Hindawi
International Journal of Computer Games Technology
Volume 2019, Article ID 1792304, 15 pages
https://doi.org/10.1155/2019/1792304

Research Article

Real-Time Large Crowd Rendering with Efficient Character and Instance Management on GPU

Yangzi Dong and Chao Peng

Department of Computer Science, University of Alabama in Huntsville, Huntsville, AL 35899, USA

Correspondence should be addressed to Yangzi Dong; yangzi.dong@uah.edu

Received 20 October 2018; Revised 26 January 2019; Accepted 3 March 2019; Published 26 March 2019

Academic Editor: Ali Arya

Achieving the efficient rendering of a large animated crowd with realistic visual appearance is a challenging task when players interact with a complex game scene. We present a real-time crowd rendering system that efficiently manages multiple types of character data on the GPU and integrates seamlessly with level-of-detail and visibility culling techniques. The character data, including vertices, triangles, vertex normals, texture coordinates, skeletons, and skinning weights, are stored as either buffer objects or textures in accordance with their access requirements at the rendering stage. Our system preserves the view-dependent visual appearance of individual character instances in the crowd and is executed with a fine-grained parallelization scheme. We compare our approach with the existing crowd rendering techniques. The experimental results show that our approach achieves better rendering performance and visual quality. Our approach is able to render a large crowd composed of tens of thousands of animated instances in real time by managing each type of character data in a single buffer object.

1. Introduction

Crowd rendering is an important form of visual effects. In video games, thousands of computer-articulated polygonal characters with a variety of appearances can be generated to inhabit in a virtual scene like a village, a city, or a forest. Movements of the crowd are usually programmed through a crowd simulator [1–4] with given goals. To achieve a realistic visual approximation of the crowd, each character is usually tessellated with tessellation algorithms [5], which increases the character's mesh complexity to a sufficient level, so that fine geometric details and smooth mesh deformations can be preserved in the virtual scene. As a result, the virtual scene may end up with a composition of millions of, or even hundreds of millions of, vertices and triangles. Rasterizing such massive amount of vertices and triangles into pixels is a high computational cost. Also, when storing them in memory, the required amount of memory may be beyond the storage capability of a graphic hardware. Thus, in the production of video games [6–9], advanced crowd rendering technologies are needed in order to increase the rendering speed and reduce memory consumption while preserving the crowd's visual fidelity.

To improve the diversity of character appearances in a crowd, a common method is duplicating a character's mesh many times and then assigning each duplication with a different texture and a varied animation. Some advanced methods allow developers to modify the shape proportion of duplications and then retarget rigs and animations to the modified meshes [10, 11] or synthesize new motions [12, 13]. With the support of hardware-accelerated geometry-instancing and pseudo-instancing techniques [9, 14–16], multiple data of a character, including vertices, triangles, textures, skeletons, skinning weights, and animations, can be cached in the memory of a graphics processing unit (GPU). At each time when the virtual scene needs to be rendered, the renderer will alter and assemble those data dynamically without the need of fetching them from CPU main memory. However, storing the duplications on the GPU consumes a large amount of memory and limits the number of instances that can be rendered. Furthermore, even though the instancing technique reduces the CPU-GPU

communication overhead, it may suffer the lack of dynamic mesh adaption (e.g., continuous level-of-detail).

In this work, we present a rendering system, which achieves a real-time rendering rate for a crowd composed of tens of thousands of animated characters. The system ensures a fully utilization of GPU memory and computational power through the integration with continuous level-of-detail (LOD) and View-Frustum Culling techniques. The size of memory allocated for each character is adjusted dynamically in response to the change of levels of detail, as the camera's viewing parameters change. The scene of the crowd may end up with more than one hundred million triangles. Different from existing instancing techniques, our approach is capable of rendering all different characters through a single buffer object for each type of data. The system encapsulates multiple data of each unique source characters into buffer objects and textures, which can then be accessed quickly by shader programs on the GPU as well as maintained efficiently by a general-purpose GPU programming framework.

The rest of the paper is organized as follows. Section 2 reviews the previous works about crowd simulation and crowd rendering. Section 3 gives an overview of our system's rendering pipeline. In Section 4, we describe fundamentals of continues LOD and animation techniques and discuss their parallelization on the GPU. Section 5 describes how to process and store the source character's multiple data and how to manage instances on the GPU. Section 6 presents our experimental results and compares our approach with the existing crowd rendering techniques. We conclude our work in Section 7.

2. Related Work

Simulation and rendering are two primary computing components in a crowd application. They are often tightly integrated as an entity to enable a special form of *in situ* visualization, which in general means data is rendered and displayed by a renderer in real time while a simulation is running and generating new data [17–19]. One example is the work presented by Hernandez et al. [20] that simulated a wandering crowd behavior and visualized it using animated 3D virtual characters on GPU clusters. Another example is the work presented by Perez et al. [21] that simulated and visualized crowds in a virtual city. In this section, we first briefly review some previous work contributing to crowd simulation. Then, more related to our work, we focus on acceleration techniques contributing to crowd rendering, including level-of-detail (LOD), visibility culling, and instancing techniques.

A crowd simulator uses macroscopic algorithms (e.g., continuum crowds [22], aggregate dynamics [23], vector fields [24], and navigation fields [25]) or microscopic algorithms (e.g., morphable crowds [26] and socially plausible behaviors [27]) to create crowd motions and interactions. Outcomes of the simulator are usually a successive sequence of time frames, and each frame contains arrays of positions and orientations in the 3D virtual environment. Each pair of position and orientation information defines the global status of a character at a given time frame. McKenzie et al. [28] developed a crowd simulator to generate noncombatant civilian behaviors which is interoperable with a simulation of modern military operations. Zhou et al. [29] classified the existing crowd modeling and simulation technologies based on the size and time scale of simulated crowds and evaluated them based on their flexibility, extensibility, execution efficiency, and scalability. Zhang et al. [30] presented a unified interaction framework on GPU to simulate the behavior of a crowd at interactive frame rates in a fine-grained parallel fashion. Malinowski et al. [31] were able to perform large scale simulations that resulted in tens of thousands of simulated agents.

Visualizing a large number of simulated agents using animated characters is a challenging computing task and worth an in-depth study. Beacco et al. [9] surveyed previous approaches for real-time crowd rendering. They reviewed and examined existing acceleration techniques and pointed out that LOD techniques have been used widely in order to achieve high rendering performance, where a far-away character can be represented with a coarse version of the character as the alternative for rendering. A well-known approach is using discrete LOD representations, which are a set of offline simplified versions of a mesh. At the rendering stage, the renderer selects a desired version and renders it without any additional processing cost at runtime. However, Discrete LODs require too much memory for storing all simplified versions of the mesh. Also, as mentioned by Cleju and Saupe [32], discrete LODs could cause "popping" visual artifacts because of the unsmooth shape transition between simplified versions. Dobbyn et al. [33] introduced a hybrid rendering approach that combines image-based and geometry-based rendering techniques. They evaluated the rendering quality and performance in an urban crowd simulation. In their approach, the characters in the distance were rendered with image-based LOD representations, and they were switched to geometric representations when they were within a closer distance. Although the visual quality seemed better than using discrete LOD representations, popping artifacts also occurred when the renderer switches content between image representations and geometric representations. Ulicny et al. [34] presented an authoring tool to create crowd scenes of thousands of characters. To provide users immediate visual feedback, they used low-poly meshes for source characters. The meshes were kept in OpenGL's display lists on GPU for fast rendering.

Characters in a crowd are polygonal meshes. The mesh is rigged by a skeleton. Rotations of the skeleton's joints transform surrounding vertices, and subsequently the mesh can be deformed to create animations. While LOD techniques for simplifying general polygonal meshes have been studied maturely (e.g., progressive meshes [35], quadric error metrics [36]), not many existing works studied how to simplify animated characters. Landreneau and Schaefer [37] presented mesh simplification criteria to preserve deforming features of animations on simplified versions of the mesh. Their approach was developed based on quadric error metrics, and they added simplification criteria with the consideration of vertices' skinning weights from the joints of the skeleton and

the shape deviation between the character's rest pose and a deformed shape in animations. Their approach produced more accurate animations for dynamically simplified characters than many other LOD-based approaches, but it caused a higher computational cost, so it may be challenging to integrate their approach into a real-time application. Willmott [38] presented a rapid algorithm to simplify animated characters. The algorithm was developed based on the idea of vertex clustering. The author mentioned the possibility of implementing the algorithm on the GPU. However, in comparison to the algorithm with progressive meshes, it did not produce well-simplified characters to preserve fine features of character appearance. Feng et al. [39] employed triangular-char geometry images to preserve the features of both static and animated characters. Their approach achieved high rendering performance by implement geometry images with multiresolutions on the GPU. In their experiment, they demonstrated a real-time rendering rate for a crowd composed of 15.3 million triangles. However, there could be a potential LOD adaptation issue if the geometry images become excessively large. Peng et al. [8] proposed a GPU-based LOD-enabled system to render crowds along with a novel texture-preserving algorithm on simplified versions of the character. They employed a continuous LOD technique to refine or reduce mesh details progressively during the runtime. However, their approach was based on the simulation of single virtual humans. Instantiating and rendering multiple types of characters were not possible in their system. Savoy et al. [40] presented a web-based crowd rendering system that employed the discrete LOD and instancing techniques.

