Single Pass Depth Peeling via CUDA Rasterizer

Fang Liu[‡*] Meng-Cheng Huang[‡] Xue-Hui Liu[‡] En-Hua Wu[‡§]
Institute of Software, Chinese Academy of Sciences[‡] University of Macau[§]

1 Introduction

Multi-fragment effects play important roles on many graphics applications, which require operations on more than one fragment per pixel. The classical depth peeling algorithm [Everitt 2001] peels off one layer each pass, but the performance degrades for large scenes. We prefer to capture multiple fragments in a single pass, which is difficult because the fragments generated in graphics pipeline are not allowed to be scattered to arbitrary positions of the buffers. Compute unified device architecture (CUDA) [NVIDIA 2008] provides more flexible control over the GPU memory, but accessing of the fragments generated by graphics pipeline is not yet supported. In this work we design a CUDA rasterizer so that many graphics applications can benefit from the free control of GPU memory, especially for the multi-fragment effects. We present two efficient schemes to capture and sort multiple fragments per pixel in a single geometry pass via the atomic operations of CUDA without read-modify-write (RMW) hazards. Experimental results show significant speedup to classical depth peeling, especially for large scenes.

2 CUDA Rasterizer for Depth Peeling

Our CUDA rasterizer is designed by packing the triangles into a texture and pass it to a CUDA kernel, with each thread projecting a single triangle onto the screen and rasterizing it by the scan-line algorithm. On each pixel location covered by the projected triangle, a fragment will be generated with interpolated attributes, such as depth, color, etc. We then describe two efficient schemes to capture and sort the fragments per pixel via the atomic operations of CUDA.

• **CUDA Depth Peeling 1.** A fixed size integer array is allocated per pixel as storage in global memory initialized by the maximum integer (0x7FFFFFFF). Fragments are captured and sorted per pixel on the fly using the *atomicMin* operation of CUDA. This operation performs an integer comparison between the source value in global memory and the destination value, keeps the minimum one and returns the source value. It guarantees operations on the same location will be serialized thus avoids RMW hazards. Since the normalized depth value is always positive, it can be directly cast to integer with consistent ordering. The pseudocode is showed as follows.

```
__global__ CUDADepthPeeling1(TriangleTexture, DepthArrays)
  ProjectTriangle(TriangleTexture, ThreadID);
  for each rasterized fragment on pixel location [x,y]
    newDepth ← InterpolateDepth();
    for i = 0 : DepthArraySize - 1
      oldDepth ← atomicMin(&DepthArrays[x][y][i], newDepth);
      if(oldDepth == MAX_INTEGER ) break; // 0x7FFFFFFF
      newDepth ← max(newDepth, oldDepth);
    end for
  end for
```

This strategy assures correct ascending order under concurrent updates without RMW hazards. For a certain pixel, suppose there are N threads generating N fragments in rasterization order. They will be concurrently executed with each one loops to store its depth value into the array on that pixel location (denoted as A for short). Each thread will early or late begin the first iteration of the loop and try to store its depth value into $A[0]$. The comparisons will be serialized and the first executed thread will store its value into $A[0]$ and exit the loop with the MAX_INTEGER. The others will sequentially compare their values with $A[0]$ and swap if their values are less than $A[0]$. After all the threads end the first iteration, the minimum fragment will survive in $A[0]$, while the rest $N-1$ threads will hold the rest $N-1$ values. They will later compete for $A[1]$ in the same way: the thread first update $A[1]$ will exit, and the second minimum depth value will stay in $A[1]$ finally, and so on. After all the threads have exited the loop, all the fragments will be stored into the array in ascending order for post-processing. In addition, it is capable to capture arbitrary number of front-most layers, thus is memory efficient. However, for applications involving multiple fragment attributes, since the 64-bit *atomicMin* operation is not yet available, we resort to a second scheme to solve the problem.

• **CUDA Depth Peeling 2.** Inspired by [Carpenter 1984], we allocate a struct (such as $float2$) array per pixel as a list and set a corresponding fragment counter initialized to 0 in global memory. The k^{th} incoming fragment of a certain pixel will increase the counter to $k+1$ and return the source value k using the *atomicInc* operation. Then the fragment attributes can be stored into the k^{th} entry of the array at any time without RMW hazards. In post-processing, the fragments per pixel will be loaded to registers and sorted by a bitonic sort in a following CUDA kernel for further applications.

3 Results

Model	Dragon	Buddha	Lucy	Statue	Bunny
Tri No.	871K	1.0M	2.0M	10.0M	70K
CUDP1	321fps	287fps	153fps	59fps	459fps
CUDP2	363fps	309fps	165fps	61fps	573fps
SRAB	116f/2g	106f/2g	49f/2g	8f/2g	334f/2g
DP	29f/13g	26f/13g	14f/13g	2f/15g	128f/13g

Table 1: *Comparison of frame rates(fps) and geometry passes(g).*

Table 1 shows the performance of our two schemes utilizing atomicMin (CUDP1) and atomicInc (CUDP2) operations in comparison with stencil routed A-buffer (SRAB) [Myers and Bavoil 2007] and the classical depth peeling (DP) for a 512x512 viewport on Geforce GTX 280. Both of the two schemes gain significant speedup to DP especially for large scenes because most memory access latency of atomic operations on global memory overlaps that of vertex fetching while rasterization, thus can be hidden by the thread scheduler.

References

CARPENTER, L. 1984. The a-buffer, an antialiased hidden surface method. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, 103–108.

EVERITT, C. 2001. Interactive order-independent transparency. Tech. rep., NVIDIA Corporation.

MYERS, K., AND BAVOIL, L. 2007. Stencil routed a-buffer. *ACM SIGGRAPH 2007 Technical Sketch Program.*

NVIDIA. 2008. Nvidia cuda: Compute unified device architecture.

*China Basic S&T 973 Research Grant(2009CB320802), National 863 High-Tec Grant(2008AA01Z301), NSFC(60573155)&UM Research Grant.

SIGGRAPH 2009, New Orleans, Louisiana, August 3–7, 2009.
ISBN 978-1-60558-726-4/09/0008