DOI: 10.1111/j.1467-8659.2010.01725.x

Eurographics Symposium on Rendering 2010
Jason Lawrence and Marc Stamminger
(Guest Editors)

Volume 29 (2010), Number 4

Real-Time Concurrent Linked List Construction on the GPU

Jason C. Yang, Justin Hensley, Holger Grün, and Nicolas Thibieroz

Advanced Micro Devices, Inc.

Abstract

We introduce a method to dynamically construct highly concurrent linked lists on modern graphics processors. Once constructed, these data structures can be used to implement a host of algorithms useful in creating complex rendering effects in real time. We present a straightforward way to create these linked lists using generic atomic operations available in APIs such as OpenGL 4.0 and DirectX 11. We also describe several possible applications of our algorithm. One example uses per-pixel linked lists for order-independent transparency; as a consequence, we are able to directly implement fully programmable blending, which frees developers from the restrictions imposed by current graphics APIs. The second uses linked lists to implement real-time indirect shadows.

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types

1. Introduction

Linked lists are a common and basic data structure in computer science, and, in the past, have been difficult to construct on the GPU with good performance. Pixel shaders lacked scatter capability and could only write a fixed number of bytes to a set screen position.

Recent APIs (e.g., DirectX®11 and OpenGL®4.0) and hardware features include the ability to perform atomic memory operations to arbitrary locations, which is important for building linked lists and other complex data structures. In this paper, we describe a general method of dynamically constructing linked lists on the GPU fast enough for real-time rendering. The basic idea behind our algorithm is to use one buffer to store all linked-list node data and another buffer to store head pointers that reference the start of the linked lists in the first buffer. This is possible using atomic memory operations.

Linked list-style data structures have many rendering applications. We describe two examples – order-independent transparency and indirect shadowing. In general, our method can be used to implement programmable blend, which is a desired API feature.

The contributions of this paper include:

- Fast creation of linked lists of arbitrary size using existing GPUs and APIs.
- Integration of our method into the standard graphics pipeline.
- Example applications for rendering – order-independent transparency and indirect shadows.

2. Related Work

Linked lists can be created on the GPU using multi-pass rendering methods [RBA08], but this is not efficient.

Our method is similar to the A-buffer [Car84] style of per-pixel data structures and is the most general method of handling multiple fragments per-pixel. Developed for REYES software rendering, the A-buffer maintained an unbounded, sorted list of fragment data per-pixel. Fragment data included depth, color, transparency, and pixel coverage. Our method can also be thought of as a true implementation of the A-buffer on the GPU.

There have been many variations and optimizations to the A-buffer method that improve performance or reduce the memory requirements. The R-buffer [Wit01] is a proposed hardware implementation of the A-buffer that uses a single FIFO to store all scene fragments. The irregular Z-buffer [JLBM05] proposes modifications to existing hardware that would enable construction of linked lists. The F-buffer [MP01] [HPS], stencil-routed A-buffer [MB07], Z^3 algorithm [JC99], and k-buffer [CICS05] [BCL*07] are also

hardware variations of the A-buffer where the size of the per-pixel array is fixed.

There have also been GPU shader implementations that store multiple fragment data per-pixel. [ED06] constructs a slice map (a per-pixel bit mask) representing geometry depths while voxelizing a scene. However, storing one bit per voxel or depth layer limits the types of effects that can be rendered. [LWX06] uses multiple geometry passes to construct a fixed array of fragments with transparency. Expanding on the slice map idea, [LHLW09] uses MAX/MIN blending operations to capture multiple layers of geometry depth. Each layer or bucket will contain the exact depth; however, if there is contention, only the max or min depth will survive. To limit this problem, a slice map is used to adaptively build the buckets per-pixel. All of these methods are limited by the number of hardware render targets and, therefore, the number of layers that can be represented.

Closer to our work is the FreePipe architecture [LHLW10]. Although a fixed-size array per-pixel is used, data is stored in a global memory array and constrained only by the desired allocation size. A per-pixel counter value is atomically incremented and used to indicate the array location for the current fragment update. One of the key differences with our implementation is that our linked list array is not allocated per-pixel and can be of arbitrary length, so memory usage is more efficient. Also, by using hardware-implemented counters, our rendering performance is often better than when using memory operations. Finally, our method integrates into the standard graphics pipeline and can take advantage of features such as MSAA rendering.

