine the visibility of a character by testing the BS against the view frustum. Second, we generate a desired LOD based on the size of the projected area of the BS.

To avoid the problematic texture mapping onto the surface of each LOD, we present a set of texture-preserving criteria as an auxiliary control on how edges are collapsed at each iteration of simplification process. We describe our texture-preserving criteria in Section 4.

3.2 Run-Time Rendering Pipeline

At runtime, we employs the GPU parallel architecture to increase the rendering performance. Our rendering pipeline includes the following three steps to render an image frame:

1. LOD Selection. We compute the desired complexity for each character. To do this, we determine the visibilities of the characters by testing the BSes against the view frustum. If a character is visible, the LOD level is determined with the appropriate vertex and triangle counts.
2. LOD Generation. Based on the desire complexity of a character, we select a set of triangles from the original data and reform those triangles by performing the precomputed edge collapsing operations.
3. LOD Deformation. We deform the generated LODs to calculate the final positions of vertices with the skeleton, skin and motion data.

4 Mesh Simplification By Perserving Texture Appearance

We first split the vertices associating with multiple sets of texture coordinates. Second, we describe how to categorize the vertices and apply different collapsing rules to them.

An articulated character model is represented as a triangulated mesh associated with a set of texture coordinates. Thus, mesh M can be denoted as a triple, (V, U, T), where $V = \{v_i, v_2, \dots, v_m\}$ is m vertices; $U = \{u_1, u_2, \dots, u_r\}$ is r texture coordinates; $T = \{t_1, t_2, \dots, t_n\}$ is n triangles. t_i is a triple of index pairs, denoted as $t_i = \{(vdx_1, udx_1), (vdx_2, udx_2), (vdx_3, udx_3)\}$, where $vdx_j (j \in [1, 3])$ is a vertex index in the range of $[1, m]$, and $udx_j (j \in [1, 3])$ is a texture coordinate index in the range of $[1, r]$.

4.1 Splitting Vertices According to Texture Coordinates

In OpenGL-based graphics pipeline, *Vertex Buffer Objects* (VBOs) are a common mechanism for storing vertex data on GPU memory. In a mesh, since multiple sets of texture coordinates may associate to a vertex, they will be indexed differently from other vertex attributes. But the OpenGL only supports one index stream used by all vertex attributes. As such, we need to duplicate the vertices which have more than one texture coordinate set so that the index pairs of a triangle, (vdx_j, udx_j), can be replaced with a single universal index, and we call this process *Vertex Splitting*. As a result, each vertex of the mesh will have only one set of texture coordinates, and the vertex count will be equal to the texture

coordinate count. Then, a triangle t_i can be redefined as $\{idx_j\}(j \in 3)$, where idx_j is the universal index for all vertex attributes.

Druing vertex splitting, we also record the splitting information into an array, called *Adjacent Vertices*, denoted as $adjV$. The $adjV[i]$, associated to the vertex v_i, is a set of vertices that are split from the same original vertex v_i, including the original vertex v_i. For example, in Figure 2 (a), vertex v_1 is split into v_9 and v_{10}, therefore, $adjV[1] = \{v_1, v_9, v_{10}\}$. Note that a vertex split from a boundary vertex is still treated as a boundary vertex.

4.2 Texture-Preserving Criteria

QEM [5] is a general and efficient criterion for mesh simplification. At each iteration, each edge is weighted by its cost. Then the edge with the lowest cost is chosen to be collapsed, and the triangles associated with this edge are removed. QEM considers the lengths of edges and face normals when computing the costs of edges. Thus, by using the criteria, we can avoid the mesh inversion problem and preserve the mesh boundaries so that the fidelities of the mesh approximations can be maintained. However, QEM is not sufficient to preserve the texture appearance on the surface, especially after the vertex splitting.

In order to perform a texture-preserving process of mesh simplification, we classify the vertices into three categories according to the splitting information recored in adjacent arrays. We define the categories and the rules of classification as follows (also see Fig. 2(a-b)):

Fixed Vertices. If a vertex, v_i, is a boundary vertex, or the number of vertices in adjV[i] is larger than 2, v_i is classified as a fixed vertex. The fixed vertices are non-collapsible and not involved in the iterations for collapsing edges. They constitute the simplest version of original mesh.