Visibility culling technique is another type of acceleration techniques for crowd rendering. With visibility culling, a renderer is able to reject a character from the rendering pipeline if it is outside the view frustum or blocked by other characters or objects. Visibility culling techniques do not cause any loss of visual fidelity on visible characters. Tecchia et al. [41] performed efficient occlusion culling for a highly populated scene. They subdivided the virtual environmental map into a 2D grid and used it to build a KD-tree of the virtual environment. The large and static objects in the virtual environment, such as buildings, were used as occluders. Then, an occlusion tree was built at each frame and merged with the KD-tree. Barczak et al. [42] integrated GPU-accelerated View-Frustum Culling and occlusion culling techniques into a crowd rendering system. The system used a vertex shader to test whether or not the bounding sphere of a character intersects with the view frustum. A hierarchical Z buffer image was built dynamically in a vertex shader in order to perform occlusion culling. Hernandez and Isaac Rudomin [43] combined View-Frustum Culling and LOD Selection. Desired detail levels were assigned only to the characters inside the view frustum.

Instancing techniques have been commonly used for crowd rendering. Their execution is accelerated by GPUs with graphics API such as DirectX and OpenGL. Zelsnack [44] presented coding details of the pseudo-instancing technique using OpenGL shader language (GLSL). The pseudo-instancing technique requires per-instance calls sent to and executed on the GPU. Carucci [45] introduced the geometry-instancing technique which renders all vertices and triangles of a crowd scene through a geometry shader using one call. Millan and Rudomin [14] used the pseudo-instancing technique for rendering full-detail characters which were closer to the camera. The far-away characters with low details were rendered using impostors (an image-based approach). Ashraf and Zhou [46] used a hardware-accelerated method through programmable shaders to animated crowds. Klein et al. [47] presented an approach to render configurable instances of 3D characters for the web. They improved XML3D to store 3D content in a more efficient way in order to support an instancing-based rendering mechanism. However, their approach lacked support for multiple character assets.

3. System Overview

Our crowd rendering system first preprocesses source characters and then performs runtime tasks on the GPU. Figure 1 illustrates an overview of our system. Our system integrates View-Frustum Culling and continuous LOD techniques.

At the preprocessing stage, a fine-to-coarse progressive mesh simplification algorithm is applied to every source character. In accordance with the edge-collapsing criteria [8, 35], the simplification algorithm selects edges and then collapses them by merging adjacent vertices iteratively and then removes the triangles containing the collapsed edges. The edge-collapsing operations are stored as data arrays on the GPU. Vertices and triangles can be recovered by splitting the collapsed edges and are restored with respect to the order of applying coarse-to-fine splitting operations. Vertex normal vectors are used in our system to determine a proper shading effect for the crowd. A bounding sphere is computed for each source character. It tightly encloses all vertices in all frames of the character's animation. The bounding sphere will be used during the runtime to test an instance against the view frustum. Note that bounding spheres may be in different sizes because the sizes of source characters may be different. Other data including textures, UVs, skeletons, skinning weights, and animations are packed into textures. They can be accessed quickly by shader programs and the general-purpose GPU programming framework during the runtime.

The runtime pipeline of our system is executed on the GPU through five parallel processing components. We use an *instance ID* in shader programs to track the index of each instance, which corresponds to the occurrence of a source character at a global location and orientation in the virtual scene. A unique *source character ID* is assigned to each source character, which is used by an instance to index back to the source character that is instantiated from. We assume that the desired number of instances is provided by users as a parameter in the system configuration. The global positions and orientations of instances simulated from a crowd simulator are passed into our system as input. They determine where the instances should occur in the virtual scene. The component of View-Frustum Culling determines the visibility of instances. An instance will be considered to be visible if its bounding sphere is inside or intersects with the

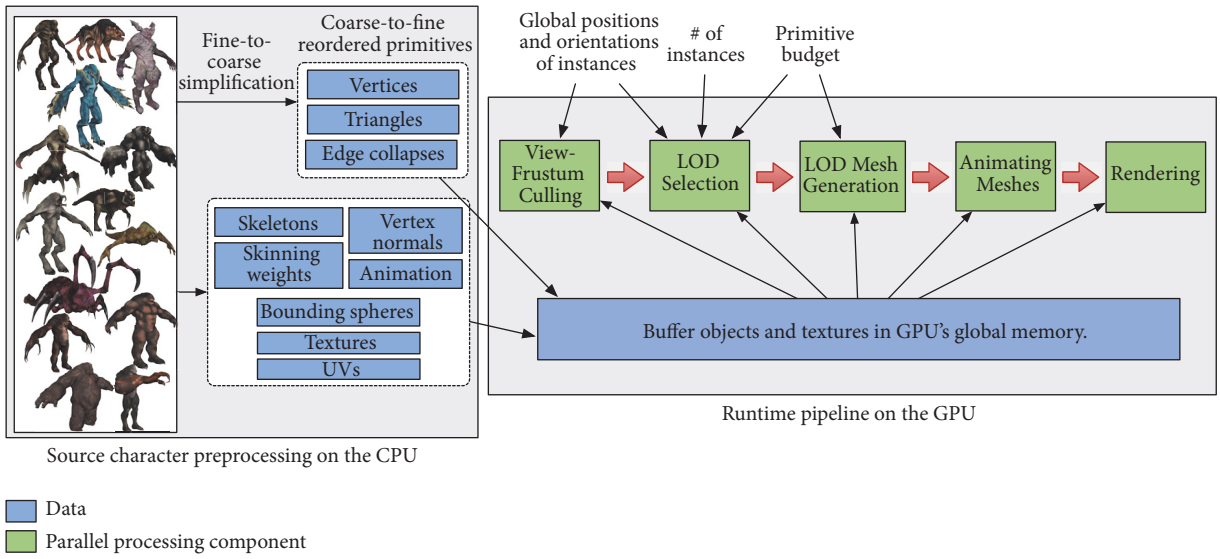

Figure 1: The overview of our system.

view frustum. The component of LOD Selection determines the desired detail level of the instances. It is executed with an instance-level parallelization. A detail level is represented as the numbers of vertices and triangles selected which are assembled from the vertex and triangle repositories. The component of LOD Mesh Generation produces LOD meshes using the selected vertices and triangles. The size of GPU memory may not be enough to store all the instances at their finest levels. Thus, a configuration parameter called the primitive budget is passed into the runtime pipeline as a global constraint to ensure the generated LOD meshes fit into the GPU memory. The component of Animating Meshes attaches skeletons and animations to the simplified versions (LOD meshes) of the instances. At the end of the pipeline, the rendering component rasterizes the LOD meshes along with appropriate textures and UVs and displays the result on the screen.

4. Fundamentals of LOD Selection and Character Animation

During the step of preprocessing, the mesh of each source character is simplified by collapsing edges. Same as existing work, the collapsing criteria in our approach preserves features at high curvature regions [39] and avoids collapsing the edges on or crossing texture seams [8]. Edges are collapsed one-by-one. We utilized the same method presented in [48], which saved collapsing operations into an array structure suitable for the GPU architecture. The index of each array element represents the source vertex and the value of the element represents the target vertex it merges to. By using the array of edge collapsing, the repositories of vertices and triangles are rearranged in an increasing order, so that, at runtime, the desired complexity of a mesh can be generated by selecting a successive sequence of vertices and triangles from the repositories. Then, the skeleton-based animations are applied to deform the simplified meshes. Figure 2 shows the different levels of detail of several source characters that are used in our work. In this section, we brief the techniques of LOD Selection and character animation.

