FreePipe: a Programmable Parallel Rendering Architecture for Efficient Multi-Fragment Effects

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

Abstract

In the past decade, modern GPUs have provided increasing programmability with vertex, geometry and fragment shaders. However, many classical problems have not been efficiently solved using the current graphics pipeline where some stages are still fixed functions on chip. In particular, multi-fragment effects, especially order-independent transparency, require programmability of the blending stage, that makes it difficult to be solved in a single geometry pass. In this paper we present FreePipe, a system for programmable parallel rendering that can run entirely on current graphics hardware and has performance comparable with the traditional graphics pipeline. Within this framework, two schemes for the efficient rendering of multi-fragment effects in a single geometry pass have been developed by exploiting CUDA atomic operations. Both schemes have achieved significant speedups compared to the state-of-the-art methods that are based on traditional graphics pipelines.

CR Categories: I.3.7 [Computing Methodologies]: Computer Graphics—Three-Dimensional Graphics and Realism

Keywords: depth peeling; compute unified device architecture (CUDA); graphics hardware; programmable graphics pipeline; rasterizer; multi-fragment effects; order-independent transparency; atomic operation.

1 Introduction

Hardware graphics are increasingly evolving away from fixed functionality to pipelines where some stages such as the vertex, pixel and geometry shaders are programmable. However, the rasterization and blending stages are still mostly fixed functions in modern GPUs. Because hardware rasterization unit processes triangles in submission order rather than depth order, many important problems that depend on the global information of the scene, such as order-independent transparency and occlusion culling, have not been efficiently solved. On the other hand, NVIDIA CUDA [NVIDIA 2008] is a general purpose parallel computing architecture built with a SIMT (single-instruction, multiple-thread) programming model. All concurrently executed threads are organized into grids of blocks that can cooperate amongst themselves through shared memory and synchronization. CUDA is a powerful tool for general purpose computing that provides more flexible control over the graphics hardware. However, CUDA's interaction with the graphics pipeline is limited to pixel and vertex buffer objects, and textures: direct access of the fragments generated by the traditional graphics pipeline is not yet supported.

I3D 2010, Washington, DC, February 19 – 21, 2010.
© 2010 ACM 978-1-60558-938-1/10/0002 $10.00

In this paper, we present FreePipe: a system for a fully programmable rendering architecture on current graphics hardware. Our system bridges between the traditional graphics pipeline and a general purpose computing architecture while retaining the advantages of both sides. The core of our system is a z-buffer based triangle rasterizer that is entirely implemented in CUDA. All stages of the pipeline can be mapped to CUDA programming model. Our system is fully programmable while still having comparable performance to the traditional graphics pipeline.

In our system, we propose two schemes as modifications to the traditional graphic pipeline for efficient rendering of multi-fragment effects. All layers of the scene can be captured and sorted in depth order within a single geometry pass by exploiting CUDA atomic operations. Both schemes have achieved significant speedup compared to the state-of-the-art depth peeling algorithm [Mammen 1989; Everitt 2001] that is based on the traditional graphics pipeline. We believe other problems such as occlusion culling can also benefit from this flexible architecture. The contributions of our paper include:

- A demonstration system that has a fully programmable graphics pipeline that can run on current graphics hardware. Many graphics applications can benefit from flexible control over all the stages of the pipeline.
- Two novel schemes have been proposed within this architecture for efficient rendering of multi-fragment effects in a single geometry pass using CUDA atomic operations. Both schemes have significant speed improvement over the stat-of-the-art depth peeling algorithm.

The rest of the paper is organized as follows. Section 2 summarizes the related works on graphics rendering pipeline and multi-fragment effects. Section 3 overviews the system and the details of the GPU implementation on current graphics hardware. Section 4 proposes two schemes for efficient rendering of multi-fragment effects in a single geometry pass. Experimental results and a general discussion on performance, memory consumption and limitations are presented in section 5. Finally, we propose future work in section 6 and conclude our paper in section 7.

2 Related Work

Graphics Rendering Pipeline. Over the past decades, many advanced rendering pipelines such as ray tracing, REYES [Cook et al. 1987] have been developed for photorealistic rendering. [Horn et al. 2007; Popov et al. 2007] use stackless kd-tree traversal and packet tracing on the GPU to achieve performance better than well designed CPU ray tracers for static scenes. [Zhou et al. 2008] accelerated ray tracing by building k-d trees on the GPU for dynamic scenes. [Tatarinov and Kharlamov 2009] proposes alternative rendering pipelines using CUDA for both ray tracing and REYES system. Renderants [Zhou et al. 2009] parallelizes each stage of the REYES system on the GPU to achieve interactive performance. [Fatahalian et al. 2009] also utilizes CUDA and proposes a data-parallel algorithm to rasterize micropolygons for efficient defocus and motion blur. Though these methods can generate high quality rendering results, the

performance is still not satisfactory for real time applications. As a result, graphics vendors prefer the rasterization-based techniques [Catmull 1974] for better performance. The earliest GPUs were built based on multi-core platform to perform massively parallel computations, such as vertex transformation, rasterization, pixel shading, and frame buffer blending. All of these stages were fixed function units on chips leaving only an assortment of options and parameters application-configurable. With the increasing demanding for general shading, the graphics vendors began to provide programmability at different stages of the graphics pipeline. The vertex, pixel and geometry shaders have gradually been open to programmers [Blythe 2006].