Companioned Vertices. If a vertex, v_i, is not a boundary vertex, and the number of vertices in adjV[i] is equal to 2, v_i is classified as a companioned vertex. if the vertex in adjV[i] is denoted as $\overline{v}_i$, $\overline{v}_i$ will be the companion of v_i during the edge-collapsing.

Regular Vertices. If a vertex, v_i, is not a boundary vertex, and adjV[i] is equal to 1 (did not split), v_i is classified as a regular vertex.

At each iteration, an edge, (v_a, v_b), weighted with the lowest cost is collapsed by merging v_a to v_b. We define that an edge is valid for collapsing if either of the following conditions is satisfied:

1. v_a is a regular vertex.
2. v_a is a companioned vertex; and $(\overline{v}_a, v_c)$ is an actual edge in both the world space and the texture space, where $v_c \in adjV[b]$.

When collapsing the edge (v_a, v_b), if v_a is a companioned vertex, we also collapse the edges $(\overline{v}_a, v_c)$ by merging $\overline{v}_a$ to v_c. This is because (v_a, v_b) and $(\overline{v}_a, v_c)$ are identical in 3D space, and the texture must appear continuously

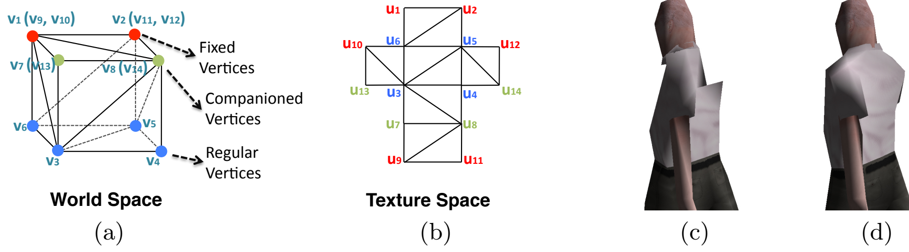


Fig. 2. (a-b) shows an example of vertex categories. (a) is the mesh after splitting vertices in world space; (b) is the texture coordinates of the mesh in texture space. The set of fixed vertices is $\{v_1, v_2, v_9, v_{10}, v_{11}, v_{12}\}$; the set of companioned vertices is $\{v_7, v_8, v_{13}, v_{14}\}$; the set of regular vertices is $\{v_3, v_4, v_5, v_6\}$. **(c-d) shows the comparison of accuracy on the deformed mesh.** (c) is the LOD based on the simplification of the static bind-pose mesh; (d) is the LOD based on the simplification considering all frames of motions. Note that both (a) and (b) are composed of 300 triangles. As a result, (d) is more accurate than (c) in visual fidelity.

crossing those edges. Thus, we collapse both the edges concurrently to avoid any distorted texturing effects .

For a deformable mesh, the actual shape of the mesh changes between frames. If we perform the simplification process only on a static shape of the mesh, the LOD generated for the actual shape may not be sufficiently accurate (see Fig. 2(c-d)). This is because when a bone of the skeleton is bent, the flat region of the static shape around this joint may significantly change to a high curvature region, where more features are demanded [3]. To solve this problem, we collapse an edge by considering the actual shapes of all frames in the motions. That means that the cost to collapse an edge is the average of the costs computed from all frames. We show an example of a sequence of LODs in Fig. 3.

Fig. 3. A sequences of LODs generated by using texture-preserved criteria. From the left to the right, the LODs have 2620, 655, 388, 276,190 triangles respectively.

5 Rendering Pipeline for Animated Characters

In this section, we describe the runtime pipeline for rendering a crowd of animated characters.