4.1. LOD Selection. Let us denote K as the total number of instances in the virtual scene. A desired level of details for an instance can be represented as the pair of $\{vNum, tNum\}$, where $vNum$ is the desired number of vertices, and $tNum$ is the desired number of triangles. Given a value of $vNum$, the value of $tNum$ can be retrieved from the prerecorded edge-collapsing information [48]. Thus, the goal of LOD Selection is to determine an appropriate value of $vNum$ for each instance, with considerations of the available GPU memory size and the instance's spatial relationship to the camera. If an instance is outside the view frustum, $vNum$ is set to zero. For the instances inside the view frustum, we used the LOD Selection metric in [48] to compute $vNum$, as shown in

$$vNum_i = N \frac{w_i^{1/\alpha}}{\sum_{i=1}^{K} w_i^{1/\alpha}}, \quad \text{where } w_i = \beta \frac{A_i}{D_i} P_i^{\beta}, \ \beta = \alpha - 1 \tag{1}$$

Equation (1) is the same as the first-pass algorithm presented by Peng and Cao [48]. It originates from the model perception method presented by Funkhouser et al. [49] and is improved by Peng and Cao [48, 50] to accelerate the rendering of large CAD models. We found that (1) is also a suitable metric for the large crowd rendering. In the equation, N refers to the total number of vertices that can be retained on the GPU, which is a user-specified value computed based on the available size of GPU memory. The value of N can be tuned to balance the rendering performance and visual quality. w_i is the weight computed with the projected area of the bounding sphere of the ith instance on the screen (A_i) and

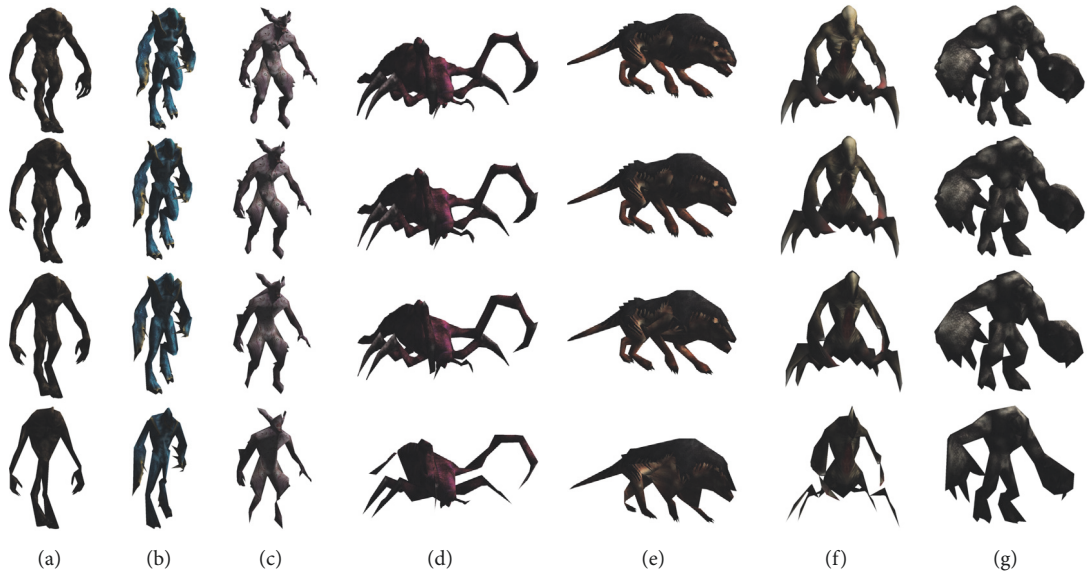

Figure 2: The examples showing different levels of detail for seven source characters. From top to bottom, the numbers of triangles are as follows: (a) Alien: 5,688, 1,316, 601, and 319; (b) Bug: 6,090, 1,926, 734, and 434; (c) Daemon: 6,652, 1,286, 612, and 444; (d) Nasty: 6,848, 1,588, 720, and 375; (e) Ripper Dog: 4,974, 1,448, 606, and 309; (f) Spider: 5,868, 1,152, 436, and 257; (g) Titan: 6,518, 1,581, 681, and 362.

the distance to the camera (D_i). α is the perception parameter introduced by Funkhouser et al. [49]. In our work, the value of α is set to 3.

With $vNum$ and $tNum$, the successive sequences of vertices and triangles are retrieved from the vertex and triangle repositories of the source character. By applying the parallel triangle reformation algorithm [48, 50], the desired shape of the simplified mesh is generated using the selected vertices and triangles.

4.2. Animation. In order to create character animations, each LOD mesh has to be bound to a skeleton along with skinning weights added to influence the movement of the mesh's vertices. As a result, the mesh will be deformed by rotating joints of the skeleton. As we mentioned earlier, each vertex may be influenced by a maximum of four joints. We want to note that the vertices forming the LOD mesh are a subset of the original vertices of the source character. There is not any new vertex introduced during the preprocess of mesh simplification. Because of this, we were able to use original skinning weights to influence the LOD mesh. When transformations defined in an animation frame are loaded on the joints, the final vertex position will be computed by summing the weighted transformations of the skinning joints. Let us denote each of the four joints influencing a vertex v as Jnt_i, where $i \in [0, 3]$. The weight of Jnt_i on the vertex v is denoted as W_{Jnt_i}. Thus, the final position of the vertex v, denoted as P'_v, can be computed by using

$$P'_v = G\sum_{i=0}^{3} W_{Jnt_i} T_{Jnt_i} B^{-1}_{Jnt_i} P_v; \quad \text{where } \sum_{i=0}^{3} W_{Jnt_i} = 1 \qquad (2)$$

In (2), P_v is the vertex position at the time when the mesh is bound to the skeleton. When the mesh is first loaded without the use of animation data, the mesh is placed in the initial binding pose. When using an animation, the inverse of the binding pose needs to be multiplied by an animated pose. This is reflected in the equation, where $B^{-1}_{Jnt_i}$ is the inverse of binding transformation of the joint Jnt_i, and T_{Jnt_i} represents the transformation of the joint Jnt_i from the current frame of the animation. G is the transformation representing the instance's global position and orientation. Note that the transformations $B^{-1}_{Jnt_i}$, T_{Jnt_i}, and G are represented in the form of 4×4 matrix. The weight W_{Jnt_i} is a single float value, and the four weight values must sum to 1.

5. Source Character and Instance Management

Geometry-instancing and pseudo-instancing techniques are the primary solutions for rendering a large number of instances, while allowing the instances to have different global transformations. The pseudo-instancing technique is used in OpenGL and calls instances' drawing functions one-by-one. The geometry-instancing technique is included in Direct3D since the version 9 and in OpenGL since version 3.3. It advances the pseudo-instancing technique in terms of reducing the number of drawing calls. It supports the use of a single drawing call for instances of a mesh and therefore reduces the communication cost of sending call requests from CPU to GPU and subsequently increases the performance. As regards data storage on the GPU, buffer objects are used for shader programs to access and update data quickly. A buffer object is a continuous memory block on the GPU and allows the renderer to rasterize data in a retained mode. In particular, a *vertex buffer object* (VBO) stores vertices. An *index buffer object* (IBO) stores indices of vertices that form triangles or other polygonal types used in our system.

In particular, the geometry-instancing technique requires a single copy of vertex data maintained in the VBO, a single copy of triangle data maintained in the IBO, and a single copy of distinct world transformations of all instances. However, if the source character has a high geometric complexity and there are lots of instances, the geometry-instancing technique may make the uniform data type in shaders hit the size limit, due to the large amount of vertices and triangles sent to the GPU. In such case, the drawing call has to be broken into multiple calls.

There are two types of implementations for instancing techniques: *static batching* and *dynamic batching* [45]. The single-call method in the geometry-instancing technique is implemented with static batching, while the multicall method in both the pseudo-instancing and geometry-instancing techniques are implemented with dynamic batching. In static batching, all vertices and triangles of the instances are saved into a VBO and IBO. In dynamic batching, the vertices and triangles are maintained in different buffer objects and drawn separately. The implementation with static batching has the potential to fully utilize the GPU memory, while dynamic batching would underutilize the memory. The major limitation of static batching is the lack of LOD and skinning supports. This limitation makes the static batching not suitable for rendering animated instances, though it has been proved to be faster than dynamic batching in terms of the performance of rasterizing meshes.

In our work, the storage of instances is managed similarly to the implementation of static batching, while individual instances can still be accessed similarly to the implementation of dynamic batching. Therefore, our approach can be seamlessly integrated with LOD and skinning techniques, while taking the use of a single VBO and IBO for fast rendering. This section describes the details of our contributions for character and instance management, including texture packing, UV-guided mesh rebuilding, and instance indexing.

5.1. Packing Skeleton, Animation, and Skinning Weights into Textures. Smooth deformation of a 3D mesh is a computationally expensive process because each vertex of the mesh needs to be repositioned by the joints that influence it. We packed the skeleton, animations, and skinning weights into 2D textures on the GPU, so that shader programs can access them quickly. The skeleton is the binding pose of the character. As explained in (2), the inverse of the binding pose's transformation is used during the runtime. In our approach, we stored this inverse into the texture as the skeletal information. For each joint of the skeleton, instead of storing individual translation, rotation, and scale values, we stored their composed transformation in the form of a 4×4 matrix. Each joint's binding pose transformation matrix takes four RGBA texels for storage. Each RGBA texel stores a row of the matrix. Each channel stores a single element of the matrix. By using OpenGL, matrices are stored as the format of GL_RGBA32F in the texture, which is a 32-bit floating-point type for each channel in one texel. Let us denote the total number of joints in a skeleton as K. Then, the total number of texels to store the entire skeleton is $4K$.

We used the same format for storing the skeleton to store an animation. Each animation frame needs $4K$ texels to store the joints' transformation matrices. Let us denote the total number of frames in an animation as Q. Then, the total number of texels for storing the entire animation is $4KQ$. For each animation frame, the matrix elements are saved into successive texels in the row order. Here we want to note that each animation frame starts from a new row in the texture.

The skinning weights of each vertex are four values in the range of $[0, 1]$, where each value represents the influencing percentage of a skinning joint. For each vertex, the skinning weights require eight data elements, where the first four data elements are joint indices, and the last four are the corresponding weights. In other words, each vertex requires two RGBA texels to store the skinning weights. The first texel is used to store joint indices, and the second texel is used to store weights.