However, the rasterization and blending stages are still fixed functions on chip, leaving many classical problems, such as order-independent transparency, occlusion culling and anti-aliasing inefficiently solved. As a result, there are several theoretical modifications to the current graphics pipeline. [Aila et al. 2003] introduces a delay stream between the vertex and pixel processing units. The triangles are delayed in a cache on chip for references by the incoming ones. [Jon Hasselgren 2007] designs a programmable culling unit to perform quick culling on entire blocks of fragments. [Sugerman et al. 2009] proposes GRAMPS, a programming model for graphics pipelines that provides a general set of abstractions for building parallel applications with both task and data-level parallelism for future GPUs. Recently, Intel has announced Larrabee [Seiler et al. 2008], a highly parallel model that makes the rendering pipeline completely programmable. It is clear that modern graphics hardware are evolving towards full programmability.

Multi-fragment Effects. Multi-fragment effect requires operations on more than one fragment per pixel location. It plays important roles in many graphics applications, such as order-independent transparency, volume rendering, collision detection, refraction and caustics, global illumination, .etc. Efficient rendering of multi-fragment effects has not been well solved on current graphics pipeline because fragments have to be processed in depth order rather than submission order.

One catalogue of solutions to this problem works at the primitive level and pre-sorts geometries according to the viewing distances, then renders them from back to front. The most typical one of them is the painter's algorithm. [Wexler et al. 2005] discretizes the scenes into grids and sorts the grids in depth order. Then the sorted grids per pixel are divided into batches, where each batch will be handled by the classical depth peeling. Visort [Govindaraju et al. 2005a] sorts geometries on the GPU by exploiting temporal coherence between successive frames. Coherent layer peeling [Carr et al. 2008] exploits the same property and peels away sorted layers each iteration. In the worst case of reverse order, it peels away only two layers per iteration. For these methods, sorting of primitives will become the performance bottleneck, especially for large scenes.

Another catalogue of algorithms works at the fragment level performing fragment sorting on the GPU. Classical depth peeling algorithm [Mammen 1989; Everitt 2001] peels off the front-most layer each geometry pass using Z-buffer [Catmull 1974]. However, for large scenes with high complexity, multiple vertex transformations will dominate the computation thus becoming the performance bottleneck. A-buffer [Carpenter 1984] is a theoretical extension of Z-buffer that stores all fragments per pixel into an unbounded list in submission order. The list will be sorted according to depth values in post-processing. Other extensions include R-buffer [Wittenbrink 2001], Z^3 algorithm [Jouppi and Chang 1999], and F-Buffer [Mark and Proudfoot 2001]. K-buffer [Bavoil et al. 2007; Liu et al. 2006] captures multiple fragments in a single geometry pass by inserting each incoming fragment into an array per pixel maintained on multiple render targets (MRT). K-buffer gains great speedup to the classical depth peeling, but suffers from serious read-modify-write hazards. Though the problem can be alleviated by pre-sorting of the geometry, triangle batches or multi-pass approach, the performance of the algorithm will be consequently degraded. Stencil routed A-buffer [Myers and Bavoil 2007] sequentially routes fragments into sub-pixels of the multi-sampling anti-aliasing (MSAA) buffers using the stencil test. A fullscreen shader pass is then performed to sort the fragments using a bitonic sort before deferred shading. However, the theoretical speedup compared to depth peeling is limited to 8 times due to the maximum number of MSAA samples. Dual depth peeling [Bavoil and Myers 2008] manages to capture both of the nearest and furthest layers simultaneously each pass by utilizing the 32-bit floating-point MAX/MIN blending, but only gains two times speedup. Bucket depth peeling [Liu et al. 2009] exploits the same feature and captures up to 32 fragments in a single geometry pass using bucket sort on MRT. The potential fragment collisions can be alleviated by an additional pass or an adaptive scheme. The algorithm gains a great speedup compared to depth peeling, but its approximate nature and large memory overhead will limit the utility of the algorithm. Ideally we wish to perform correct fragment sorting in a single geometry pass, that is difficult under current graphics pipeline where the blending stage is fixed on chip.

3 System Overview

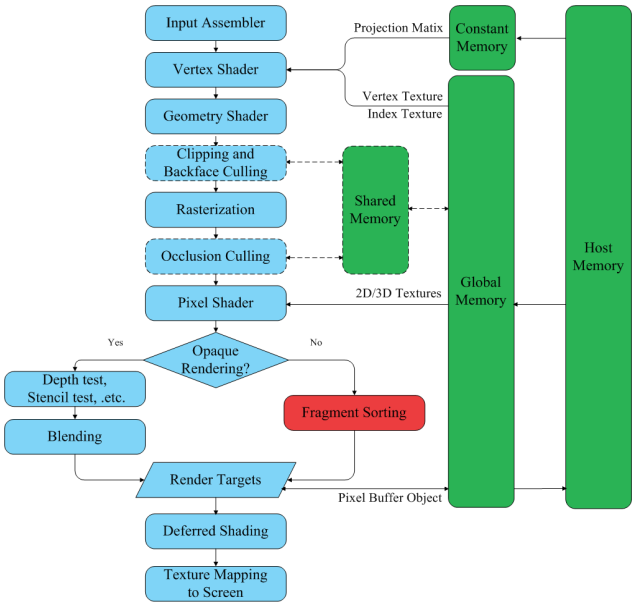


Figure 1: ***System overview of FreePipe.*** *The alternative stage is marked in red.*

Figure 1 overviews our system that is based on triangle rasterization with an OpenGL/D3D style [Segal and Akeley 2009]. The traditional graphics pipeline can be mapped to CUDA generating blocks of concurrently executed threads with each thread processing a single triangle independently.

