Out-of-core GPU Ray Tracing of Complex Scenes

Kirill Garanzha (KIAM RAS)   Alexander Bely (CentiLeo)   Simon Premoze   Vladimir Galaktionov (KIAM RAS)

Abstract

Increased demand for global illumination, image based-lighting and simplified workflow have pushed raytracing into mainstream. Many rendering and simulation algorithms that were considered strictly offline are becoming more interactive on massively parallel GPUs. Unfortunately, the amount of available memory on modern GPUs is relatively small. Scenes for feature film rendering and visualization have large geometric complexity and can easily contain millions of polygons and a large number of texture maps and other data attributes. In this talk, we describe a general purpose out-of-core ray tracing engine for the GPU where we address data management, ray-intersection and shading. We utilize a GPU data cache that enables efficient access of out-of-core data. We develop a novel ray intersection algorithm built around acceleration structure that brings needed data on demand using page-swapping. We further reduce memory usage by using a simple geometry quantization. The ray tracing engine is used to implement a variety of rendering and light transport algorithms.

Data Management

Modern GPUs have enormous computational power (1 Tflops) and large memory bandwidth (150 Gb/s). Relatively small amount of memory and a lack of virtual memory system forced us to design a GPU data cache that increases the virtual GPU memory size by an order of magnitude. An application that works on a large dataset creates a GPU data manager and processes data using request and process data kernel() in the following loop:

```
void out_of_core_data_processing()
{
  available_new_data = 1
  while( available_new_data ) {
    // Process in-core data
    // Request out-of-core data
    request_and_process_data_kernel(data);
    // Bring missing data into GPU memory
    available_new_data = swap_requested_pages(gpu_data_manager);
  }
}
```

We first make a data request to the data manager. If some desired data blocks are out-of-core, we mark them as requested. The data manager brings requested data blocks into the GPU memory (swap_requested_pages()). On subsequent loop iterations, needed data is in the GPU memory and the kernel can do computation. This process of requesting, computing and transferring data is repeated until no more out-of-core data is requested by the application. The virtual data manager allows us to access large data sets and arbitrary arrays.

Out-of-Core Intersection

Efficient raytracing requires an acceleration data structure to compute ray intersections with the scene geometry. Since the scene does not necessarily fit into the GPU memory, we create a multi level Bounding Volume Hierarchy. We implement a very fast BVH builder on the GPU that produces high quality BVHs suitable for fast ray tracing of out-of-core geometry and fully animated scenes. We traverse this acceleration structure knowing that requested geometry may not be available in memory. The intersection algorithm successfully hides data transfer latency by performing intersections on in-core geometry. We also use a simple geometry quantization scheme to reduce geometry size and further reduce expensive memory transfer size (description is available in supplemental paper).

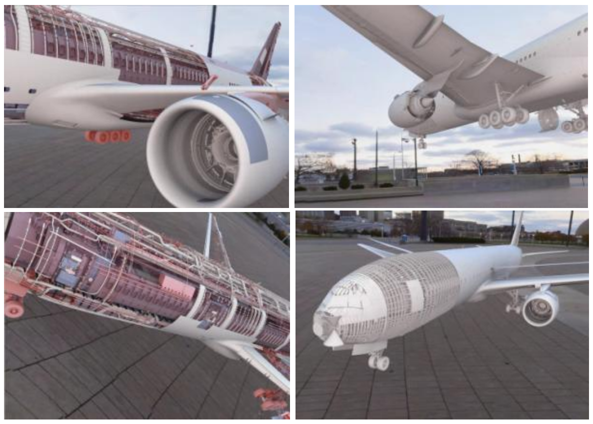

Figure 1: Frames from Boeing 777 animation sequence. It takes 15 seconds to render a fully converged final frame with global illumination on a single NVIDIA GTX 480 GPU. The final image accumulates 100 progressive path tracing refinements (i.e. 100 Monte Carlo samples / pixel are computed in 15 seconds).

Shading

We support programmable shaders: ray generation, materials and light transport. Sophisticated shading and global illumination computation may require an arbitrary amount of data. We use the GPU data cache to access large number of textures and geometry attributes (normals, texture coords, etc…). The shaders can generate arbitrary number of new rays to compute lighting and shadowing. We organize rays in a ray queue that helps with improving ray coherency, hides latency (overlap computation with data transfers) and provides simple ordering and synchronization mechanism. Shaders submit rays to a global ray queue from which rays are consumed by the out-of-core ray-intersector. We use a large ray queue (32M rays) to increase the rendering pipeline throughput. The implementation of texture on demand (Peachey) is almost completed and we can potentially support large textures which are stored on the disc (some data is available in supplemental document). The work in progress is implementing Metropolis Light Transport algorithm where large number of rays may be active simultaneously.

Summary

We have implemented a general purpose out-of-core ray tracer for rendering complex scenes on the GPU. Figure 1 shows several frames from a Boeing 777 animation sequence. The model has 360 million polygons. Using a single Nvidia GTX 480 graphics card, it takes 15 seconds to render a full converged final frame (1024x768) with global illumination (path tracing, three bounces, 100 progressive image iterations). Path tracing demonstrates a stress test for GPU Virtual memory manager because path tracing can be formulated as the number of sparse memory access to the scene geometry.