5.2. UV-Guided Mesh Rebuilding. A 3D mesh is usually a seamless surface without boundary edges. The mesh has to be cut and unfolded into 2D flatten patches before a texture image can be mapped onto it. To do this, some edges have to be selected properly as boundary edges, from which the mesh can be cut. The relationship between the vertices of a 3D mesh and 2D texture coordinates can be described as a texture mapping function $F(x, y, z) \longrightarrow \{(s_i, t_i)\}$. Inner vertices (those not on boundary edges) have a one-to-one texture mapping. In other words, each inner vertex is mapped to a single pair of texture coordinates. For the vertices on boundary edges, since boundary edges are the cutting seams, a boundary vertex needs to be mapped to multiple pairs of texture coordinates. Figure 3 shows an example that unfolds a cube mesh and maps it into a flatten patch in 2D texture space. In the figure, u_i stands for a point in the 2D texture space. Each vertex on the boundary edges is mapped to more than one points, which are $F(v_0) = \{u_1, u_3\}$, $F(v_3) = \{u_{10}, u_{12}\}$, $F(v_4) = \{u_0, u_4, u_6\}$, and $F(v_7) = \{u_7, u_9, u_{13}\}$.

In a hardware-accelerated renderer, texture coordinates are indexed from a buffer object, and each vertex should associate with a single pair of texture coordinates. Since the texture mapping function produces more than one pairs of texture coordinates for boundary vertices, we conducted a mesh rebuilding process to duplicate boundary vertices and mapped each duplicated one to a unique texture point. By doing this, although the number of vertices is increased due to the cuttings on boundary edges, the number of triangles is the same as the number of triangles in the original mesh. In our approach, we initialized two arrays to store UV information. One array stores texture coordinates, the other array stores the indices of texture points with respect to the order of triangle storage. Algorithm 1 shows the algorithmic process to duplicate boundary vertices by looping through all triangles. In the algorithm, *Verts* is the array of original vertices storing 3D coordinates (x, y, z). *Tris* is the array of original triangles storing the sequence of vertex indices. Similar to *Tris*, *TexInx* is the array of indices of texture points in 2D texture space and represents the same triangular topology as the mesh. Note that the order of triangle storage

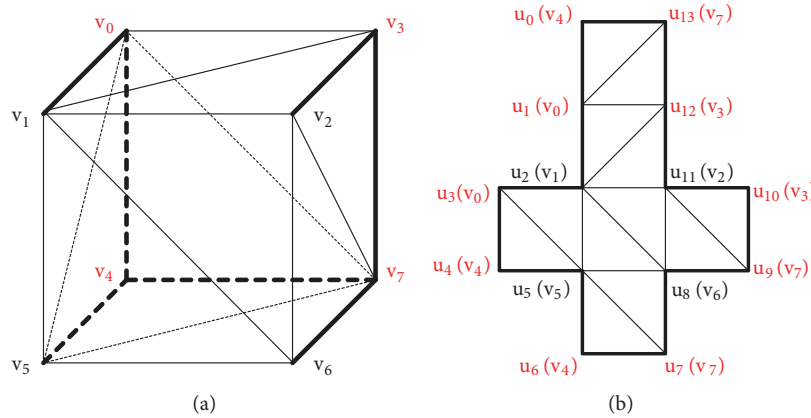

FIGURE 3: An example showing the process of unfolding a cube mesh and mapping it into a flatten patch in 2D texture space. (a) is the 3D cube mesh formed by 8 vertices and 6 triangles. (b) is the unfolded texture map formed by 14 pairs of texture coordinates and 6 triangles. Bold lines are the boundary edges (seams) to cut the cube, and the vertices in red are boundary vertices that are mapped into multiple pairs of texture coordinates. In (b), u_i stands for a point (s_i, t_i) in 2D texture space, and v_i in the parenthesis is the corresponding vertex in the cube.

RebuildMesh(
Input: *Verts*, *Tris*, *TexCoords*, *Norms*, *TexInx*, *oriTriNum*, *texCoordNum*;
Output: $Verts'$, $Tris'$, $Norms'$)

(1) $checked[texCoordNum] \longleftarrow false$;
(2) **for** each triangle i of $oriTriNum$ **do**
(3) **for** each vertex j in the ith triangle **do**
(4) $texPoint\_id \longleftarrow TexInx[3 * i + j]$;
(5) **if** $checked[texPoint\_id] = false$ **then**
(6) $checked[texPoint\_id] \longleftarrow true$;
(7) $vert\_id \longleftarrow Tris[3 * i + j]$;
(8) **for** each coordinate k in the kth vertex **do**
(9) $Verts'[3 * texPoint\_id + k] = Verts[3 * vert\_id + k]$;
(10) $Norms'[3 * texPoint\_id + k] = Norms[3 * vert\_id + k]$;
(11) **end for**
(12) **end if**
(13) **end for**
(14) **end for**
(15) $Tris' \longleftarrow TexInx$;

ALGORITHM 1: UV-Guided mesh rebuilding algorithm.

for the mesh is the same as the order of triangle storage for the 2D texture patches. *Norms* is the array of vertex normal vectors. *oriTriNum* is the total number of original triangles, and *texCoordNum* is the number of texture points in 2D texture space.

After rebuilding the mesh, the number of vertices in $Verts'$ will be identical to the number of texture points *texCoordNum*, and the array of triangles ($Tris'$) is replaced by the array of indices of the texture points (*TexInx*).

5.3. Source Character and Instance Indexing. After applying the data packing and mesh rebuilding methods presented in Sections 5.1 and 5.2, the multiple data of a source character are organized into GPU-friendly data structures. The character's skeleton, skinning weights, and animations are packed into textures and read-only in shader programs on the GPU. The vertices, triangles, texture coordinates, and vertex normal vectors are stored in arrays and retained on the GPU. During the runtime, based on the LOD Selection result (see Section 4.1), a successive subsequence of vertices, triangles, texture coordinates, and vertex normal vectors are selected for each instance and maintained as single buffer objects. As mentioned in Section 4.1, the simplified instances are constructed in a parallel fashion through a general-purpose GPU programming framework. Then, the framework interoperates with the GPU's shader programs and allows shaders to perform rendering tasks for the instances. Because continuous LOD and animated instancing techniques are assumed to be used in our approach, instances have to be rendered one-by-one, which is the same as the way of rendering animated instances in geometry-instancing and pseudo-instancing techniques. However, our approach needs

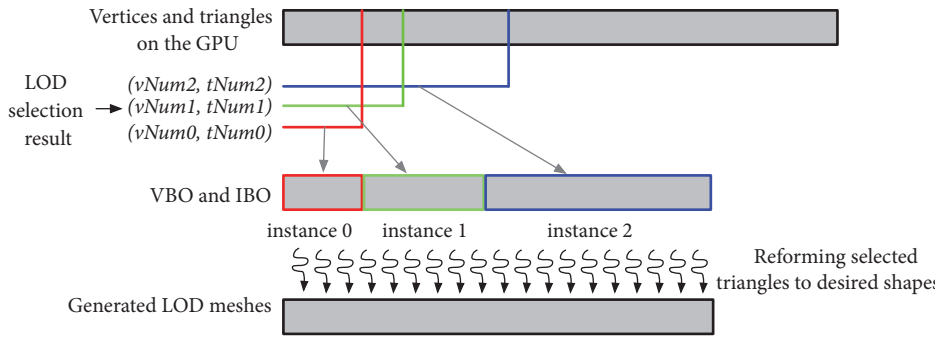

FIGURE 4: An example illustrating the data structures for storing vertices and triangles of three instances in VBO and IBO, respectively. Those data are stored on the GPU and all data operations are executed in parallel on the GPU. The VBO and IBO store data for all instances that are selected from the array of original vertices and triangles of the source characters. *vNum* and *tNum* arrays are the LOD section result.

to construct the data within one execution call, rather than dealing with per-instance data.

Figure 4 illustrates the data structures of storing VBO and IBO on the GPU. Based on the result of LOD Selection, each instance is associated with a *vNum* and a *tNum* (see Section 4.1) that represent the amount of vertices and triangles selected based on the current view setting. We employed CUDA Thrust [51] to process the arrays of *vNum* and *tNum* using the prefix sum algorithm in a parallel fashion. As a result, for example, each $vNum[i]$ represents the offset of vertex count prior to the ith instance, and the number of vertices for the ith instance is $(vNum[i+1] - vNum[i])$.