Our system takes a scene geometry and camera parameters as input. The modelview-projection matrix is pre-computed based on scene configurations in each frame. We pack the attributes of each vertex and the vertex indices of each triangle into two textures for memory

coherency. The matrix must be accessed three times per thread so is to loaded into constant memory to reduce memory latency.

The vertices of the triangle are transformed from model space to screen space by multiplying the modelview-projection matrix in vertex shader. After projection, if back face culling is enabled, the triangle will be tested according to its normal and back faces will be discarded. Otherwise, the vertex order is reversed for consistency. The back-face culling test is performed with double-precision, or some faces might be culled or reversed by error due to the limited precision of the floating point arithmetic.

Frustum clipping is done by calculating the axis-aligned bounding volume of the triangle in screen space according to its three vertices. If the bounding volume is totally out of the view frustum, the triangle will be discarded and the thread will terminate. Otherwise, the bounding box of the triangle on the screen plane will be clipped according to the size of the screen.

We then perform rasterization using a scan-line algorithm. Pixel locations covered by the clipped triangle bounding box will be tested. Each pixel inside the triangle will generate a fragment with interpolated attributes. The pixel-level early z-culling is an optional operation to cull the fragments that fail the test before shading. Next, a customized fragment shader is executed for each fragment. The rendering result will be output to the corresponding pixel location in the render targets that reside in global memory.

Post-fragment tests such as depth test and stencil test can be estimated by atomic compare and bitwise operations. An atomic function performs a read-modify-write operation on a 32-bit or 64-bit word in global or shared memory. Each atomic operation on the same location will be performed in one transaction without interference from other threads, thus can avoid read-modify-write hazards. However, updating the depth buffer and color buffer together might introduce inconsistency since CUDA does not support 64 bit atomic compare operations or any critical area strategy. We will describe a heuristic method in Section 4.1.2 to partially address this problem.

Our system is flexible and offers full programmability in all stages, as shown by blue components in Figure 1. For multi-fragment effects, we modify the blending stage of the pipeline and develop two efficient schemes to capture and sort multiple fragments per pixel in a single geometry pass.

4 Single Pass Fragment Sorting

The fragment level methods for multi-fragment effects always require multiple rasterizations of the scene to ensure correct results, that will become the performance bottleneck, especially for large scenes. Ideally, we prefer to perform fragment sorting in a single geometry pass. Though there are several algorithms for efficient parallel sorting on the GPU [Govindaraju et al. 2005b; Sengupta et al. 2007; Cederman and Tsigas 2008; Sintorn and Assarsson 2008; Satish et al. 2009], their data sets are always pre-determined and free to be accessed many times when necessary. Threads have been carefully designed to access data at different memory locations simultaneously and synchronized to avoid read-modify-write hazards. On the contrary, the main challenge for fragment sorting on the GPU is that all the fragments are generated unpredictably. They have to be captured on the fly and cannot be reproduced unless at the cost of an additional rasterization. K-buffer and bucket depth peeling try to perform parallel insert sort and bucket sort on the GPU but suffers from read-modify-write hazards and fragment collisions due to the fixed blending stage on chip. As a result, it is difficult to implement fragment sorting with the traditional graphics pipeline in a single geometry pass.

In our system, we can modify the blending stage in graphics pipeline to make the problem tractable. Two efficient schemes are proposed to perform fragment sorting in a single geometry pass. The first scheme is called multi-depth test scheme, which captures and sorts multiple fragments per pixel on the fly via $atomicMin/atomicMax$ operations. The second scheme is called A-buffer scheme, which captures all fragments per pixel in submission order via *atomicInc* operation and post-sorts them before deferred shading. We will describe these two schemes in more details.

4.1 Multi-Depth Test Scheme

4.1.1 The Algorithm

This scheme performs fragment sorting on the GPU using a novel sorting algorithm for unpredictable data via CUDA *atomicMin* operation. The $atomicMin$ operation reads a 32-bit source value located at an address in global or shared memory, performs an integer comparison between the source and destination values, stores the minimum one back to memory at the same address, and returns the source value. Since *atomicMin* operation only works with signed and unsigned integers, the depth values should be directly cast to integers before comparison while maintaining consistent results.

We begin by allocating a fixed size integer array per pixel as storage in global memory. The size can be set to the maximum depth complexity in all view angles to avoid overflow, provided sufficient memory is available. Without loss of generality, we initialize each entry of the array to the maximum integer (0x7FFFFFFF) before rasterization. The standard memory initialization runtime API $cudaMemset$ is inefficient for such a large amount of memory. Instead we use a simple CUDA kernel with each thread initializing a single entry of the array. Through this way, the initializing operation can be performed entirely on the GPU without the involvement of the CPU and gains more throughput.

