Efficient Layered Fragment Buffer Techniques 20

Pyarelal Knowles, Geoff Leach, and Fabio Zambetta

20.1 Introduction

Rasterization typically resolves visible surfaces using the depth buffer, computing just the front-most layer of fragments. However, some applications require all fragment data, including those of hidden surfaces. In this chapter, we refer to these data and the technique to compute them as a *layered fragment buffer* (LFB). LFBs can be used for order-independent transparency, multilayer transparent shadow maps, more accurate motion blur, indirect illumination, ambient occlusion, CSG, and relief imposters.

With the introduction of atomic operations and random access to video memory exposed via image units in OpenGL 4.2, it is now possible to capture all fragments in a single rendering pass of the geometry. This chapter describes and compares two approaches to packing this data: linked list–based and linearized array–based approaches.

Transparency is a well-known effect that requires data from hidden surfaces. It is used here to demonstrate and compare different LFB techniques. To render transparency, an LFB is constructed, and the fragments at each pixel are sorted. Figure 20.1 shows an example of fragment layers after sorting where each layer contains fragments of the same depth index. Note that surfaces are discretized and there is no fragment connectivity information. The sorted fragments are then blended in back-to-front order. Unlike polygon-sorting approaches to transparency, the LFB can resolve intersecting geometry and complex arrangements such as in Figure 20.2.

Previous approaches to capturing LFB data involve multiple rendering passes or suffer from read-modify-write problems, discussed in Section 20.2. OpenGL atomic

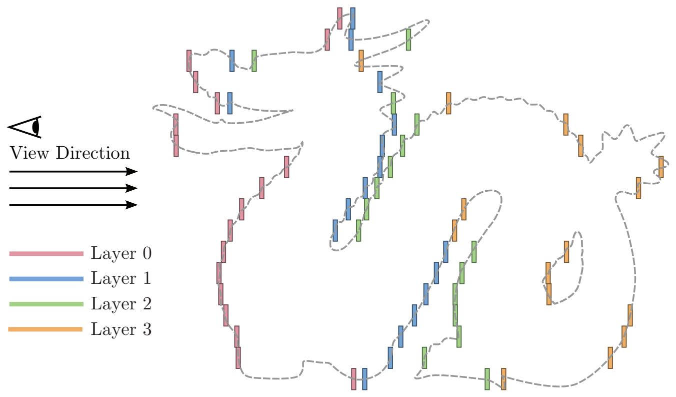

Figure 20.1. Fragment layers resulting from sorting fragments by depth.

operations allow the LFB to be accurately computed in a single rendering pass. The basic, *brute force LFB* technique for single-pass construction is to allocate a 3D array, storing a fixed number of layers in z for each pixel x, y. A per-pixel atomic counter is incremented to write fragments into the correct layer. Typical scenes have varying depth complexities, i.e., per-pixel fragment counts, so while the brute force LFB is fast, the fixed z dimension commonly wastes memory or overflows. The following are two general approaches to packing the data, solving this issue:

1. Dynamic linked list construction.

2. Array based linearization.

Both aim to pack the data with minimal overhead; however, there are some significant differences. Exploring these differences and comparing performance is the primary focus of this chapter.

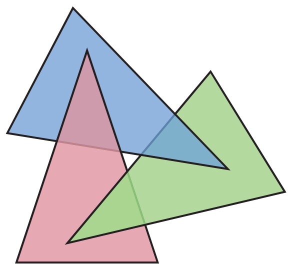

Figure 20.2. Cyclically overlapping geometry.

20.2 Related Work

There are a number of techniques to capture multiple layers of rasterized fragments. Before atomic operations in shaders were available, techniques used the depth buffer's fragment serialization and multiple rendering passes of the geometry. With the introduction of image units and atomic operations, techniques to capture fragments in a single pass have been proposed.

Depth peeling [Everitt 01, Mammen 89] is an established approach to capturing LFB data. The scene's geometry is rendered multiple times, capturing a single layer with each pass using the depth buffer. For complex geometry and high-depth complexities, this method is not practical for most real-time applications. Wei and Xu [Wei and Xu 06] use multiple framebuffer attachments to increase the speed of depth peeling by peeling multiple layers at once. This algorithm suffers from fragment collisions, i.e., concurrent read/write hazards; however, is guaranteed to resolve errors progressively with each pass. Dual depth peeling [Bavoil and Myers 08] improves the performance of depth peeling by peeling both front and back layers simultaneously using blending. Depth peeling via bucket sort [Liu et al. 09a] uses framebuffer attachments and blending to route fragments into buckets, defined by uniformly dividing the depth range. An adaptive approach that uses nonuniform divisions can be used to reduce artifacts caused by fragment collisions. Unlike the previous techniques, the k-buffer [Bavoil et al. 07] captures and sorts fragments in a single pass using insertion sort. Atomic operations were not available at the time, so this method suffers from fragment collisions that give rise to significant artifacts, although heuristics were described to reduce them.