[Mic10] demonstrates a method of exactly computing per-pixel array offsets into a buffer used to store fragment data for transparency calculations. Geometry is rendered to an accumulation buffer to determine the total number of fragments per-pixel. Then a prefix sum is computed for each pixel, which holds the sum of all the fragments in the preceding pixels. The scene is re-rendered and fragment data is stored using the prefix sum as the offset into the storage buffer. Although this solves the problem of memory efficiency, there is a performance penalty in computing the prefix sum and rendering the scene twice.

Other multi-pass methods are geared toward the processing of multiple fragments versus storage. Depth peeling-type methods ([Eve01], [BCL*07], [BM08]) use separate geometry passes to peel off layers in depth order. The drawback of depth peeling is that performance decreases as the number of depth layers increases, so the number of passes increases.

So far, we have focused the discussion on per-pixel data arrays. However, our method can be extended to more complex data structures. [ZHWG08] describe kd-tree construction that takes advantage of new scatter and atomic memory operations.

3. Algorithm Overview

Creating the linked-list data structure is straightforward and similar to traditional CPU linked lists. Our method currently only handles singly linked lists. Two buffers are required – the "head pointer" buffer and the "node" buffer. The node buffer contains the nodes of the linked lists, and the head pointer buffer is an array of pointers, each pointing to the start of a linked list in the node buffer. The node buffer can be of arbitrary data type, but generally a struct is used to encapsulate the payload and a "next" pointer, which is the location to the next element in the linked list or a special flag marking the end of the list (EOL). We also create a counter that is atomically incremented and represents the location of the next available space where a node can be written in the node buffer.

Using rendering as an example, we can build a linked list of fragment data at each pixel location. The head pointer buffer is allocated to the dimensions of the frame buffer and seeded with the EOL flag (e.g., -1), so in the initial state each head pointer indicates an empty list. The size of the node buffer must be large enough to handle all possible fragments, or there must be a scheme to handle overflow. The counter to the node buffer is set to zero, which means the first node to be inserted will be written to location zero in the node buffer.

Figure 1 illustrates the insertion of a fragment into the linked list-structure. For each pixel being rendered, instead of writing to a render target, the shader will:

1. Retrieve the address (P') of the next free space in the node buffer. This is done by reading and atomically incrementing the node buffer counter. This is thread-safe because of the atomic operation.
2. Perform an atomic exchange between the recently retrieved node address (P') and the original pointer (P) in the head pointer buffer at the current rendered pixel location. This results in the head pointer value pointing to the node element we are inserting into the current pixel location's linked list. In effect, we are building the linked list in reverse.
3. Use the old value (P) from the head pointer buffer as the next pointer value and write the node element to the node buffer at the location (P') from step 1. The old head pointer value either will point to the location of the next node in the linked list or is the EOL flag.

See Appendix A for a DirectX 11 shader sample.

Although we are using rendering to illustrate the linked list construction algorithm, we need not use rendering to create the linked lists (i.e., GPGPU) and can use other head pointer buffer dimensions. Other data structures, such as for binning, can be constructed as well.

3.1. Node Buffer Counter

The node buffer counter can be allocated as a global counter in GPU memory and can be updated using atomic opera-

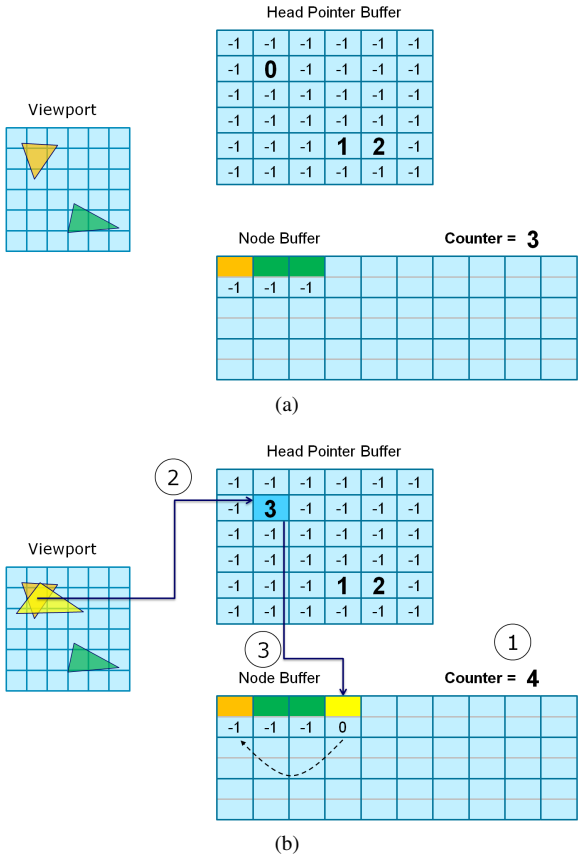


Figure 1: *Inserting one node into the linked lists. (a) Initial state before inserting the next fragment. (b) Step 1) Get next free space (counter = 3) and increment the counter. Step 2) Atomic exchange with the head pointer buffer (swap 0 with 3). Step 3) Insert the new node into the node buffer with the payload and next pointer (0) at location 3.*