Each incoming fragment will loop through the pixel array at corresponding pixel locations and store the depth value to the first empty entry via *atomicMin* operation. For a non-empty entry, if its value is smaller than the current depth value, the entry will be left unchanged and the fragment will continue to test the next entry. Otherwise, *atomicMin* operation assures the current depth value to be stored into that entry and the returned source value will be used to test the next entry. This operation loops until the fragment is stored into the first empty entry. Otherwise, it will be discarded at the end of the array. The pseudo-code of the algorithm is shown in Table 1.

This strategy guarantees correct ascending depth order under concurrent updates of the array per pixel without read-modify-write hazards. For a certain pixel location, suppose there are N threads generating N fragments in rasterization order. They will concurrently execute 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 $A[0]$. These comparisons on the same location will be serialized and the first executed thread will always succeed to store its value into $A[0]$ and exit the loop with the maximum integer returned. 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 depth will be stored into $A[0]$, while the rest $N-1$ threads will hold the rest $N-1$ values. In the second round they will compete for $A[1]$ in the same way: the thread that first compares with $A[1]$ will store its value into A[1] and exit the loop with the maximum integer returned, and the second minimum fragment depth will be stored into $A[1]$, with the rest $N-2$ threads holding the rest $N-2$ values, and so on. After

all the threads have exited the loop, all the depth values of the fragments will be stored into the array in ascending order. They can also be sorted on the fly in descending order utilizing *atomicMax* operation in a similar way with the array initialized to 0. Then the sorted depth array per pixel can be passed to a following CUDA kernel for post-processing. The results can be rendered into a pixel buffer object and directly drawn onto the screen using texture mapping.

Giving an array of fixed size N, the multi-depth test scheme can correctly capture and sort the minimum N fragment depth values. In other words, it is capable of peeling off N front-most layers. This is useful for some applications when only the first N layers of the scene are needed, such as shadow mapping [Bavoil et al. 2008]. In contrast, [Myers and Bavoil 2007] has to capture all the layers before post-sorting, that might cause memory exhaustion especially for complex scenes, thus our scheme is more memory efficient. In practice, we can also expect the hardware vendors to extend the depth test on traditional graphics pipeline to multiple levels in the future according to this algorithm, since it has been proven doable on current hardware. We also expected some new extensions in OpenGL/D3D to bind multiple depth buffers to frame buffer objects and capture N front-most layers.

4.1.2 Capture Depth and Color Together

For applications that involve fragment color, such as order-independent transparency, since CUDA does not support 64-bit atomic operations, we cannot simply pack the color and the depth into a 64-bit word and sort together. Meanwhile, there is not any built-in critical area strategy in CUDA. So update the depth buffer and color buffer together can not be performed in a single atomic transaction, even for rendering opaque surfaces. Using an additional pass to capture the color could be a natural solution but the performance will be halved. Here we provide an alternative heuristic to partially address this problem in a single pass.

The fragment depth value is always normalized to a range of [0,1]. We first map the depth range to [0.5,1] with some loss of precision for fragments very close to the near plane. As a result, all the depth values will have the same signs (1bit) and exponents (8 bits), which can be easily verified according to the IEEE-754 standard for binary floating-point arithmetic. We then cast each depth value to an unsigned integer, shift it left by 9 bits and set the last 12 bits to zero. Now we lose 3 more bits precision of the depth values but have 12 free bits at tail while maintaining consistent depth order. For constant alpha values, we map the RGB channels of each fragment color to [0,255] and compose a 24-bits color attribute. The high and the low 12 bits of the color attribute can be separately catched to the 12 free bits of two copies of the depth values. The two composed values will update the color buffer and the depth buffer using two sequential atomic operations. Since the higher 20 bits of the two values are identical and consistent with the original depth order, the two buffers will always hold the color for the same fragment, except when two fragments are too close to each other. During post processing, the color attribute can be retained from the two buffers by reversed operations.

This strategy helps to assure the consistent update of the depth buffer and the color buffer, generating visually identical results to the ground truth when rendering opaque or transparent scenes. Due to the lost depth precision, it might cause inconsistency when two fragments are too close. Also the performance will be degraded by the doubled atomic operations in global memory. Modern graphics hardware has provided some built-in strategy to correctly update depth and color together, but it has not been exposed to CUDA yet. We believe the future version of CUDA will probably investigate the hardware and support such functions.