5.1 LOD Selection

At runtime, we determine the complexity of each character. In our system, the complexity is represented with appropriate vertex count and triangle count. The farther a character is from the camera, the less complexity is needed to render. A common solution to compute the complexities is a distance-based method [10]. Although it is a simple and efficient method, the major disadvantage is that an arbitrary point for each character must be chosen for distance calculation, which would lead to inaccurant results. Alternatively, we use a screen-based method utilizing the projected area of BSes on the image plane. Simuliar to [13, 16], we use Equation 1 to compute the complexities, and the total triangle count of all characters is smaller than a predefined maximal count.

$$k = N \frac{A_i^{\frac{1}{\alpha}}}{\sum_{i=1}^{l} A_i^{\frac{1}{\alpha}}} \tag{1}$$

In Equation 1, k is the appropriate vertex count computed out of l characters; N is the maximal vertex count, which is predefined according to the rendering performance or quality. A_i is the projected area of BS of the ith character. $\frac{1}{\alpha}$ is a parameter determining how the model perception contributes to the selection process (Refer to [17] for the details for choosing a value of α). By mapping k to the corresponding number of triangles, we can obtain the appropriate triangle count, q, where $q \in [1, n]$.

If a character is out of the view frustum, it is invisible to users, and we set its complexity to zero. By using NVIDIA's CUDA computing framework, we first test BSes against the view frustum in parallel. Then we apply Equation 1 for all the characters inside the view frustum.

5.2 Generating LODs By Reforming Triangles

After LOD selection, for each character, we select the successive sets of vertices and triangles from the original data. The vertex and triangle data of the mesh are re-arranged according to the order of the edge collapses in preprocess. The first vertex of the mesh is the last one collapsed; and the fist triangle of the mesh is the last one removed. Thus, according to the computed complexity, we create those two sets by picking the successive vertices and triangles starting from the first. Then, the LOD of the character is generated by reforming those selected triangles. As a result, the vertex indices of a triangle are replaced with the target indices returned from the per-vertex lookup of the collapsing information. Please refer to [13] for detailed description.

5.3 Deforming LODs

In order to animate a 3D mesh, we apply the standard smooth skinning algorithm, which uses information defined as*skin* and *skeleton*. *Skeleton* is a set of interconnected bones organized in an inverted tree hierarchy. *Skin* indicates how the vertices are influenced by the skeleton. In our system, we allow a maximum of 4 bones controlling one vertex, and each of 4 bones associates with a scaling factor, called *deforming weights*, to specify its strength influencing the vertex. Thus, we animate a character by deforming its LOD mesh with a *skeletal pose* defined by the movements of bones. In our system, we use the pre-captured motions, where a motion is composed of a sequence of skeletal poses. At runtime, a skeletal pose can be chosen from the motion data to deform the mesh.

To calculate the final positions of vertices, each vertex is transformed by applying the transformation matrices of its associated bones. We show the per-vertex calculation in Equation 2.

$$\overrightarrow{p}' = \sum_{i=1}^{4} w_{b_i} \cdot \boldsymbol{G}\boldsymbol{T}_{b_i}\boldsymbol{B}_{b_i}^{-1}\overrightarrow{p};\ \text{where}\ \sum_{i=1}^{4} w_{b_i} = 1; \qquad (2)$$

where b_i is the index of the bone influencing the vertex; $\boldsymbol{G}$ is the world transformation matrix of the character having this vertex; $\boldsymbol{T}_{b_i}$ is the transformation matrix of the bone; $\boldsymbol{B}_{b_i}^{-1}$ is the inverse binding matrix of the bone defined in the initial skeleton. $\overrightarrow{p}$ is the original position of the vertex. To do this calculation efficiently, we store the skeleton, motions and skin in texture memories of GPUs, then calculate $\overrightarrow{p}'$ by reading them in the vertex shader.

6 Experiments and Results

Our system is designed to render a large number of animated characters. In this section, we show the rendering results, and evaluate the performance.

6.1 Implementation and Experiment

We have implemented our system for crowd rendering on an workstation equipped with Intel Core i7 2.67 GHz, 12GB of RAM, and a Nvidia Quadro 5000 graphics card with 2.5GB device memory. Our program uses Nvidia CUDA Toolkit v3.2. In our experiments, we use a character composed of 1642 vertices and 2620 triangles. We applied the criteria presented in Section 4 in preprocess. As shown in Fig. 3, the continuity of the texture appearing on different version of LODs is perserved. In Fig. 4, we also show a live-captured image rendering 512 characters.