tions. The disadvantage here is contention from all threads trying to update the same location in memory. One optimization is to use multiple counters to alleviate contention, but at the cost of compactness in memory because each counter must map to its own memory space (similar to previous works with fixed array sizes per-pixel).

DirectX 11 introduces new memory operations that allow multiple threads to write to memory in a linear fashion without the shader needing to know about order. Within a shader, this can be done implicitly by calling Append(T), which appends a value to the first free space in a buffer, or explicitly by calling IncrementCounter(), which returns and increments the underlying hardware counter associated with a buffer, and then manually writing to the returned address. In the later case, standard memory operations are used and are executed in parallel. In our implementation we use IncrementCounter() because it returns the counter information, and although it is a single hardware counter, performance is up to 60% faster compared to the single global counter stored in memory. Also, returned counter values become standard shader variables, so the same counter can be used as an address to multiple buffers of different types.

3.2. List Ordering

One limitation of our linked list method on current graphics hardware is that the order of thread execution on the hardware is not guaranteed. If the same input data is used in successive linked list creation passes, the order of the list elements might not be the same. Therefore, if ordering is important, such as for blending, some sorting is required as a final step. We demonstrate in Section 4 that this could be done entirely on the GPU using a shader.

3.3. Memory

Unlike on the CPU, memory on the GPU is not dynamically allocated for each node. A single node buffer is allocated to store all the potential nodes. If the total number of nodes to be inserted is known, the node buffer is simply allocated to match that number. Often times, such as during rendering, the total number of node insertions is unknown and can change from frame to frame. Therefore, the developer must take care in creating a buffer large enough to handle overflow or detect when overflow happens (e.g., checking the number of pixels rendered) and resize the node buffer as needed.

4. Applications

We present two applications of our linked-list algorithm – order-independent transparency (an example of general programmable blend) and indirect shadowing.

4.1. Order-Independent Transparency

Alpha blending is a classic computer graphics problem because it is an order-dependent operation, so sorting is often required. Sorting meshes is difficult if there are multiple interacting meshes or intersecting triangles. Order-independent transparency (OIT) is the rendering of semi-transparent objects without relying on a predetermined sorting order. Depth peeling [Eve01]-based methods have been the traditional solution on the GPU.

We can apply our linked-list method to OIT by constructing a linked list of fragments per-pixel. By developing our implementation using DirectX 11, our method is directly integrated into the standard graphics pipeline and will be compatible with common features such as depth-stencil buffers and MSAA (Figure 2). The process is:

1. Render all opaque or background objects as usual, including depth testing.

Figure 2: *Stanford dragon rendered with our linked list method interacting with opaque objects.*

2. Render all objects with transparency, and – instead of writing to the render target – insert the fragment data into the per-pixel linked-list structure. Depth testing is still supported, so any fragments occluded by opaque elements in the previous step will be dropped.
3. Resolve each linked list by first sorting and then traversing the list in sorted order for alpha blending. This is performed in a single shader pass using a full screen quad.

The creation of a node buffer can become an unreasonable burden on video memory requirements when render dimensions, the size of the node structure, and the number of fragments to store become large. For this reason it is important to be economical with those variables when circumstances allow it. For instance, the node structure can be optimized so its variables are stored in packed formats instead of floating-point vectors (fragment color in particular would benefit greatly from this approach).

4.1.1. Sorting

Once the linked list is constructed, any number of sorting techniques can be used to sort the fragments into proper order. Since the number of fragments per list is relatively low, we have found that insertion sort works quite well. Besides reasonable performance, insertion sort has a big advantage: it enables applications to efficiently extract the n closest layers. This makes it extremely easy for developers to smoothly trade rendering accuracy for performance by limiting the number of layers that are displayed.

4.1.2. MSAA

Multi-sampling anti-aliasing (MSAA) is a graphics hardware feature allowing the rasterization of primitives at a higher granularity by defining and employing several samples per pixel allocated in a multi-sampled surface. Once rasterization is complete, a resolve operation is employed to convert the multi-sampled results to a normal-resolution buffer in order to produce anti-aliased contents.