Algorithm Multi Depth Test Scheme

```
texture<float4, 1, cudaReadModeElementType> VertexTex;
texture<int4, 1, cudaReadModeElementType> IndexTex;
__constant__ int ScreenWidth,ScreenHeight,DepthArraySize;
__constant__ float MVP[16]; // Modelview-projection matrix

__global__ CUDADepthPeeling1(DepthArray)
 1:  [V1,V2,V3] = VertexShader(VertexTex,IndexTex,MVP,ThreadID);
 2:  if(!isFrontFace(V1,V2,V3))
 3:      swap(V1,V2); // Reverse vertex order of the back face
 4:  end if
 5:  BV ← GetBoungingVolume(V1,V2,V3);
 6:  FrustumClipping(BV,ScreenWidth,ScreenHeight);
 7:  for y = BV.miny : BV.maxy
 8:      for x = BV.minx : BV.maxx
 9:          if(isInsideTriangle(x,y,V1,V2,V3))
10:              ZNew ← InterpolateDepth();
11:              for i = 0 : DepthArraySize - 1
12:                  ZOld ← atomicMin(&DepthArray[x][y][i], ZNew);
13:                  if(ZOld == MAX_INTEGER ) // 0x7FFFFFFF
14:                      break;
15:                  end if
16:                  ZNew ← max(ZNew, ZOld);
17:              end for
18:          end if
19:      end for
20:  end for
```

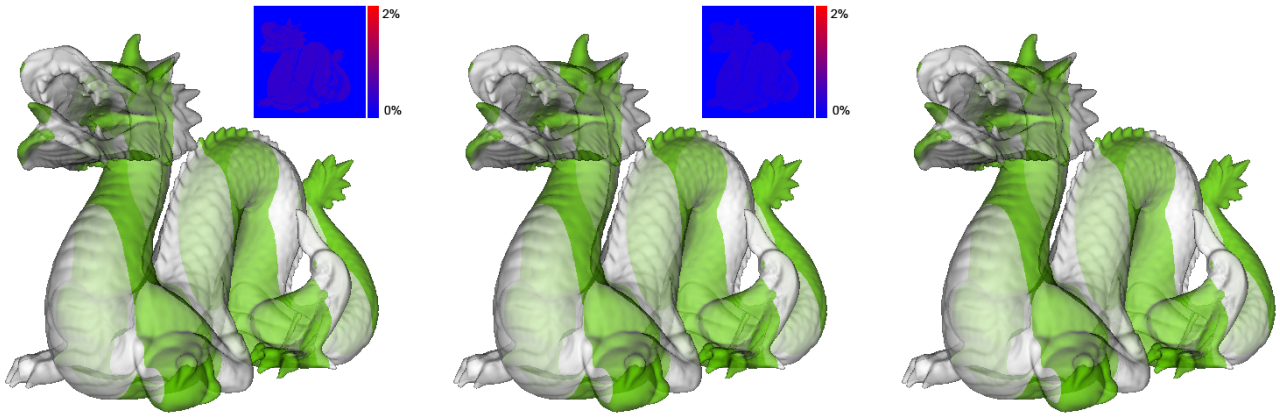
Table 1: *Pseudocode of the multi-depth test scheme for single pass depth gather.*

4.2 A-buffer Scheme

The multi-depth test scheme performs fragment sorting in global memory in a time complexity of $O(N^2)$. It will suffer from high memory latency for complex scenes. In addition, the alternative strategy to capture depth and color all together may cause artifacts and waste extra memory to store multiple copies of the depth value. So we resort to a second scheme to alleviate the problems.

Inspired by [Carpenter 1984], we allocate a fixed size *struct* (such as $float2$ for depth and color in order-independent transparency) array per pixel in global memory. We also set up a fragment counter for each pixel location. The counter is initialized to 0 by a CUDA kernel in a similar way for more throughput. Each incoming fragment will first increase the corresponding counter by 1 using the *atomicInc* operation. The *atomicInc* operation reads a 32-bit word located in global or shared memory, increases its value by 1, stores the result back to the memory at the same address, and returns the source value. In other words, the k^{th} incoming fragment of a pixel will atomically increase the counter to $k+1$ and return the source value k. Then the depth value and the additional attributes of the k^{th} fragment can be stored into the k^{th} entry of the array in submission order without read-modify-write hazards. The captured fragments will be sent to a post-sort CUDA kernel, with each thread handling only one pixel. All fragments per pixel will be loaded into an array allocated in the registers of that thread and sorted by insert sort or bitonic sort. The results then can be directly accessed in correct depth order for deferred shading in post-processing without extra writes and reads in global memory.

This scheme overcomes the problem of the multi-depth test scheme in that it can capture multiple fragment attributes simultaneously.



(a) Multi-Depth Test Scheme (b) A-buffer Scheme (c) Depth Peeling

Figure 2: ***Transparent effect on Stanford Dragon (871K triangles).*** *(a) Rendered by multi-depth test scheme; (b) Rendered by A-buffer Scheme; and (c) Rendered by depth peeling. The difference maps shown at the top right corners of (a) and (b) display the differences between our methods and depth peeling.*

Meanwhile, the sorting is performed at the register level without memory latency, that is more efficient than sorting in global memory for scenes with high depth complexity.

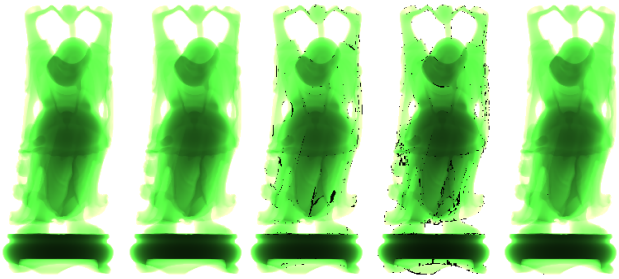
5 Results

This section shows the experimental results of our two fragment sorting schemes. We also compare them with the ground truth rendered by depth peeling, k-buffer and bucket depth peeling. All results are taken on a commodity PC of Intel Duo Core 2.4G Hz with 3GB memory, and NVIDIA Geforce 280 GTX (compute capability 1.3) with CUDA version 2.1.