6.2 Performance Evaluation

We evaluate the performance of the rendering system by comparing with the OpenGL pseudo-instancing approach. Instancing technique is to render multiple

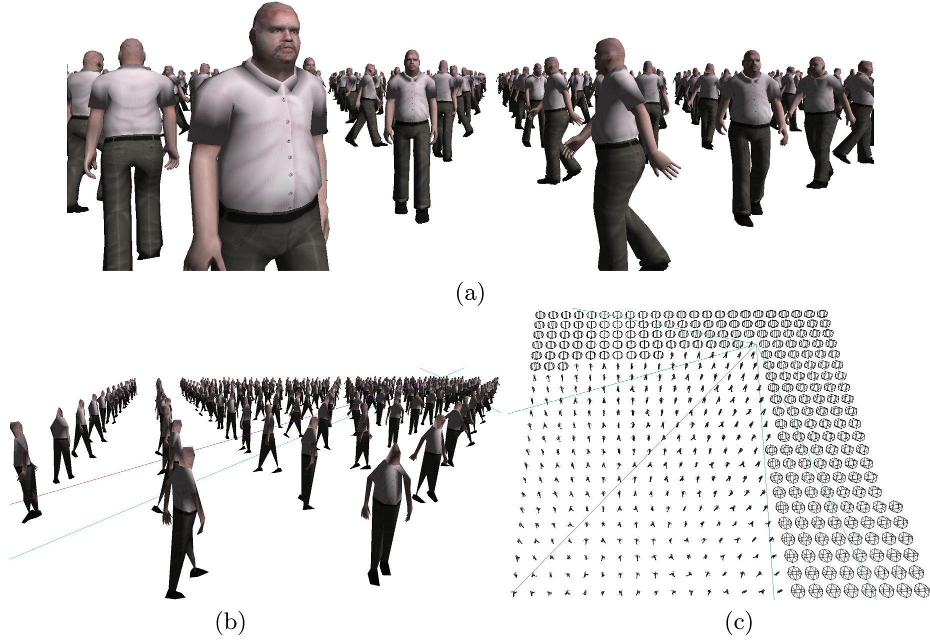


Fig. 4. An example of rendering 512 characters with 98,638 triangles in total. (a) shows the rendering result from an user camera. (b) demonstrates the texture appearance on the LODs generated based on the camera setting in (a). In (c), we show the result of view-frustum culling of (a). The spheres indicate the characters outside the view frustum.

instances of the same character with a single drawing call. In OpenGL graphics pipeline, pseudo-instancing has been widely, for example, in [11, 12, 18], where the crowd rendering is optimized by sharing vertex data, primitive counts and types, among all instances. It minimizes the amount of duplicated data. The technique accesses persistent vertex attributes associated in vertex shader, and per-instance data(e.g. world transforms) as textures.

We present the performance evaluation of our approach with varying counts of character instance in Table 1. Arbitrary number N is chosen to satisfy the certain base level of rendering quality. In our test case, we set such N that guarantees no over-simplification on the farthest characters from the camera. $\alpha = 3$ is configured to produce the equivalent result of Funkhouser's benefit function as in [4, 17]. "Updating Animation" is the time for changing the frame index of motion, and "Rendering" is the time for the process of deformation and rasterization. "# of triangles" represents the number of triangles used in our approach comparing to that of pseudo-instancing. Since our approach adjusts the complexity of character meshes at runtime, only 7.2% $\sim$ 47.7% of triangles are maintained while preserving the visual fidelity.

Table 1. Performance benchmark of our approach with varying number of characters.