Algorithm 2 describes the vertex transformation process in parallel in the vertex shader. It transforms the instance's vertices to their destination positions while the instance is being animated. In the algorithm, *charNum* represents the total number of source characters. The inverses of the binding pose skeletons are a texture array denoted as $invBindPose[charNum]$. The skinning weights are a texture array denoted as $skinWeights[charNum]$. We used a walk animation for each source character, and the texture array of the animations is denoted as $anim[charNum]$. *gMat* is the global 4×4 transformation matrix of the instance in the virtual scene. This algorithm is developed based on the data packing formats described in Section 5.1. Each source character is assigned with a unique source character ID, denoted as *c_id* in the algorithm. The drawing calls are issued per instance, so *c_id* is passed into the shader as an input parameter. The function of *GetLoc*() computes the coordinates in the texture space to locate which texel to fetch. The input of *GetLoc*() includes the current vertex or joint index (*id*) that needs to be mapped, the width (w) and height (h) of the texture, and the number of texels (*dim*) associating with the vertex or joint. For example, to retrieve a vertex's skinning weights, the *dim* is set to 2; to retrieve a joint's transformation matrix, the *dim* is set to 4. In the function of *TransformVertices*(), vertices of the instance are transformed in a parallel fashion by the composed matrix (*composedMat*) computed from a weighted sum of matrices of the skinning joints. The function *Sample*() takes a texture and the coordinates located in the texture space as input. It returns the values encoded in texels. The *Sample*() function is usually provided in a shader programming framework. Different from the rendering of static models, animated characters change geometric shapes over time due to continuous pose changes in the animation. In the algorithm, *f_id* stands for the current frame index of the instance's animation. *f_id* is updated in the main code loop during the execution of the program.

6. Experiment and Analysis

We implemented our crowd rendering system on a workstation with Intel i7-5930K 3.50GHz PC with 64GBytes of RAM and an Nvidia GeForce GTX980 Ti 6GB graphics card. The rendering system is built with Nvidia CUDA Toolkit 9.2 and OpenGL. We employed 14 unique source characters. Table 1 shows the data configurations of those source characters. Each source character is articulated with a skeleton and fully skinned and animated with a walk animation. Each one contains an unfolded texture UV set along with a texture image at the resolution of 2048×2048. We stored those source characters on the GPU. Also, all source characters have been preprocessed by the mesh simplification and animation algorithms described in Section 4. We stored the edge-collapsing information and the character bounding spheres in arrays on the GPU. In total, the source characters require 184.50MB memory for storage. The size of mesh data is much smaller than the size of texture images. The mesh data requires 16.50MB memory for storage, which is only 8.94% of the total memory consumed. At initialization of the system, we randomly assigned a source character to each instance.

6.1. Visual Fidelity and Performance Evaluations. As defined in Section 4.1, N is a memory budget parameter that determines the geometric complexity and the visual quality of the entire crowd. For each instance in the crowd, the corresponding bounding sphere is tested against the view frustum to determine its visibility. The value of N is only distributed across visible instances.

We created a walkthrough camera path for the rendering of the crowd. The camera path emulates a gaming navigation behavior and produces a total of 1,000 frames. The entire

GetLoc(
Input: w, h, id, dim
Output: $unit\_x, unit\_y, x, y$)

(1) $unit\_x \longleftarrow 1/w$;
(2) $unit\_y \longleftarrow 1/h$;
(3) $x \longleftarrow dim * (id\%(w/dim)) * unit\_x$;
(4) $y \longleftarrow (id/(w/dim)) * unit\_y$;

TransformVertices(
Input: $Verts'$, $invPose[charNum]$, $skinWeights[charNum]$, $anim[charNum]$, $gMat$, c_id, f_id;
Output: $Verts''$)

(1) **for** each vertex v_i in $Verts'$ **in parallel do**
(2) $Matrix4 \times 4 : composedMat \longleftarrow identity$;
(3) $\{unit\_x, unit\_y, x, y\} \longleftarrow GetLoc(skinweights[c\_id].width, skinweights[c\_id].height, i, 2)$;
(4) $h \longleftarrow skinweights[c\_id].height$;
(5) $Vector4 : jntInx \longleftarrow Sample(skinWeights[c\_id], (0 * unit\_x + x), y)$;
(6) $Vector4 : weights \longleftarrow Sample(skinWeights[c\_id], (1 * unit\_x + x), y)$;
(7) **for** each j in 4 **do**
(8) $Matrix4 \times 4 : invBindMat$;
(9) $\{unit\_x, unit\_y, x, y\} \longleftarrow GetLoc(invPose[c\_id].width, invPose[c\_id].height, jntInx[j], 4)$;
(10) **for** each k in 4
(11) $invBindMat_{row[k]} \longleftarrow Sample(invPose[c\_id], (k * unit\_x + x), y)$;
(12) **end for**
(13) $\{unit\_x, unit\_y, x, y\} \longleftarrow GetLoc(anim[c\_id].width, anim[c\_id].height, jntInx[j], 4)$;
(14) $Offset \longleftarrow f\_id * (totalJntNum/anim[c\_id].width/4) * unit\_y$;
(15) $Matrix4 \times 4 : animMat$;
(16) **for** each k in 4 **do**
(17) $animMat_{row[k]} \longleftarrow Sample(anim[c\_id], (k * unit\_x + x), Offset + y)$;
(18) **end for**
(19) $composedMat += weights[j] * animMat * invBindMat$;
(20) **end for**
(21) $Verts''[i] = modelViewProjectionMat * gMat * composedMat * Verts'[i]$;
(22) **end for**

Algorithm 2: Transforming vertices of an instance in vertex shader.

Table 1: The data information of the source characters used in our experiment. Note that the data information represents the characters that are prior to the process of UV-guided mesh rebuilding (see Section 5.2).

Names of Source Characters	# of Vertices	# of Triangles	# of texture UVs	# of Joints
Alien	2846	5688	3334	64
Bug	3047	6090	3746	44
Creepy	2483	4962	2851	67
Daemon	3328	6652	4087	55
Devourer	2975	5946	3494	62
Fat	2555	5106	2960	50
Mutant	2265	4526	2649	46
Nasty	3426	6848	3990	45
Pangolin	2762	5520	3257	49
Ripper Dog	2489	4974	2982	48
Rock	2594	5184	3103	62
Spider	2936	5868	3374	38
Titan	3261	6518	3867	45
Troll	2481	4958	2939	55

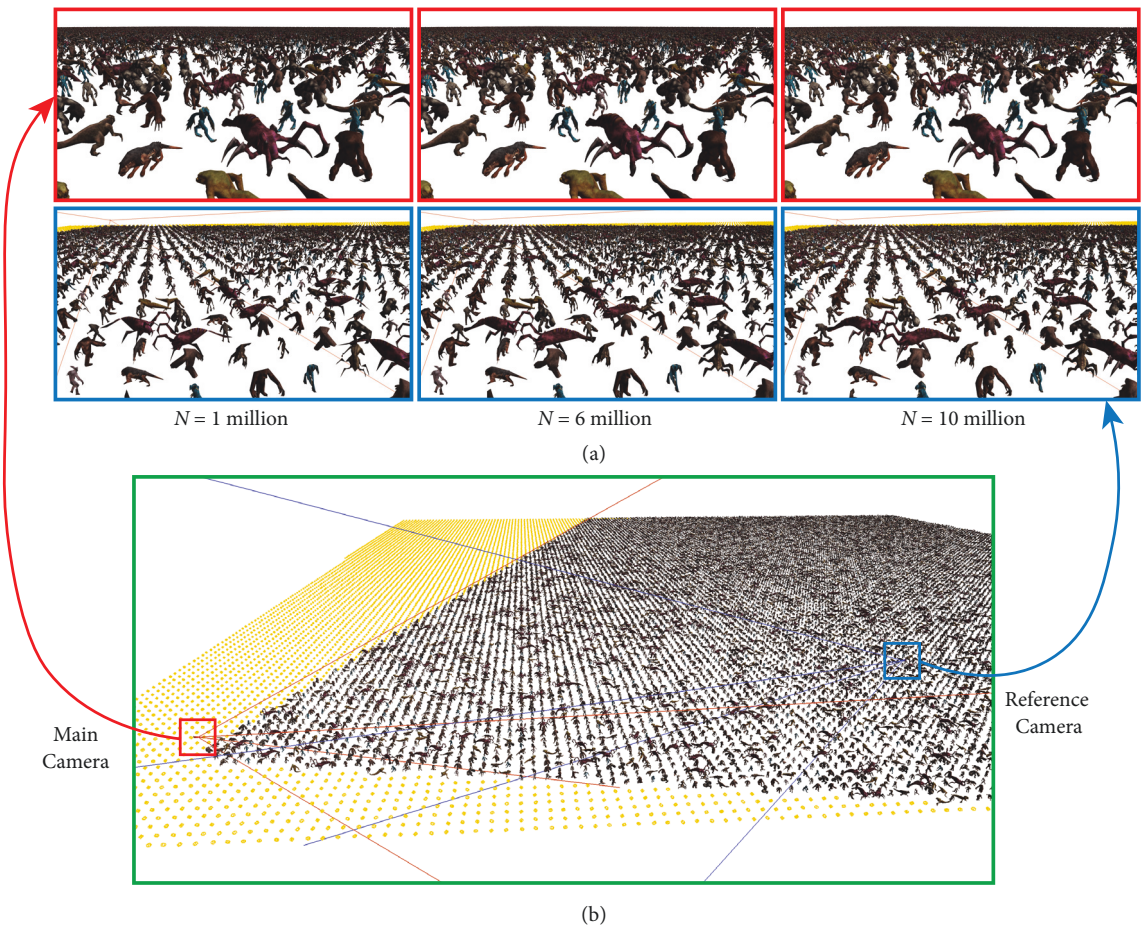

FIGURE 5: An example of the rendering result produced by our system using different N values. (a) shows the captured images with $N = 1, 6, 10$ million. The top images are the rendered frame from the main camera. The bottom images are rendered based on the setting of the reference camera, which aims at the instances that are far away from the main camera. (b) shows the entire crowd including the view frustums of the two cameras. The yellow dots in (b) represent the instances outside the view frustum of the main camera.

crowd contains 30,000 instances spread out in the virtual scene with predefined movements and moving trajectories. Figure 5 shows a rendered frame of the crowd with the value of N set to 1, 6, and 10 million, respectively. The main camera moves on the walkthrough path. The reference camera is aimed at the instances far away from the main camera and shows a close-up view of those far-away instances. Our LOD-based instancing method ensures the total number of selected vertices and triangles is within the specified memory budget, while preserving the fine details of instances that are closer to the main camera. Although the far-away instances are simplified significantly, because the long distances to the main camera, their appearance in the main camera do not cause a loss of visual fidelity. Figure 5(a) shows visual appearance of the crowd rendered from the viewpoint of the main camera (top images), in which far-away instances are rendered using the simplified versions (bottom images).