Transparency. Figure 2 shows order-independent transparent effect on Stanford Dragon rendered by our two schemes for a 512x512 viewport. The depth values and the corresponding color attributes are captured and sorted in ascending order in a single geometry pass, and blended in front-to-back order during post-processing. Both results of our two schemes are almost identical to depth peeling, with only slight differences as shown in the difference maps in Figure 2. This is mainly caused by different implementation details as well as the quantization errors of the color attribute. The error rate of multi-depth test scheme is slightly higher than that of A-buffer scheme due to the lost precision of the depth values during packing as described in the previous section.

Translucency. The translucent effect can be simulated by only accounting for absorption and ignoring reflection [NVIDIA 2005; Akenine-Möller et al. 2008]. The ambient term can be computed using Beer-Lambert's law, and the light travel distance through the model can be accumulated by the thickness between every two successive layers per pixel (See Figure 3). The Fresnel' effect can also be rendered using Schlick's approximation as described in [Bavoil et al. 2007]. More results generated by our techniques can be found in Figure 4.

Other applications such as volume rendering, depth-of-field, shadow mapping and refraction could also greatly benefit from our algorithms.



(a) (b) (c) (d) (e)

Figure 3: ***Translucent effect on Buddha (1,087K triangles) with different methods.*** *(a) Rendered by multi-depth test scheme; (b) Rendered by A-buffer scheme; (c) Rendered by bucket depth peeling; (d) Rendered by k-buffer without modifications; and (e) Rendered by depth peeling.*

5.1 Performance Analysis

The performance of our FreePipe is compared to OpenGL for opaque rendering on a set of common models as shown in Table 2. The performance is comparable to the traditional graphics pipeline for large scenes where vertex transformations dominate the computation. However, the performance will be nearly halved for small scenes due to the doubled atomic operations for depth and color attributes.

Model	Dragon	Buddha	Lucy	Neptune	Bunny
Tri No.	871K	1.0M	2.0M	4.0M	70K
FreePipe	322fps	256fps	183fps	98fps	645fps
OpenGL	425fps	317fps	212fps	47fps	1255fps

Table 2: ***Frame rates(fps) for various scenes by FreePipe and OpenGL.*** *All the scenes are rendered for opaque surfaces at a viewport of 512x512 at the same view angle on Windows XP 32 bit platform. The block size of CUDA kernel in our methods is 64 for rasterization and post-processing, and 512 for initializing the render targets.*

(a) Neptune (b) Tank

(c) Lucy (d) Buddha

Figure 4: ***More results of our techniques.*** *(a) Transparent effect on Neptune rendered by multi-depth test scheme; (b) Transparent effect on Tank rendered by A-buffer scheme; (c) Translucent effect on Lucy rendered by multi-depth test scheme; and (d) Fresnel's effect on Buddha rendered by A-buffer scheme.*

The two schemes for multi-fragment effects are evaluated by the performance of translucent effect on the same models with different methods. The analysis demonstrates that our algorithm has significant performance improvement over the previous methods that are based on the traditional pipeline while maintaining correct results, as shown in Table 3. For a scene with depth complexity N, both schemes gain a performance improvement of about N times to depth peeling, and $N/2$ times to dual depth peeling. Both k-buffer and bucket depth peeling have similar speedup as ours but the former suffers from severe read-modify-write hazards while the latter is subject to fragment collisions (See Figure 3). The speedups of the two schemes compared to stencil routed A-buffer and the extensions of bucket depth peeling are more than two times due to the two geometry passes needed for these methods. In addition, our schemes outperform the extension of k-buffer [Liu et al. 2006] that needs multiple passes to assure correct results. Both of our two schemes have significant speedup compared to depth peeling especially for large scenes because most memory latency of the atomic operations in global memory overlaps that of vertex fetching while rasterizing, thus can be hidden by the thread scheduler. For small scenes such as bunny model, performance degrades due to dense atomic operations in global memory. In particular, multi-depth test scheme is a little slower than A-buffer scheme because the sorting in multi-depth test scheme is performed in global memory with high latency, while in A-buffer scheme, fragments are loaded into the registers and sorted without latency.

5.2 Memory Analysis

For the scenes in Table 3, we allocate 20 layers for storage since their maximum depth complexities are all near 20. So the memory consumption of the render target array for translucency by multi-depth test scheme is 20 MB. A-buffer scheme only requires 1 MB extra memory for the fragment counter per pixel. The amounts for both schemes are acceptable on current hardware. However, for high definition resolutions and irregular distributed scenes, both schemes may have a sparse memory layout in render targets that is a big memory overhead. Both stencil-routed A-buffer and k-buffer need similar amount as our algorithm. The bucket depth peeling and its extensions need much more memory due to the massive usage of MRTs. The dual depth peeling needs 4 MB memory for both the nearest and furthest layers, while in contrast, depth peeling only need 2 MB memory for the nearest layer in the previous pass and the current pass. In summary, our algorithm is a trade-off between memory consumption and performance. However, all other fragment level methods have similar problems.