	Teapot	Dragon
Linked list	743 fps	338 fps
Precalc [Mic10]	285 fps	143 fps
Depth peeling	579 fps	45 fps
Bucket depth peeling	N/A	256 fps †
Dual depth peeling	N/A	94 fps †

Table 1: *Performance results for different OIT methods. Data is normalized to ATI Radeon HD 5770 performance.*

Producing multi-sampled and ordered transparent geometry requires that all sorting and blending be performed on a per-sample basis. Unfortunately, storing individual samples in the node buffer would result in a dramatic increase in memory requirements, making the technique impractical for real-time scenarios. To overcome this issue, we define an additional member of the fragment data structure containing a bit field representing the pixel coverage of each fragment. The shader for the sorting pass can either be executed on a per-sample or per-pixel basis. If processed per-sample, a multi-sampled render target is bound and the shader will only process fragments that contribute to the particular sample as determined by the coverage information. MSAA hardware is used to perform the final resolve. If processed per-pixel, a non-multisampled render target is bound and the shader will sort and average the final contributions from all the samples in the pixel.

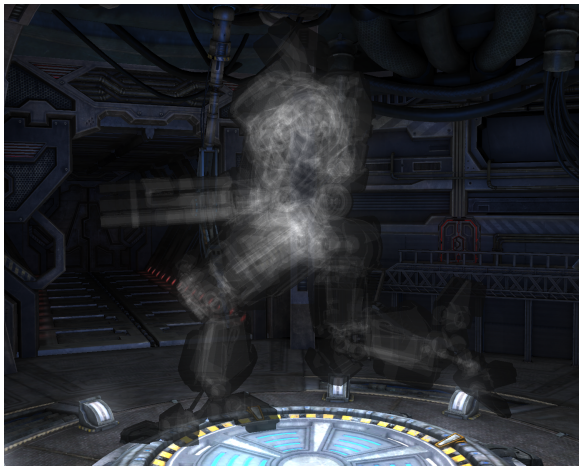
One potential problem with all per-pixel fragment list methods in general is sub-pixel intersecting triangles. We can employ the A-buffer solution of storing both min and max depth and estimating coverage, but at additional computational cost. MSAA can alleviate this problem, but we currently only store the centroid depth for all samples. We could further store the correct depth at each sample at additional memory cost, but this situation is rare and has not produced any noticeable artifacts.

4.1.3. Results

We tested our method on an ATI Radeon™HD 5770. In Table 1, we compare our method to several other OIT methods. When rendering a comparable view of the Stanford Dragon at a resolution of 512x512, our linked-list OIT method runs

† Bucket depth peeling and dual depth peeling performance results are adjusted to account for the differences between an NVIDIA GTX 280 and an ATI Radeon HD 5770. Based on publicly available benchmarks [Fut], an NVIDIA GTX 280 is 23% faster than an ATI Radeon HD 5770.

(a)

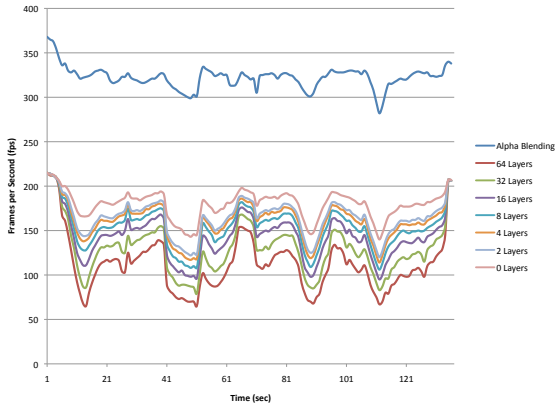


(b)

Figure 3: *(a) Realtime screen capture from the mecha animation with transparency computed using our linked list OIT technique algorithm. (b) illustrates the complexity of the model with the brightest pixels representing 64 layers of depth.*

	Min	Max	Ave
Alpha-blending	282 fps	368 fps	322 fps
0 layers	140 fps	215 fps	179 fps
2 layers	120 fps	215 fps	165 fps
4 layers	114 fps	215 fps	160 fps
8 layers	106 fps	215 fps	153 fps
16 layers	95 fps	215 fps	143 fps
32 layers	79 fps	215 fps	129 fps
64 layers	65 fps	215 fps	112 fps

(a)



(b)

