GPU-Based Scene Management for Rendering Large Crowds

Joshua Barczak
AMD, Inc.

Natalya Tatarchuk
AMD, Inc.

Christopher Oat
AMD, Inc.

1 Introduction

Many rendering scenarios, such as battle scenes or urban environments, require rendering of large numbers of autonomous characters. Crowd rendering in large environments presents a number of challenges, including visibility culling, animation, and level of detail (LOD) management. These have traditionally been CPU-based tasks, trading some extra CPU work for a larger reduction in the GPU load, but the per-character cost can be a serious bottleneck. Furthermore, CPU-side scene management is difficult if objects are simulated and animated on the GPU. We present a practical solution that allows rendering of large crowds of characters, from a variety of viewpoints, with stable performance and excellent visual quality. Our system uses DirectX10® functionality to perform view-frustum culling, occlusion culling, and LOD selection entirely on the GPU, allowing thousands of GPU-simulated characters to be rendered with full shadows in arbitrary environments. To our knowledge this is the first system presented that supports this functionality.

2 Scene Management

We start with a vertex buffer containing all of the per-instance data needed to render each character, such as character positions and orientations. In our case, this information is obtained from a GPU-based crowd simulation, but a CPU-based simulation or user input could also be used. A key idea behind our system is the use of geometry shaders that act as *filters* for a set of character instances. Given the instance data, we want to generate a set of buffers containing the visible instances at each level of detail. We do this by repeatedly rendering the instances as point primitives, using a geometry shader to filter the stream by re-emitting points that pass a particular test. Multiple filtering passes can be chained together by using successive *DrawAuto* calls, and different tests can be set up simply by using different shaders.

2.1 Instance Culling

It is straightforward to implement view-frustum culling using a stream filtering pass, as described above. In a frustum culling pass, the vertex shader performs a frustum intersection test against the character bounding sphere, and the geometry shader re-emits characters that pass the test.

A unique benefit of our method is that we can also perform occlusion culling by examining the depth buffer in a vertex shader, and comparing it to the minimum depth of the character's bounding volume. This allows culling against arbitrary occluders in dynamic environments, without preprocessing, and without the use of queries or predication, which are too expensive to apply per-instance. Prior to culling, we build a Hierarchical Z (Hi-Z) image [Greene et al. 1993] using the information contained in the Z buffer. At cull-time, each object chooses a MIP level in the Hi-Z image based on the projected size of its bounding volume, and fetches a fixed number of texels for the occlusion test.

2.2 LOD Selection

After culling, it is still necessary to group visible instances by LOD. In general the number of LODs can be selected due to the size or the complexity of the environment and the number of characters rendered. In our system, we use a static, three-level LOD scheme, in which the distance to the object centroid determines LOD. LOD sorting is performed by using three successive filtering passes into three output buffers, where each pass emits only the instances that fall into a particular LOD bucket. Note that the culling tests are performed only once per instance, and the results are re-used during the LOD passes.

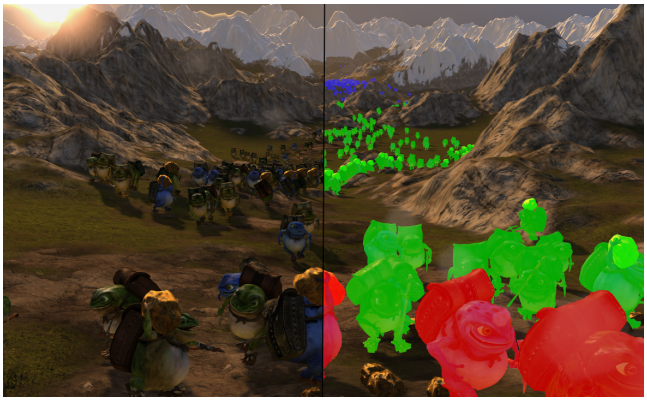

Figure 1: *A crowd of GPU-simulated characters (left). Characters are color coded by LOD (right).*

3 Rendering

After the LOD filtering passes, a GPU query is used to read the number of instances for each LOD, and separate *DrawInstanced* calls are issued to render the LOD groups. The readback of the instance count is necessary in order to pass the count to the *DrawInstanced* calls, but the GPU stall it introduces can be avoided if additional rendering work is submitted prior to the readback. We use hardware tessellation and displacement mapping for the closest LOD for high amount of details in close-ups, conventional rendering for the middle LOD, and simplified geometry and shaders for the furthest LOD rendering. To animate our characters, we sample the skeletal animations for each animation cycle (running, digging, etc), and pack the resulting curves into a texture array which is sampled by the characters' vertex shaders. This allows us full animation control for each character directly on the GPU.

4 Shadows

High quality rendering system requires dynamic shadows cast by characters onto the environment and themselves. To manage shadow map resolution, our system implements Parallel Split Shadow Maps [Zhang et al. 2006]. The view-frustum test described in Section 2.1 is used to ensure that only characters that are within a particular parallel split frustum are rendered. Occlusion culling could also be used for shadow maps as well, but we do not do this in our system, because only characters and smaller scene elements are rendered into the shadow maps and there is little to cull the characters against (shadows cast by terrain are handled separately). We use aggressive filtering for generation of soft shadows. This allows us to use further mesh simplification for the LOD rendered into shadow maps. For characters in the higher-detail shadow frusta, we use the same simplified geometry that is used for the *most distant* level of detail during normal rendering. For more distant shadows, we can use a more extreme simplification.

References

GREENE, N., KASS, M., AND MILLER, G. 1993. Hierarchical z-buffer visibility. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 231–238.

ZHANG, F., SUN, H., XU, L., AND LUN, L. K. 2006. Parallel-split shadow maps for large-scale virtual environments. In *VRCIA '06: Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications*, ACM, New York, NY, USA, 311–318.