Model	Dragon	Buddha	Lucy	Neptune	Bunny	Mem.
MDTS	297f	257f	192f	120f	487f	20MB
ABS	363f	309f	220f	132f	614f	21MB
K-buffer	304f	252f	149f	46f	875f	20MB
BDP	317f	256f	160f	49f	645f	32MB
BDP2	142f	116f	80f	25f	322f	64MB
ADP	116f	98f	75f	24f	212f	52MB
SRAB	106f	89f	49f	21f	334f	16MB
	2g	2g	2g	2g	2g	
Liu	60f	42f	20f	6f	183f	20MB
	5g	6g	9g	8g	6g	
DDP	40f	34f	23f	8f	160f	4MB
	8g	8g	8g	6g	8g	
DP	29f	26f	14f	5f	128f	2MB
	13g	13g	13g	9g	13g	

Table 3: ***Frame rates(f) and geometry passes(g) for various scenes.*** *The multi-depth test scheme (MDTS) and the A-buffer Scheme (ABS) are evaluated by comparing with k-buffer and its multi-pass extension (Liu), bucket depth peeling in a single pass (BDP) and two passes (BDP2), adaptive bucket peeling (ADP), stencil routed A-buffer (SRAB), dual depth peeling (DDP) and depth peeling (DP). All the scenes are rendered for translucency on the same environment as those in Table 2 except the stencil routed A-buffer on Vista. The last column shows the memory consumption of the scenes for each method.*

5.3 Limitation

The proposed schemes, however, have several limitations. First of all, since the 64-bit $atomicMin/atomicMax$ operation is not yet available in CUDA, the multi-depth test scheme will suffer from artifacts when capturing multiple fragment attributes. We believe that the next version of CUDA will add support to these functions so that multiple attributes can be handled efficiently. In addition, performance will degrade when the depth complexity of the scene is very high, due to the non-linear sorting complexity. The third problem is about the memory consumption as discussed above. This problem can be partially alleviated in a similar way as [Zhou et al. 2009], by dividing the screen into tiles and allocating a global memory buffer of an estimated size for each tile in a preprocessing pass.

6 Future Work

Occlusion culling is also difficult to be solved efficiently on current graphics hardware for a similar reason: it is hard to determine whether a current fragment will be occluded by future fragments. In the framework of our system, we may divide the screen into tiles and build a hierarchical depth buffer in global mem-

ory [Morein 2000; Hasselgren et al. 2009]. Each tile only maintains an over-estimated maximum depth value of all the fragments within that tile. The bounding volume of all the triangles per block can be pre-computed and projected to screen space, where the minimum depth value will be tested on each tile that it covers. If it is greater than the maximum depth values of these tiles, the whole block of triangles has been entirely occluded and so can be culled before rasterization. Otherwise, each triangle will be rasterized and tested per pixel where the maximum depth value of each covered tile will be simultaneously updated by that of the bounding volume using the *atomicMax* operation. Within each block, threads among different warps can communicate through shared memory and synchronization as shown circled with dash lines in Figure 1. The culling test is performed on batches of triangles without involvement of the CPU thus could be more efficient. Other operations such as frustum clipping, back face culling may also be accelerated in a similar way with only a few extra computations.

7 Conclusions

This paper presents FreePipe, a demonstration system for fully programmable rendering pipeline in parallel. The whole graphics pipeline is open to programmers without any hardware modifications. Within this architecture, two schemes are proposed for efficient rendering of multi-fragment effects in a single geometry pass using CUDA atomic operations. Experimental results show significant speed improvements over the previous methods with accurate results. Other problems may also benefit from the flexible control over all pipeline stages. Our system is a proof-of-concept prototype by now. We believe a mature programmable graphics pipeline using CUDA will be developed in the near future.

Acknowledgments. We would like to thank anonymous reviewers for their constructive comments and suggestions and Sean McDirmid for the proofreading of the final paper. We also like to acknowledge Stanford 3D Scanning Repository and AIM@SHAPE Repository for providing the meshes used in our paper. This work has been supported by National Basic Research Program of China (973 Program) (Grant No. 2009CB320802), National 863 High-Tec. Project (Grant No. 2008AA01Z301), National Science Foundation of China (Grant No. 60573155) and University of Macau Research Grant.

References

AILA, E., MIETTINEN, V., AND NORDLUND, P. 2003. Delay streams for graphics hardware. *ACM Transactions on Graphics*, 792–800.

AKENINE-MÖLLER, T., HAINES, E., AND HOFFMAN, N. 2008. *Real-Time Rendering*, third ed. A.K. Peters.

BAVOIL, L., AND MYERS, K. 2008. Order independent transparency with dual depth peeling. Tech. rep., NVIDIA Corporation.

BAVOIL, L., CALLAHAN, S. P., LEFOHN, A., AO L. D. COMBA, J., AND SILVA, C. T. 2007. Multi-fragment effects on the GPU using the k-buffer. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, 97–104.

BAVOIL, L., CALLAHAN, S. P., AND SILVA, C. T. 2008. Robust soft shadow mapping with backprojection and depth peeling. *journal of graphics, gpu, and game tools 13*, 1, 19–30.

BLYTHE, D. 2006. The Direct3D 10 system. *ACM Transactions on Graphics 25*, 3, 724–734.

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.