Figure 4: *(a) shows the minimum, maximum, and average rendering performance of the mecha sequence with varying number of layers. (b) shows frames per second during the mecha sequence. "Alpha Blending" refers to using only standard alpha blending and results in an incorrect image. "0 Layers" refers to capturing all fragments, but not sorting or displaying any of the fragments. Data captured on a system with an ATI Radeon HD 5870 and an AMD Phenom II X4 940 running 64-bit Microsoft Windows 7. For the slowest frames, approximately 370K fragment lists are created which accounts for 40% of the frame.*

at 338 frames per second (FPS). When rendering with 2x MSAA, the application runs at 230 FPS, and with 4x MSAA it runs at 152 FPS. Currently, bucket depth peeling using the FreePipe architecture [LHLW10] is one of the fastest OIT methods. After adjusting for differences in the tested GPUs, our linked-list method is still appreciably faster than bucket depth peeling.

We have also implemented our OIT technique inside a complex DirectX 11 rendering engine to test in a more real-world environment. Figure 3(a) shows a screen capture from the mecha animation sequence. The animation lasts approximately two minutes and shows the mecha from a variety of views and poses. The entire scene consist of 602K triangles, with the mecha constituting 254K of the triangles. The number of physical layers in the mecha model varies depending on view, but the worst case view has almost 64 layers across the shoulders and hips of the model (Figure 3(b)). Performance data was captured for the mecha sequence rendered at 1280x720 resolution using an ATI Radeon HD 5870 GPU and an AMD Phenom II X4 940 CPU.

Figure 4 shows how the number of layers affects rendering performance over the duration of the mecha sequence and lists the minimum, maximum, and average frame rates for the different number of rendered layers. When using simple alpha blending, the sequence averages 322 FPS. If all fragments are captured, but none are sorted or rendered (labeled "0 layers"), the sequence averages 179 FPS. Given this difference in performance, it takes on average 2 ms to construct the unsorted linked lists on the ATI Radeon HD 5870 for each frame of the mecha sequence. The other data series

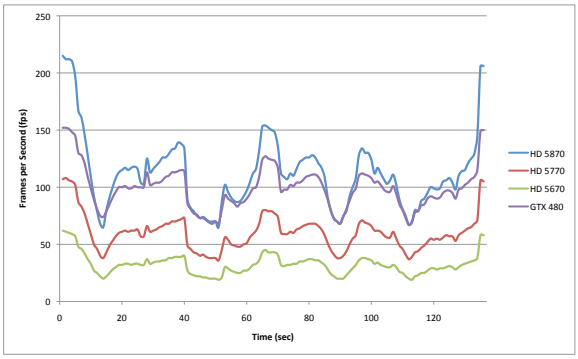


Figure 5: *Linked-list OIT rendering performance on different classes of GPUs for the mecha sequence. Even with rather modest GPUs, the method still remains real-time.*

refer to how many of the first n layers are displayed. The average cost to simply traverse the linked lists for each pixel can be estimated by looking at the difference between rendering 0 layers and 2 layers, which is approximately 0.5 ms. Additionally, by limiting the number of displayed layers, developers can easily trade performance for image correctness. For example, the peak signal-to-noise ratio between rendering all 64 layers and only the first 8 layers is only 13.5 dB, while rendering only the first 8 layers is almost 36% faster than rendering all layers.

Figure 5 shows how the OIT example scales on GPUs with different performance levels. As previously stated, the average framerate over the mecha sequence for an ATI Radeon HD 5870 (peak 2.72 TFLOPS) is 112 FPS. While a NVIDIA®GeForce®GTX 480 (1.35 TFLOPS) averages 98 FPS. With an ATI Radeon HD 5770 (peak 1.36 TFLOPS), the mecha sequence averages 60 FPS, and the ATI Radeon HD 5670 (peak 620 GFLOPS) averages 32 FPS. As can be seen from the relative performance of the tested GPUs, our linked-list OIT method performs quite well on a range of different GPUs, and makes it suitable for inclusion in a wide variety of real-time rendering applications.

4.2. Indirect Shadowing

Real-time indirect illumination techniques for fully dynamic scenes are an active research topic. There are a number of publications (e.g., [RGS09], [NW09], [Kap09], [DS06], and [DS05]) that deal with indirect one-bounce illumination, but they do not account for indirect shadows. Only a handful of methods for indirect illumination have been described that also include support for indirect shadows for fully dynamic scenes (e.g., [RGK*08], [TR09], and [KD10]).

