GPU Tessellation for Detailed, Animated Crowds

Natalya Tatarchuk
AMD, Inc.

Joshua Barczak
AMD, Inc.

Budirijanto Purnomo
AMD, Inc.

Figure 1: *Our system allows rendering large crowds of characters (left) with extreme details in close-up (center) when using tessellation. For comparison, the same character without using tessellation (right) would look significantly less detailed*

1 Introduction

In games we frequently see large gatherings of crowds of characters; for example, an army or a rowdy crowd in a stadium. These crowds may seem realistic from a far away view, however, up close this impression fails as the viewer approaches the characters and the illusion of detailed creatures rapidly breaks down. We describe a method for managing level of detail and maintaining high quality rendering of character close-ups in massive crowd scenarios using GPU tessellation, instancing and stream out features of DirectX10®. We present a technique to increase the individual character's animation quality and overall system performance with a two-pass animation approach. We reduce memory bandwidth via shader-based vertex compression and decompression. Lastly, we developed an algorithm minimizing discontinuities across *uv* seams for displacement mapping.

2 GPU Tessellation

We designed an API for a GPU tessellation pipeline taking advantage of hardware fixed-function tessellator unit available on recent consumer GPUs. The tessellator unit generates *uv*s and topology connectivity for subdivided input primitives amplifying the original data up to 411 times. The generated vertex data is directly consumed by the vertex shader invoked for each new vertex. The super-primitive vertex IDs and barycentric coordinates are used to evaluate the new surface position. The amount of amplification can be controlled either by a per-draw call tessellation level or by dynamically computing tessellation factors per-primitive edge for the input mesh. Combining tessellation with instancing allows us to render diverse crowds of characters with minimal memory footprint. We can use this method along with stream out buffers and geometry shaders for level of detail management and texture arrays to create visually interesting and varied crowd of characters.

3 Level of Detail Control

For our level of detail management system we combine rendering several levels of details, using highly tessellated characters for extreme close-ups. In order to achieve maximum details with stable and predictable performance regardless of the number of characters, we control the tessellation levels as a function of the crowd density rendered in the current viewpoint. This is effective in avoiding a polycount explosion and retains the performance benefits of geometry instancing. Other metrics can be used for controlling polygonal density, such as using silhouette details or screen-space area preservation metrics.

We only need to store low resolution character model (6K triangles), effectively reducing memory footprint and bandwidth during rendering. At the same time we are able to preserve complex details in close up rendering by using a displacement map post tessellation, rendering a 1.6M triangle character at highly interactive rates. We can perform complex per-vertex computations on the original mesh since we only need to store animation data for the control cage of the character. GPU tessellation allows us to provide the data to GPU at coarse resolution, while rendering with high levels of details.

4 Rendering Optimizations

In the first pass we perform input mesh animation and transformations; rendering the base mesh vertices as instanced sets of point primitives, skinning them, and streaming out the results. Since we are performing this computation on the coarse input mesh, we can afford higher quality skinning as well as significantly reduce geometry transform cost. In the next pass, we tessellate the already animated and transformed mesh, using the original input primitive vertices' *vertex ID* and *instance ID* to retrieve and interpolate the transformed vertices from the stream out buffer, and apply displacement mapping. We use a compression scheme to pack the transformed vertices into a compact 128-bit format, allowing the tessellation pass to load a full set of vertex data using a single vertex fetch. Although the compression scheme requires a number of ALU cycles for de-compression, the reduction in stream out and vertex memory bandwidth more than compensates, yielding over 30% speedup on balance.

4.1 Displacement Map Generation

One challenge when rendering characters with displacement maps that contain texture *uv* borders is the introduction of texture *uv* seams. Unless neighboring *uv* borders are laid out with the same orientations and lengths, displacing with these maps will introduce geometry cracks along the *uv* borders. To solve this problem, we post-process our displacement maps by correcting all the texture *uv* borders as follows. First, we identify the border triangle edges (*i.e.* edges that contain vertices with more than one set of texture coordinates). Then, for each border edge, we compute the texel locations for the vertices; fetch, average, and update the texels for matching vertices. As long as all these border vertices map to unique texel locations, we can ensure a crack-free displacement mapping using nearest neighbor texture filtering. To improve *uv* seams for linear texture filtering and/or hardware tessellation, for each border edge, we sample the edge with equidistant points. Then, for each sampled point, we fetch, average and update the texels for matching points. Because the points might not map one-to-one, we repeat the above process several times to enhance the result. At the end, we dilate the *uv* borders. This technique is attractive because it is fast, simple to implement and it generates good results. Compared to previously known techniques [Purnomo et al. 2004], we do not change the *uv* layout, thus, we do not have to re-sample the displacement maps.

References

PURNOMO, B., COHEN, J., AND KUMAR, S. 2004. Seamless texture atlases. In *Symposium of Geometry Processing*, 65–74.