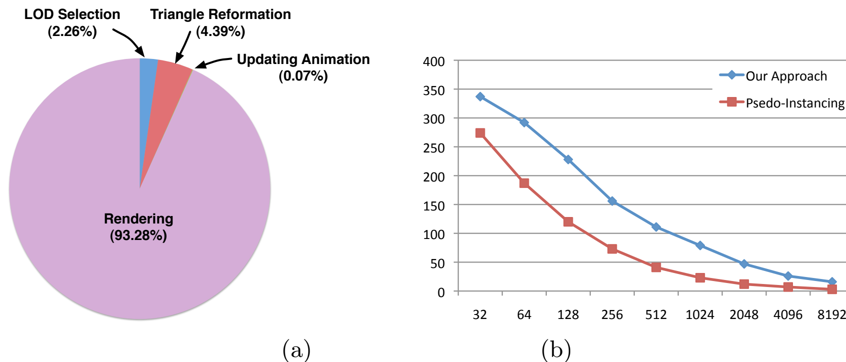
# of instances	FPS	LOD Selection	Triangle Reformation	Updating Animation	Rendering	# of triangles
32	337	0.28 ms	0.084 ms	0.0004 ms	2.603 ms	40,056/83,840
64	292	0.32 ms	0.11 ms	0.0004 ms	2.994 ms	49,236/167,680
128	228	0.34 ms	0.13 ms	0.001 ms	3.915 ms	78,058/335,360
256	156	0.35 ms	0.21 ms	0.002 ms	5.848 ms	139,642/670,720
512	111	0.42 ms	0.32 ms	0.004 ms	8.265 ms	211,685/1,341,440
1,024	79	0.43 ms	0.49 ms	0.006 ms	11.732 ms	329,594/2,682,880
2,048	47	0.44 ms	0.89 ms	0.014 ms	19.933 ms	617,031/5,365,760
4,096	26	0.48 ms	1.75 ms	0.032 ms	36.200 ms	1,182,280/10,731,520
8,192	16	0.58 ms	3.08 ms	0.054 ms	58.786 ms	1,946,977/21,463,040

"Triangle Reformation" time shows our approach scales well with the number of triangles. The number of triangles and reformation time to render $2,048$ characters are set to 1.0, and the ratios of triangles number and reformation time for the rest test cases show the linear relation in Table 2. The GPU we used has 11 stream processors in which each processor is composed of 128 computational cores. Hence, at least $11 \times 128 = 1,408$ threads need to run concurrently to prevent GPU from underutilization. In our parallel implementation, each thread performs a kernel of LOD selection and triangle reformation, therefore the test case of $2,048$ instances is chosen to the base for the comparison.

Table 2. Performance and scalability.

# of Instances	# of Triangles	Triangle Reformation	Ratio of Triangles	Ratio of Triangle Reformation
2,048	617,031	0.95 ms	1.00	1.00
4,096	1,182,280	1.75 ms	1.91	1.84
8,192	1,946,977	3.08 ms	3.15	3.24
16,384	3,103,167	5.23 ms	5.02	5.49
32,768	6,184,824	10.47 ms	10.02	11.02

Although our approach introduce additional costs on the steps of LOD selection and LOD generation, it is not the performance bottleneck of the system. As shown in Fig. 5(a), the sum of computation time spent on those two stages is 6.65% of the total time. Fig. 5(b) shows the performance comparison measured by FPS(Frames Per Second) between our approach to pseudo-instancing. The same camera setting is used for both approaches on the same number of instances. The procedure for changing the frame index of motion is also set to be identical. Our LOD-based approach achieves 1.22X to 5.33X speedup comparing to the pseudo-instancing approach.



(a) (b)

Fig. 5. (a) shows the percentages of different computation times of our approach listed in Table 1. The pie chart is generated by averaging the timing results of all nine“‘ configuration of instance counts. (b) shows the performance over different instance counts according to the FPS results in Table 1.

7 Conclusion and Future Work

We present a LOD-based real-time crowd rendering system for animated characters. Our contributions are focused on a set of texture-preserving criteria for parallel and progressive mesh simplification. In the preprocess stage, we first splitting the original vertices based on the layout of texture coordinates in order to preserve the continuity of texture appearing on the surface. Then, the vertices are classified into three different categories. Each category is set with different rules of how the vertices should be merged during the iterative process of edge collapsing. In our rendering pipeline, we generate the appropriate LODs on the fly, and deform them to obtain the actual shape of mesh for the current animation frame. We leverage the computing power of the parallel architecture of GPUs, and achieve the real time performance for our test cases.

In the future, we like to apply the texture preserving criteria to other rendering attributes, such as vertex normals. We would also like to research on the parallel algorithms for occlusion culling based on frame-to-frame coherence.