CARR, N., MECH, R., AND MILLER, G. 2008. Coherent layer peeling for transparent high-depth-complexity scenes. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, 33–40.

CATMULL, E. E. 1974. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah.

CEDERMAN, D., AND TSIGAS, P. 2008. A practical quicksort algorithm for graphics processors. In *Proceedings of the 16th Annual European Symposium on Algorithms*, 246–258.

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The reyes image rendering architecture. In *Computer Graphics (Proceedings of ACM SIGGRAPH 87)*, ACM, vol. 21, 95–102.

EISEMANN, E., AND DÉCORET, X. 2006. Fast scene voxelization and applications. In *SIGGRAPH 2006 Technical Sketch Program*.

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

FATAHALIAN, K., LUONG, E., BOULOS, S., AKELEY, K., MARK, W. R., AND HANRAHAN, P. 2009. Data-parallel rasterization of micropolygons with defocus and motion blur. In *Proceedings of the Conference on High Performance Graphics 2009*, ACM, 59–68.

GOVINDARAJU, N. K., HENSON, M., LIN, M. C., AND MANOCHA, D. 2005. Interactive visibility ordering and transparency computations among geometric primitives in complex environments. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, 49–56.

GOVINDARAJU, N. K., RAGHUVANSHI, N., HENSON, M., TUFT, D., AND MANOCHA, D. 2005. A cache-efficient sorting algorithm for database and data mining computations using graphics processors. Tech. rep., University of North Carolina-Chapel Hill.

HASSELGREN, J., MUNKBERG, J., AND AKENINE-MÖLLER, T. 2009. Automatic pre-tessellation culling. *ACM Transactions on Graphics 28*, 2.

HORN, D. R., SUGERMAN, J., HOUSTON, M., AND HANRAHAN, P. 2007. Interactive k-d tree gpu raytracing. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, 167–174.

JON HASSELGREN, T. A.-M. 2007. PCU: the programmable culling unit. *ACM Transactions on Graphics*, 92.

JOUPPI, N. P., AND CHANG, C.-F. 1999. z^3: an economical hardware technique for high-quality antialiasing and transparency. 85–93.

LIU, B.-Q., WEI, L.-Y., AND XU, Y.-Q. 2006. Multi-layer depth peeling via fragment sort. Tech. rep., Microsoft Research Asia.

LIU, F., HUANG, M.-C., LIU, X.-H., AND WU, E.-H. 2009. Efficient depth peeling via bucket sort. In *Proceedings of the 1th High Performance Graphics conference*, 51–57.

MAMMEN, A. 1989. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Computer Graphics and Applications 9*, 4, 43–55.

MARK, W. R., AND PROUDFOOT, K. 2001. The F-buffer: a rasterization-order fifo buffer for multi-pass rendering. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, 57–64.

MOREIN, S. 2000. ATI Radeon - HyperZ technology. In *Proceedings of the Hot 3D Workshop on Graphics Hardware*.

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

NVIDIA. 2005. GPU programming exposed: the naked truth behind nvidia's demos. Tech. rep., NVIDIA Corporation.

NVIDIA. 2008. NVIDIA CUDA: Compute unified device architecture. NVIDIA Corporation.

POPOV, S., GÜNTHER, J., SEIDEL, H.-P., AND SLUSALLEK, P. 2007. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum 26*, 3, 415–424.

SATISH, N., HARRIS, M., AND GARLAND, M. 2009. Designing efficient sorting algorithms for manycore GPUs. In *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium*, 1–10.

SEGAL, M., AND AKELEY, K. 2009. The OpenGL graphics system: A specification.

SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., , AND HANRAHAN, P. 2008. Larrabee: A many-core x86 architecture for visual computing. *ACM Transactions on Graphics 27*, 3.

SENGUPTA, S., HARRIS, M., ZHANG, Y., AND OWENS, J. D. 2007. Scan primitives for GPU computing. In *Proceedings of the 22nd ACM Siggraph/Eurographics Symposium on Graphics Hardware*, 97–106.

SINTORN, E., AND ASSARSSON, U. 2008. Fast parallel GPU-sorting using a hybrid algorithm. *Journal of Parallel and Distributed Computing archive 68*, 1381–1388.

SUGERMAN, J., FATAHALIAN, K., BOULOS, S., AKELEY, K., AND HANRAHAN, P. 2009. Gramps: A programming model for graphics pipelines. *ACM Transactions on Graphics 28*, 4.

TATARINOV, A., AND KHARLAMOV, A. 2009. Alternative rendering pipelines on nvidia cuda. Tech. rep., NVIDIA Corporation.

WEXLER, D., GRITZ, L., ENDERTON, E., AND RICE, J. 2005. GPU-accelerated high-quality hidden surface removal. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 7–14.

WITTENBRINK, C. M. 2001. R-buffer: a pointerless a-buffer hardware architecture. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, 73–80.

ZHOU, K., HOU, Q., WANG, R., AND GUO, B. 2008. Real-time KD-tree construction on graphics hardware. *ACM Transactions on Graphics*.

ZHOU, K., HOU, Q., REN, Z., GONG, M., SUN, X., AND GUO, B. 2009. Renderants: Interactive REYES rendering on GPUs. *ACM Transactions on Graphics*.