The ability to concurrently construct linked lists on the GPU opens the door for a new class of algorithms to compute indirect shadows for fully dynamic scenes using ray tracing. The basic idea behind these techniques is to dynamically build data structures that contain lists of triangles.

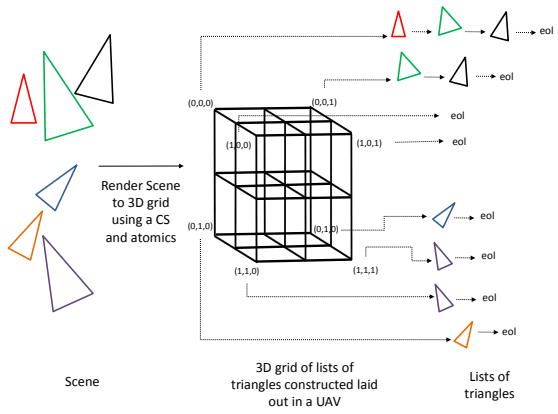


Figure 6: *3D grid of lists of triangles as seen from the light.*

These data structures are then traversed during ray tracing to detect if some amount of the indirect light is blocked. Although this approach can be used to implement ray tracing of dynamic scenes in general, the following discussion only considers indirect shadows.

We developed an example application for real-time indirect one-bounce illumination accounting for blocked indirect light utilizing a dynamically generated 3D grid of triangle lists.

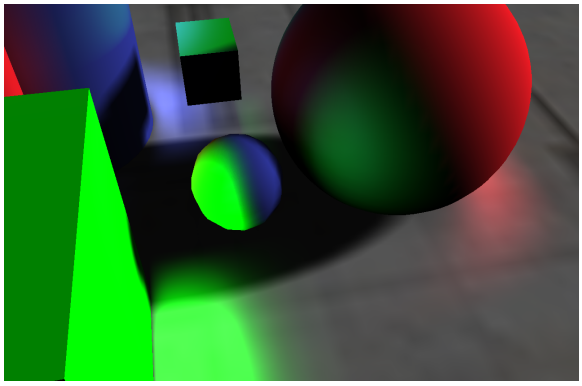
We start by rendering a g-buffer as seen from the eye and a reflective shadow map (RSM) [DS05] as seen from the light. Reflective shadow Maps use the observation that one-bounce indirect illumination from a light source is mainly caused by surfaces that are visible from the light's point of view.

Next, we dynamically construct a 3D grid containing the triangles of all dynamic blocker geometry of indirect light. A compute shader, looking at many triangles in parallel, inserts each triangle into all grid cells touched by the bounding box of the triangle. We chose to only render low level-of-detail versions of blocker objects to keep the number of triangles in each list small. Figure 6 illustrates the 3D grid representation.

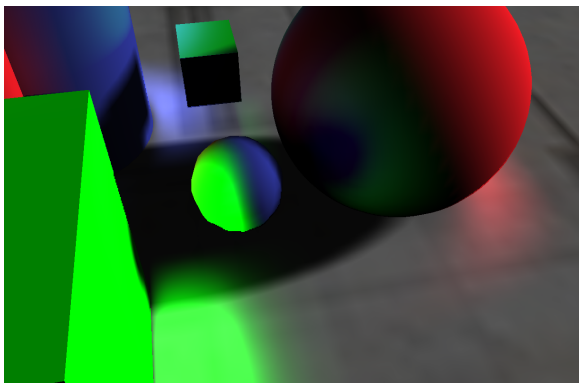
Then we render a screen aligend buffer A that accumulates all indirect light from a rectangular kernel of RSM texels at each screen location ignoring occlusions.

Now a low resolution buffer B is drawn that only accumulates blocked indirect light from RSM texels. Blocked RSM texels are detected by tracing a ray from the position of the current g-buffer pixel to the position of a texel in the RSM. Each grid cell that is entered by the ray is examined for triangles that block the ray similar in spirit to [WIK*06]. If a blocking triangle is found the ray is preempted and the RSM texel is marked as blocked

Lastly, we render the final image using the g-buffer. We combine a traditional shadow map for shadows, direct illumination and indirect illumination. Indirect illumination is



(a)



(b)

Figure 7: *(a) shows a closeup of one-bounce indirect light without shadowing. As can be seen, the background sphere is too bright given the foreground occluder. (b) shows the same scene with indirect shadowing using our linked-list construction algorithm.*

computed by subtracting a blurred version of the blocked indirect light (buffer B) from the buffer (A) that accumulates all indirect light. This has the nice effect that the resolution used to compute blocked indirect light is decoupled from the resolution used to compute indirect light.