If all instances are rendered at the level of full detail, the total number of triangles would be 169.07 million. Through the simulation of the walkthrough camera path, we had an average of 15,771 instances inside the view frustum. The maximum and minimum numbers of instances inside the view frustum are 29,967 and 5,038, respectively. We specified different values for N. Table 2 shows the performance breakdowns with regard to the runtime processing components in our system. In the table, the "# of Rendered Triangles" column includes the minimum, maximum, and averaged number of triangles selected during the runtime. As we can see, the higher the value of N is, the more the triangles are selected to generate simplified instances and subsequently the better visual quality is obtained for the crowd. Our approach is memory efficient. Even when N is set to a large value, such as 20 million, the crowd needs only 26.23 million triangles in average, which is only 15.51% of the original number of triangles. When the value of N is small, the difference between the averaged and the maximum number of triangles is significant. For example, when N is equal to 5 million, the difference is at a ratio (*average*/*maximum*) of 73.96%. This indicates that the number of triangles in the crowd varies significantly according to the change of instance-camera relationships (including instances' distance to the camera and their visibility). This is because a small

Table 2: Performance breakdowns for the system with a precreated camera path with total 30,000 instances. The FPS value and the component execution times are averaged over 1000 frames. The total number of triangles (before the use of LOD) is 169.07 million.

N	FPS	# of Rendered Triangles (million)			Component Execution Times (millisecond)		
		Min.	Max.	Avg.	View-Frustum Culling + LOD Selection	LOD Mesh Generation	Animating Meshes + Rendering
1	47.91	1.88	9.70	5.20	0.68	2.87	26.06
5	43.30	6.12	10.14	7.50	0.65	3.87	26.44
10	32.82	11.64	13.78	13.04	0.65	6.27	29.40
15	23.00	17.88	20.73	19.54	0.71	9.45	36.43
20	18.71	23.13	27.73	26.23	0.68	12.52	42.85

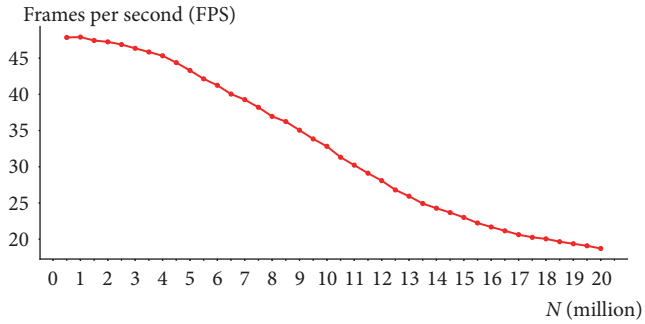

Figure 6: The change of FPS over different values of N by using our approach. The FPS is averaged over the total of 1,000 frames.

value of N limits the level of details that an instance can reach up. Even if an instance is close to the camera, it may not obtain a sufficient cut from N to satisfy a desired detail level. As we can see in the table, when the value of N becomes larger than 10 million, the ratio is increased to 94%. The View-Frustum Culling and LOD Selection components are implemented together, and both are executed in parallel at an instance level. Thus, the execution time of this component does not change as the value of N increases. The component of LOD Mesh Generation is executed in parallel at a triangle level. Its execution time increases as the value of N increases. Animating Meshes and Rendering components are executed with the acceleration of OpenGL's buffer objects and shader programs. They are time-consuming, and their execution time increases as more triangles need to be rendered. Figure 6 shows the change of FPS over different values of N. As we can see in the figure, the FPS decreases as the value of N increases. When N is smaller than 4 million, the decreasing slope of FPS is small. This is because the change on the number of triangles over frames of the camera path is small. When N is small, many close-up instances end down to the lowest level of details due to the insufficient memory budget from N. When N increases from 4 to 17 million, the decreasing slop of FPS becomes larger. This is because the number of triangles over frames of the camera path varies considerably with different values of N. As N increases beyond 17 million, the decreasing slope becomes smaller again, as many instances including far-away ones reach the full level of details.

6.2. Comparison and Discussion. We analyzed two rendering techniques and compared them against our approach in terms of performance and visual quality. The pseudo-instancing technique minimizes the amount of data duplication by sharing vertices and triangles among all instances, but it does not support LOD on a per-instance level [44, 52]. The point-based technique renders a complex geometry by using a dense of sampled points in order to reduce the computational cost in rendering [53, 54]. The pseudo-instancing technique does not support View-Frustum Culling. For the comparison reason, in our approach, we ensured all instances to be inside the view frustum of the camera by setting a fixed position of the camera and setting fixed positions for all instances, so that all instances are processed and rendered by our approach. The complexity of each instance rendered by the point-based technique is selected based on its distance to the camera which is similar to our LOD method. When an instance is near the camera, the original mesh is used for rendering; when the instance is located far away from the camera, a set of points are approximated as sample points to represent the instance. In this comparison, the pseudo-instancing technique always renders original meshes of instances. We chose two different N values (N= 5 million and N= 10 million) for rendering with our approach. As shown in Figure 7, our approach results in better performance than the pseudo-instancing technique. This is because the number of triangles rendered by using the pseudo-instancing technique is much larger than the number of triangles determined by the LOD Selection component of our approach. The performance of our approach becomes better than the point-based technique as the number of N increases. Figure 8 shows the comparison of visual quality among our approach, pseudo-instancing technique, and point-based technique.

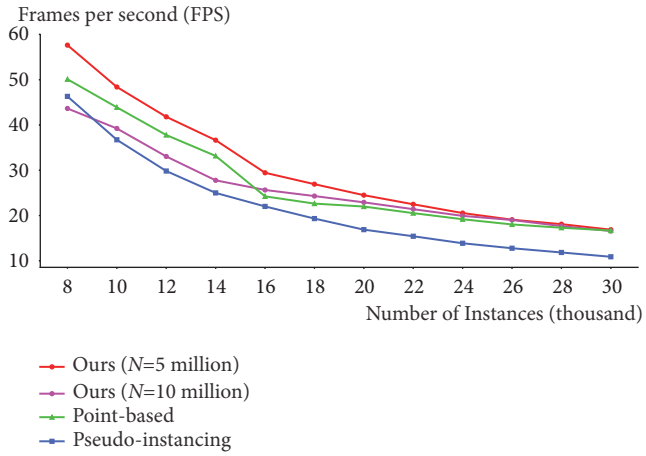

FIGURE 7: The performance comparison of our approach, pseudo-instancing technique, and point-based technique over different numbers of instances. Two values of N are chosen for our approach (N = 5 million and N = 10 million). The FPS is averaged over the total of 1,000 frames.