References

1. Cohen, J., Olano, M., Manocha, D.: Appearance-preserving simplification. In: Proceedings of the 25th annual conference on Computer graphics and interactive techniques. pp. 115–122. SIGGRAPH ’98, ACM, New York, NY, USA (1998), `http://doi.acm.org/10.1145/280814.280832`
2. DeCoro, C., Tatarchuk, N.: Real-time mesh simplification using the gpu. In: Proceedings of the 2007 symposium on Interactive 3D graphics and games. pp. 161–166. I3D ’07, ACM, New York, NY, USA (2007), `http://doi.acm.org/10.1145/1230100.1230128`
3. Feng, W.W., Kim, B.U., Yu, Y., Peng, L., Hart, J.: Feature-preserving triangular geometry images for level-of-detail representation of static and skinned meshes.

ACM Trans. Graph. 29, 11:1–11:13 (April 2010), http://doi.acm.org/10.1145/1731047.1731049
4. Funkhouser, T.A., Séquin, C.H.: Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In: Proceedings of the 20th annual conference on Computer graphics and interactive techniques. pp. 247–254. SIGGRAPH '93, ACM, New York, NY, USA (1993), http://doi.acm.org/10.1145/166117.166149
5. Garland, M., Heckbert, P.S.: Surface simplification using quadric error metrics. In: Proceedings of the 24th annual conference on Computer graphics and interactive techniques. pp. 209–216. SIGGRAPH '97, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (1997), http://dx.doi.org/10.1145/258734.258849
6. Giegl, M., Wimmer, M.: Unpopping: Solving the image-space blend problem for smooth discrete lod transitions. Computer Graphics Forum 26(1), 46–49 (2007), http://dx.doi.org/10.1111/j.1467-8659.2007.00943.x
7. Hoppe, H.: Progressive meshes. In: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques. pp. 99–108. SIGGRAPH '96, ACM, New York, NY, USA (1996), http://doi.acm.org/10.1145/237170.237216
8. Hu, L., Sander, P.V., Hoppe, H.: Parallel view-dependent refinement of progressive meshes. In: Proceedings of the 2009 symposium on Interactive 3D graphics and games. pp. 169–176. I3D '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1507149.1507177
9. Lindstrom, P., Turk, G.: Image-driven simplification. In: ACM Transactions on Graphics. vol. 19, pp. 204–241 (July 2000)
10. Luebke, D., Watson, B., Cohen, J.D., Reddy, M., Varshney, A.: Level of Detail for 3D Graphics. Elsevier Science Inc., New York, NY, USA (2002)
11. Millan, E., Rudomin, I.: Impostors and pseudo-instancing for gpu crowd rendering. In: Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia. pp. 49–55. GRAPHITE '06, ACM, New York, NY, USA (2006), http://doi.acm.org/10.1145/1174429.1174436
12. Park, H., Han, J.: Fast rendering of large crowds using gpu. In: Proceedings of the 7th International Conference on Entertainment Computing. pp. 197–202. ICEC '08, Springer-Verlag, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-540-89222-9_24
13. Peng, C., Cao, Y.: Gpu-based streaming for parallel level of detail on massive model rendering. Tech. Rep. TR-11-12, Computer Science, Virginia Tech (2011), http://eprints.cs.vt.edu/archive/00001158/
14. Ronfard, R., Rossignac, J., Rossignac, J.: Full-range approximation of triangulated polyhedra. In: Rossignac, J., Sillon, F. (eds.) Proceeding of Eurographics, Computer Graphics Forum. vol. 15(3), pp. C67–C76. Eurographics, Blackwell (August 1996)
15. Scherzer, D., Wimmer, M.: Frame sequential interpolation for discrete level-of-detail rendering. Computer Graphics Forum 27(4), 1175–1181 (2008), http://dx.doi.org/10.1111/j.1467-8659.2008.01255.x
16. Schmalstieg, D., Fuhrmann, A.: Coarse view-dependent levels of detail for hierarchical and deformable models. Tech. rep. (1999)
17. Wimmer, M., Schmalstieg, D.: Load balancing for smooth levels of detail. Tech. Rep. TR-186-2-98-31, Vienna University of Technology (1998)
18. Zelsnack, J.: Glsl pseudo-instancing. Tech. rep., NVIDIA Corporation (2004)