4.2.1. Results

Figure 7 shows images rendered from our example implementation. When rendering at 1024x768 on an ATI Radeon HD 5870, our indirect shadowing algorithm runs at a rate varying from 60 FPS up to 200 FPS when using a 3D grid of triangle lists. An ATI Radeon HD 5770 varies from 45 FPS to 115 FPS for the same setup. When rendering with an ATI Radeon HD 5890, a dual-GPU board, our linked-list indirect-shadowing method is able to cast approximately 300 million rays per second, and the ray casting accounts for up to 9 ms of the frame render time. Our measurements show that our implementation can insert on the order of 300,000 blocker triangles per second into a 3d grid. Given this level of performance, indirect shadowing using our linked-list method is suitable for real-time applications when low level of detail versions of blocker objects are used.

5. Conclusion and Future Work

We have demonstrated that linked lists can be constructed in real time on GPUs and can be used for a variety of real-time rendering applications. Aside from indirect shadowing and order-independent transparency, the ability to create linked lists and similar data structure types will lead to many new rendering and general compute approaches (e.g., bucket sorting) on the GPU. For example, our technique could be used to directly extract layers to increase the accuracy of depth-of-field techniques instead of using techniques such as depth peeling. We have already demonstrated simplified ray tracing using indirect shadows, but we can extend this to more general ray-tracing. Finally, our method could be used to improve anti-aliasing algorithms by storing all triangle information per-pixel and computing accurate pixel coverage.

One advantage of our method is that it enables developers to implement programmable blending. Since the fragments are stored in sorted order for each pixel that needs to be shaded, applications can perform arbitrary shading on the data without being limited by restrictions of current graphics APIs. For example, by accounting for the distance between layers, it would be trivial to implement shading that properly attenuates the lighting to account for the density of material between layers.

The main disadvantage of using our technique to implement OIT is the potentially large memory requirement to store all the fragments rendered. One solution to reduce the memory requirement at the expense of increased transformation cost is to allocate a smaller set of buffers and to render the scene in tiles. Another idea is to store fragments at a reduced resolution and upscale the resolved contents, but would impact the quality.

Acknowledgements

We would like to acknowledge Jakub Klarowicz for the impetus to this work.

Appendix A: Fragment insert with append buffer

```
RWStructuredBuffer      RWStructuredCounter;
RWTexture2D<int>        tRWFragmentListHead;
RWTexture2D<float4>     tRWFragmentColor;
RWTexture2D<int2>       tRWFragmentDepthAndLink;

[earlydepthstencil]
void PS( PsInput input )
{
        float4 vFragment = ComputeFragmentColor(input);
        int2   vScreenAddress = int2(input.vPositionSS.xy);

        // Get counter value and increment
        int nNewFragmentAddress = RWStructuredCounter.
              IncrementCounter();
```

```
        if ( nNewFragmentAddress == FRAGMENT_LIST_NULL ) {
            return; }

        // Update head buffer
        int nOldFragmentAddress;
        InterlockedExchange(tRWFragmentListHead[
            vScreenAddress], nNewHeadAddress,
            nOldHeadAddress );

        // Write the fragment attributes to the node buffer
        int2 vAddress = GetAddress( nNewFragmentAddress );
        tRWFragmentColor[vAddress] = vFragment;
        tRWFragmentDepthAndLink[vAddress] = int2( int(
            saturate(input.vPositionSS.z))*0x7fffffff),
            nOldFragmentAddress );

    return;
}
```

References

[BCL*07] BAVOIL L., CALLAHAN S. P., LEFOHN A., COMBA J. L. D., SILVA C. T.: Multi-fragment effects on the gpu using the k-buffer. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2007), ACM, pp. 97–104. 1, 2

[BM08] BAVOIL L., MYERS K.: *Order independent transparency with dual depth peeling*. Tech. rep., NVIDIA Corporation, 2008. 2

[Car84] CARPENTER L.: The a -buffer, an antialiased hidden surface method. *SIGGRAPH Comput. Graph. 18*, 3 (1984), 103–108. 1

[CICS05] CALLAHAN S. P., IKITS M., COMBA J. L. D., SILVA C. T.: Hardware-assisted visibility sorting for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics 11*, 3 (2005), 285–295. 1

[DS05] DACHSBACHER C., STAMMINGER M.: Reflective shadow maps. In *I3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2005), ACM, pp. 203–231. 6