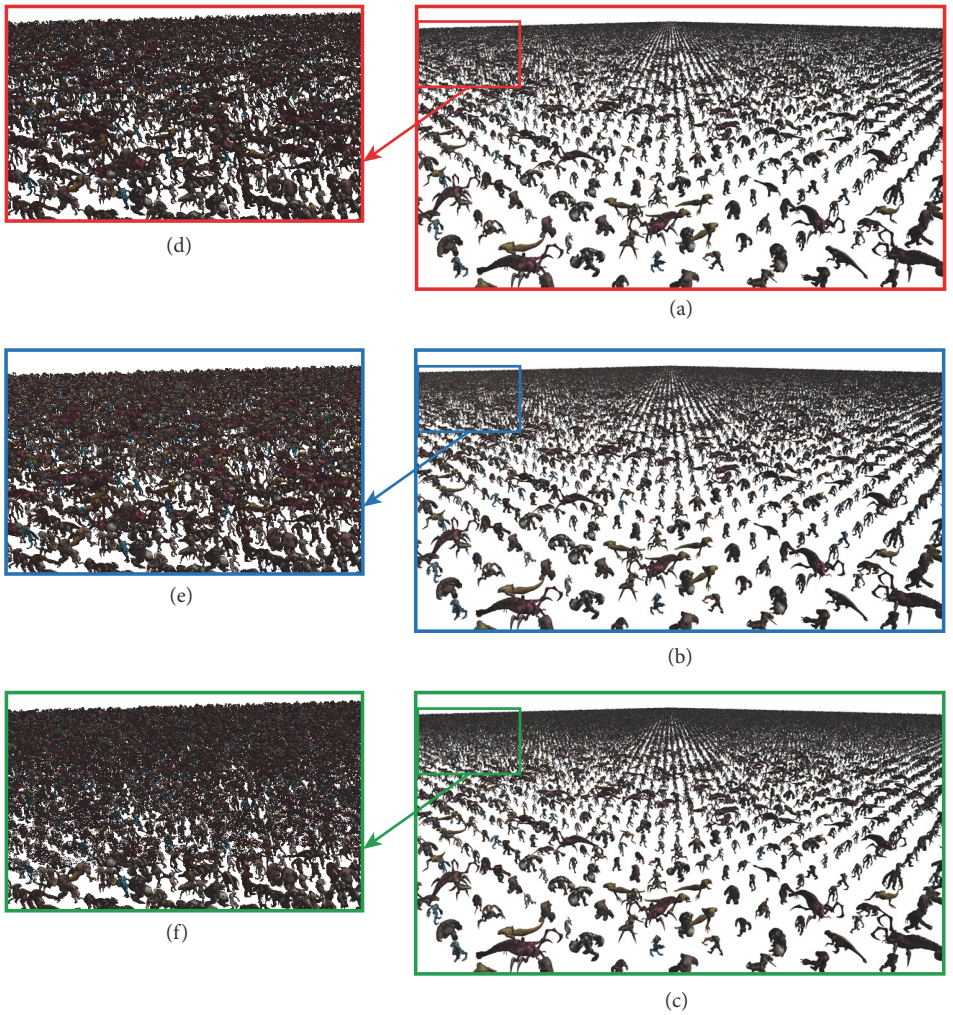

FIGURE 8: The visual quality comparison of our approach, pseudo-instancing technique, and point-based technique. The number of rendered instances on the screen is 20000. (a) shows the captured image of our approach rendering result with N = 5 million. (b) shows the captured image of pseudo-instancing technique rendering result. (c) shows the captured image of the point-based technique rendering result. (d), (e), and (f) are the captured images zoomed in on the area of instances which are far away from the camera.

The image generated from the pseudo-instancing technique represents the original quality. Our approach can achieve better visual quality than the point-based technique. As we can see in the top images of the figure, the instances far away from the camera rendered by the point-based technique appear to have "holes" due to the disconnection between vertices. In addition, the popping artifact appears when using the point-based technique. This is because the technique uses a limited number of detail levels from the technique of discrete LOD. Our approach avoids the popping artifact since continuous LOD representations of the instances are applied during the rendering.

7. Conclusion

In this work, we presented a crowd rendering system that takes the advantages of GPU power for both general-purpose and graphics computations. We rebuilt the meshes of source characters based on the flatten pieces of texture UV sets. We organized the source characters and instances into buffer objects and textures on the GPU. Our approach is integrated seamlessly with continuous LOD and View-Frustum Culling techniques. Our system maintains the visual appearance by assigning each instance an appropriate level of details. We evaluated our approach with a crowd composed of 30,000 instances and achieved real-time performance. In comparison with existing crowd rendering techniques, our approach better utilizes GPU memory and reaches a higher rendering frame rate.

In the future, we would like to integrate our approach with occlusion culling techniques to further reduce the number of vertices and triangles during the runtime and improve the visual quality. We also plan to integrate our crowd rendering system with a simulation in a real game application. Currently, we only used a single walk animation in the crowd rendering system. In a real game application, more animation types should be added, and a motion graph should be created in order to make animations transit smoothly from one to another. We also would like to explore possibilities to transplant our approach onto a multi-GPU platform, where a more complex crowd could be rendered in real time, with the supports of higher memory capability and more computational power provided by multiple GPUs.

Data Availability

The source character assets used in our experiment were purchased from cgtrader.com. The source code including GPU and shader programs were developed in our research lab by the authors of this paper. The source code has been archived in our lab and available for distribution upon requests. The supplementary video (available here) submitted together with the manuscript shows our experimental result of real-time crowd rendering.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by the National Science Foundation Grant CNS-1464323. We thank Nvidia for donating the GPU device that has been used in this work to run our approach and produce experimental results. The source characters were purchased from cgtrader.com with the buyer's license that allows us to disseminate the rendered and moving images of those characters through the software prototypes developed in this work.

Supplementary Materials

We submitted a demo video of our crowd rendering system as a supplementary material. The video consists of two parts. The first part is a recorded video clip showing the real-time rendering result as the camera moves along the predefined walkthrough path in the virtual scene. The bottom view corresponds the user's view, and the top view is a reference view showing the changes of the view frustum of the user's view (red lines). In the reference view, the yellow dots represent the character instances outside the view frustum of the user's view. At the top-left corner of the screen, the runtime performance and data information are plotted. The runtime performance is measured by the frames per second (FPS). The N value is set to 10 million, the total number of instances is set to 30,000, and the total number of original triangles of all instances is 169.07 million. The number of rendered triangles changes dynamically according to the user's view changes, since the continuous LOD algorithm is used in our approach. The second part of the video explains the visual effect of LOD algorithm. It uses a straight-line moving path that allows the main camera moves from one corner of the crowd to another corner at a constant speed. The video is played 4 times faster than the original FPS of the video clip. The top-right view shows the entire scene of the crowd. The red lines represent the view frustum of the main camera, which is moving along the straight-line path. The blue lines represent the view frustum of the reference camera, which is fixed and aimed at the far-away instances from the main camera. As we can see in the view of reference camera (top-left corner), the simplified far-away instances are gaining more details as the main camera moves towards the location of the reference camera. At the same time, more instances are outside the view frustum of the main camera, represented as yellow dots. *(Supplementary Materials)*

References

[1] S. Lemercier, A. Jelic, R. Kulpa et al., "Realistic following behaviors for crowd simulation," *Computer Graphics Forum*, vol. 31, no. 2, Part 2, pp. 489–498, 2012.

[2] A. Golas, R. Narain, S. Curtis, and M. C. Lin, "Hybrid long-range collision avoidancefor crowd simulation," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 7, pp. 1022–1034, 2014.

[3] G. Payet, "Developing a massive real-time crowd simulation framework on the gpu," 2016.

[4] I. Karamouzas, N. Sohre, R. Narain, and S. J. Guy, "Implicit crowds: Optimization integrator for robust crowd simulation," *ACM Transactions on Graphics*, vol. 36, no. 4, pp. 1–13, 2017.

[5] J. R. Shewchuk, "Delaunay refinement algorithms for triangular mesh generation," *Computational Geometry. Theory and Applications*, vol. 22, no. 1-3, pp. 21–74, 2002.

[6] J. Pettré, P. D. H. Ciechomski, J. Maim, B. Yersin, J.-P. Laumond, and D. Thalmann, "Real-time navigating crowds: scalable simulation and rendering," *Computer Animation and Virtual Worlds*, vol. 17, no. 3-4, pp. 445–455, 2006.

[7] J. Maïm, B. Yersin, J. Pettré, and D. Thalmann, "YaQ: An architecture for real-time navigation and rendering of varied crowds," *IEEE Computer Graphics and Applications*, vol. 29, no. 4, pp. 44–53, 2009.

[8] C. Peng, S. I. Park, Y. Cao, and J. Tian, "A real-time system for crowd rendering: parallel LOD and texture-preserving approach on GPU," in *Proceedings of the International Conference on Motion in Games*, vol. 7060 of *Lecture Notes in Computer Science*, pp. 27–38, Springer, 2011.

[9] A. Beacco, N. Pelechano, and C. Andújar, "A survey of real-time crowd rendering," *Computer Graphics Forum*, vol. 35, no. 8, pp. 32–50, 2016.

[10] R. McDonnell, M. Larkin, S. Dobbyn, S. Collins, and C. O'Sullivan, "Clone attack! Perception of crowd variety," *ACM Transactions on Graphics*, vol. 27, no. 3, p. 1, 2008.

[11] Y. P. Du Sel, N. Chaverou, and M. Rouillé, "Motion retargeting for crowd simulation," in *Proceedings of the 2015 Symposium on Digital Production, DigiPro '15*, pp. 9–14, ACM, New York, NY, USA, August 2015.

[12] F. Multon, L. France, M.-P. Cani-Gascuel, and G. Debunne, "Computer animation of human walking: a survey," *Journal of Visualization and Computer Animation*, vol. 10, no. 1, pp. 39–54, 1999.

[13] S. Guo, R. Southern, J. Chang, D. Greer, and J. J. Zhang, "Adaptive motion synthesis for virtual characters: a survey," *The Visual Computer*, vol. 31, no. 5, pp. 497–512, 2015.