[DS06] DACHSBACHER C., STAMMINGER M.: Splatting indirect illumination. In *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2006), ACM, pp. 93–100. 6

[ED06] EISEMANN E., DÉCORET X.: Fast scene voxelization and applications. In *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2006), ACM, pp. 71–78. 2

[Eve01] EVERITT C.: *Interactive order-independent transparency*. Tech. rep., NVIDIA Corporation, 2001. 2, 3

[Fut] FUTUREMARK: Top performance graphics cards. http://service.futuremark.com/hardware/. 4

[HPS] HOUSTON M., PREETHAM A. J., SEGAL M.: Stanford tech report-cstr 2005-05 (2005), pp. 1-6 a hardware f-buffer implementation. 1

[JC99] JOUPPI N. P., CHANG C.-F.: Z3: an economical hardware technique for high-quality antialiasing and transparency. In *HWWS '99: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (New York, NY, USA, 1999), ACM, pp. 85–93. 1

[JLBM05] JOHNSON G. S., LEE J., BURNS C. A., MARK W. R.: The irregular z-buffer: Hardware acceleration for irregular data structures. *ACM Trans. Graph. 24*, 4 (2005), 1462–1482. 1

[Kap09] KAPALANYAN A.: Light propagation volumes in cryengine 3. *Advances in Real-Time Rendering in 3D Graphics and Games Course - SIGGRAPH 2009* (2009), Chapter N. 6

[KD10] KAPLANYAN A., DACHSBACHER C.: Cascaded light propagation volumes for real-time indirect illumination. In *I3D '10: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2010), ACM, pp. 99–107. 6

[LHLW09] LIU F., HUANG M.-C., LIU X.-H., WU E.-H.: Efficient depth peeling via bucket sort. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), ACM, pp. 51–57. 2

[LHLW10] LIU F., HUANG M.-C., LIU X.-H., WU E.-H.: Freepipe: a programmable parallel rendering architecture for efficient multi-fragment effects. In *I3D '10: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2010), ACM, pp. 75–82. 2, 5

[LWX06] LIU B.-Q., WEI L.-Y., XU Y.-Q.: *Multi-layer depth peeling via fragment sort*. Tech. rep., Microsoft Research Asia, 2006. 2

[MB07] MYERS K., BAVOIL L.: Stencil routed a-buffer. In *SIGGRAPH '07: ACM SIGGRAPH 2007 sketches* (New York, NY, USA, 2007), ACM, p. 21. 1

[Mic10] MICROSOFT CORPORATION: *DirectX Software Development Kit*, February 2010 ed., 2010. 2, 4

[MP01] MARK W. R., PROUDFOOT K.: The f-buffer: a rasterization-order fifo buffer for multi-pass rendering. In *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (New York, NY, USA, 2001), ACM, pp. 57–64. 1

[NW09] NICHOLS G., WYMAN C.: Multiresolution splatting for indirect illumination. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2009), ACM, pp. 83–90. 6

[RBA08] ROZEN T., BORYCZKO K., ALDA W.: Gpu bucket sort algorithm with applications to nearest-neighbour search. *Journal of WSCG 16*, 1-3 (2008). 1

[RGK*08] RITSCHEL T., GROSCH T., KIM M. H., SEIDEL H. P., DACHSBACHER C., KAUTZ J.: Imperfect shadow maps for efficient computation of indirect illumination. *ACM Trans. Graph. 27*, 5 (2008), 1–8. 6

[RGS09] RITSCHEL T., GROSCH T., SEIDEL H.-P.: Approximating dynamic global illumination in image space. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2009), ACM, pp. 75–82. 6

[TR09] T. RITSCHEL T. ENGELHARDT T. G. H.-P. S. J. K. C. D.: Micro-rendering for scalable, parallel final gathering. *SIGGRAPH Asia Comput. Graph.* (2009). 6

[WIK*06] WALD I., IZE T., KENSLER A., KNOLL A., PARKER S. G.: Ray tracing animated scenes using coherent grid traversal. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers* (New York, NY, USA, 2006), ACM, pp. 485–493. 6

[Wit01] WITTENBRINK C. M.: R-buffer: a pointerless a-buffer hardware architecture. In *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (New York, NY, USA, 2001), ACM, pp. 73–80. 1

[ZHWG08] ZHOU K., HOU Q., WANG R., GUO B.: Real-time kd-tree construction on graphics hardware. In *SIGGRAPH Asia '08: ACM SIGGRAPH Asia 2008 papers* (New York, NY, USA, 2008), ACM, pp. 1–11. 2