[14] E. Millán and I. Rudomn, "Impostors, pseudo-instancing and image maps for gpu crowd rendering," *The International Journal of Virtual Reality*, vol. 6, no. 1, pp. 35–44, 2007.

[15] H. Nguyen, "Chapter 2: Animated crowd rendering," in *Gpu Gems 3*, Addison-Wesley Professional, 2007.

[16] H. Park and J. Han, "Fast rendering of large crowds using GPU," in *Entertainment Computing - ICEC 2008*, S. M. Stevens and S. J. Saldamarco, Eds., vol. 5309 of *Lecture Notes in Computer Science*, pp. 197–202, Springer, Berlin, Germany, 2009.

[17] K.-L. Ma, "In situ visualization at extreme scale: challenges and opportunities," *IEEE Computer Graphics and Applications*, vol. 29, no. 6, pp. 14–19, 2009.

[18] T. Sasabe, S. Tsushima, and S. Hirai, "In-situ visualization of liquid water in an operating PEMFC by soft X-ray radiography," *International Journal of Hydrogen Energy*, vol. 35, no. 20, pp. 11119–11128, 2010, Hyceltec 2009 Conference.

[19] H. Yu, C. Wang, R. W. Grout, J. H. Chen, and K.-L. Ma, "In situ visualization for large-scale combustion simulations," *IEEE Computer Graphics and Applications*, vol. 30, no. 3, pp. 45–57, 2010.

[20] B. Hernandez, H. Perez, I. Rudomin, S. Ruiz, O. DeGyves, and L. Toledo, "Simulating and visualizing real-time crowds on GPU clusters," *Computación y Sistemas*, vol. 18, no. 4, pp. 651–664, 2015.

[21] H. Perez, I. Rudomin, E. A. Ayguade, B. A. Hernandez, J. A. Espinosa-Oviedo, and G. Vargas-Solar, "Crowd simulation and visualization," in *Proceedings of the 4th BSC Severo Ochoa Doctoral Symposium*, Poster, May 2017.

[22] A. Treuille, S. Cooper, and Z. Popović, "Continuum crowds," *ACM Transactions on Graphics*, vol. 25, no. 3, pp. 1160–1168, 2006.

[23] R. Narain, A. Golas, S. Curtis, and M. C. Lin, "Aggregate dynamics for dense crowd simulation," in *ACM SIGGRAPH Asia 2009 Papers, SIGGRAPH Asia '09*, p. 1, Yokohama, Japan, December 2009.

[24] X. Jin, J. Xu, C. C. L. Wang, S. Huang, and J. Zhang, "Interactive control of large-crowd navigation in virtual environments using vector fields," *IEEE Computer Graphics and Applications*, vol. 28, no. 6, pp. 37–46, 2008.

[25] S. Patil, J. Van Den Berg, S. Curtis, M. C. Lin, and D. Manocha, "Directing crowd simulations using navigation fields," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 2, pp. 244–254, 2011.

[26] E. Ju, M. G. Choi, M. Park, J. Lee, K. H. Lee, and S. Takahashi, "Morphable crowds," *ACM Transactions on Graphics*, vol. 29, no. 6, pp. 1–140, 2010.

[27] S. I. Park, C. Peng, F. Quek, and Y. Cao, "A crowd modeling framework for socially plausible animation behaviors," in *Motion in Games*, M. Kallmann and K. Bekris, Eds., vol. 7660 of *Lecture Notes in Computer Science*, pp. 146–157, Springer, Berlin, Germany, 2012.

[28] F. D. McKenzie, M. D. Petty, P. A. Kruszewski et al., "Integrating crowd-behavior modeling into military simulation using game technology," *Simulation Gaming*, vol. 39, no. 1, Article ID 1046878107308092, pp. 10–38, 2008.

[29] S. Zhou, D. Chen, W. Cai et al., "Crowd modeling and simulation technologies," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 20, no. 4, pp. 1–35, 2010.

[30] Y. Zhang, B. Yin, D. Kong, and Y. Kuang, "Interactive crowd simulation on GPU," *Journal of Information and Computational Science*, vol. 5, no. 5, pp. 2341–2348, 2008.

[31] A. Malinowski, P. Czarnul, K. Czuryło, M. Maciejewski, and P. Skowron, "Multi-agent large-scale parallel crowd simulation," *Procedia Computer Science*, vol. 108, pp. 917–926, 2017, International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland.

[32] I. Cleju and D. Saupe, "Evaluation of supra-threshold perceptual metrics for 3D models," in *Proceedings of the 3rd Symposium on Applied Perception in Graphics and Visualization, APGV '06*, pp. 41–44, ACM, New York, NY, USA, July 2006.

[33] S. Dobbyn, J. Hamill, K. O'Conor, and C. O'Sullivan, "Geopostors: A real-time geometry/impostor crowd rendering system," in *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games, I3D '05*, pp. 95–102, ACM, New York, NY, USA, April 2005.

[34] B. Ulicny, P. d. Ciechomski, and D. Thalmann, "Crowdbrush: interactive authoring of real-time crowd scenes," in *Proceedings of the Symposium on Computer Animation*, R. Boulic and D. K. Pai, Eds., p. 243, The Eurographics Association, Grenoble, France, August 2004.

[35] H. Hoppe, "Efficient implementation of progressive meshes," *Computers Graphics*, vol. 22, no. 1, pp. 27–36, 1998.

[36] M. Garland and P. S. Heckbert, "Surface simplification using quadric error metrics," in *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques,*

SIGGRAPH '97, pp. 209–216, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1997.

[37] E. Landreneau and S. Schaefer, "Simplification of articulated meshes," *Computer Graphics Forum*, vol. 28, no. 2, pp. 347–353, 2009.

[38] A. Willmott, "Rapid simplification of multi-attribute meshes," in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics, HPG '11*, p. 151, ACM, New York, NY, USA, August 2011.

[39] W.-W. Feng, B.-U. Kim, Y. Yu, L. Peng, and J. Hart, "Feature-preserving triangular geometry images for level-of-detail representation of static and skinned meshes," *ACM Transactions on Graphics (TOG)*, vol. 29, no. 2, article no. 11, 2010.

[40] D. P. Savoy, M. C. Cabral, and M. K. Zuffo, "Crowd simulation rendering for web," in *Proceedings of the 20th International Conference on 3D Web Technology, Web3D '15*, pp. 159-160, ACM, New York, NY, USA, June 2015.

[41] F. Tecchia, C. Loscos, and Y. Chrysanthou, "Real time rendering of populated urban environments," Tech. Rep., ACM SIGGRAPH Technical Sketch, 2001.

[42] J. Barczak, N. Tatarchuk, and C. Oat, "GPU-based scene management for rendering large crowds," *ACM SIGGRAPH Asia Sketches*, vol. 2, no. 2, 2008.

[43] B. Hernández and I. Rudomin, "A rendering pipeline for real-time crowds," *GPU Pro*, vol. 2, pp. 369–383, 2016.

[44] J. Zelsnack, "GLSL pseudo-instancing," Tech. Rep., 2004.

[45] F. Carucci, "Inside geometry instancing," *GPU Gems*, vol. 2, pp. 47–67, 2005.

[46] G. Ashraf and J. Zhou, "Hardware accelerated skin deformation for animated crowds," in *Proceedings of the 13th International Conference on Multimedia Modeling - Volume Part II, MMM07*, pp. 226–237, Springer-Verlag, Berlin, Germany, 2006.

[47] F. Klein, T. Spieldenner, K. Sons, and P. Slusallek, "Configurable instances of 3D models for declarative 3D in the web," in *Proceedings of the Nineteenth International ACM Conference*, pp. 71–79, ACM, New York, NY, USA, August 2014.

[48] C. Peng and Y. Cao, "Parallel LOD for CAD model rendering with effective GPU memory usage," *Computer-Aided Design and Applications*, vol. 13, no. 2, pp. 173–183, 2016.

[49] T. A. Funkhouser and C. H. Séquin, "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, SIGGRAPH '93*, pp. 247–254, ACM, New York, NY, USA, 1993.

[50] C. Peng and Y. Cao, "A GPU-based approach for massive model rendering with frame-to-frame coherence," *Computer Graphics Forum*, vol. 31, no. 2, pp. 393–402, 2012.

[51] N. Bell and J. Hoberock, "Thrust: a productivity-oriented library for CUDA," in *GPU Computing Gems Jade Edition*, Applications of GPU Computing Series, pp. 359–371, Morgan Kaufmann, Boston, Mass, USA, 2012.

[52] E. Millán and I. Rudomn, "Impostors, pseudo-instancing and image maps for gpu crowd rendering," *Iranian Journal of Veterinary Research*, vol. 6, no. 1, pp. 35–44, 2007.

[53] L. Toledo, O. De Gyves, and I. Rudomín, "Hierarchical level of detail for varied animated crowds," *The Visual Computer*, vol. 30, no. 6-8, pp. 949–961, 2014.

[54] L. Lyu, J. Zhang, and M. Fan, "An efficiency control method based on SFSM for massive crowd rendering," *Advances in Multimedia*, vol. 2018, pp. 1–10, 2018.