Liu et al. [Liu et al. 09b] developed a CUDA rasterizer that atomically increments counters to push fragments onto constant-sized per-pixel arrays in one rendering pass. This is the brute force technique mentioned in Section 20.1. Yang et al. [Yang et al. 10] construct per-pixel linked lists of fragments dynamically on the GPU. We briefly describe this process in Section 20.3. The performance of this method originally suffered from atomic operation contention, discussed further in Section 20.5. Crassin [Crassin 10] presented a method to reduce atomic contention using "pages" of fragments. Around four to six fragments are stored in each linked-list node, thus reducing the atomic increments on the global counter at the cost of some overallocation of memory. Per-pixel counts are incremented for the index within the current page, and per-pixel semaphores are used to resolve which shader allocates new pages. We refer to this technique as the *linked pages LFB*.

An alternative to the linked-list approach is to pack the data into a linear array as described in Section 20.4. This technique is similar to the l-buffer concept [Lipowski 10], except for this method, packing is performed during rendering, reducing peak memory usage. A rudimentary implementation of this technique is included in the Direct3D 11 SDK [Microsoft Corporation 10]. The technique is also mentioned by Korostelev [Korostelev 10] and discussed by Lipowski [Lipowski 11], although this is the first detailed comparison as far as we are aware. Lipowski also packs the *lookup tables*, which reduces memory consumption by eliminating empty pixel entries.

20.3 The Linked-List LFB

The basic linked-list approach is relatively simple to implement. In one rendering pass, all fragments are placed in a global array using a single atomic counter for "allocation." Next, pointers are stored in a separate array of the same length to form linked lists. Each fragment is appended to the appropriate pixel's list via per-pixel head pointers. An atomic exchange safely inserts the fragment into the front of the list, following which, the fragment's next pointer is set to the previous head node:

```
node = atomicCounterIncrement(allocCounter);
head = imageAtomicExchange(headPtrs, pixel, node).r;
imageStore(nextPtrs, node, head);
imageStore(fragmentData, node, frag);
```

Figure 20.3 shows an example of the rendering result. The total number of fragments can be read from the atomic counter. Fragments in the same list are not guaranteed to be stored near each other in memory. Reading the fragment data is straightforward:

```
node = imageLoad(headPtrs, pixel).r;
while (node)
{
   frags[fragCount++] = imageLoad(fragmentData, node);
   node = imageLoad(nextPtrs, node).r;
}
```

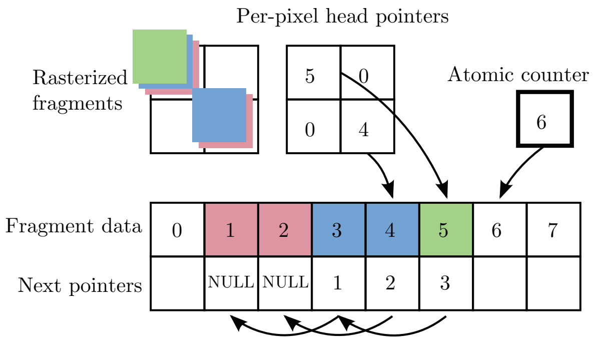

Figure 20.3. Per-pixel linked lists of fragments.

To determine the memory required for the global array, a preliminary fragment-counting pass can be performed, or the total memory required must be predicted. If insufficient memory is allocated in the latter case, data are discarded, or a complete rerender is needed.

20.4 The Linearized LFB

We now discuss the linearized LFB algorithm, which packs the fragment data into an array using an *offset* lookup table, as shown in Figure 20.4(a). This table is computed from per-pixel fragment counts, which is why a two-pass approach is used. The first pass calculates fragment counts, and the second regenerates and packs the fragment data. This produces a 1D array where all per-pixel fragments are grouped and stored one after the other, in contrast to the linked-list approach.

The linearized LFB rendering algorithm is summarized as follows:

1. Zero offset table.
2. First render pass: compute per-pixel fragment counts.
3. Compute offsets using parallel prefix sums.
4. Second render pass: capture and pack fragments.

An example of the count and offset data for three rasterized triangles is shown in Figure 20.5. In step 2, the count data is computed, for example, as shown in Figure 20.5(a). Only the fragment counts are needed, so additional fragment computation such as lighting is disabled.

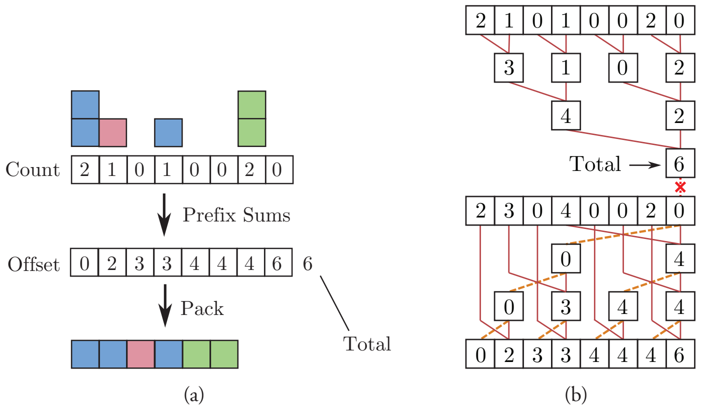

Figure 20.4. (a) Linearly packing fragment data and (b) the parallel prefix sum algorithm [Ladner and Fischer 80] used to create the offset table.

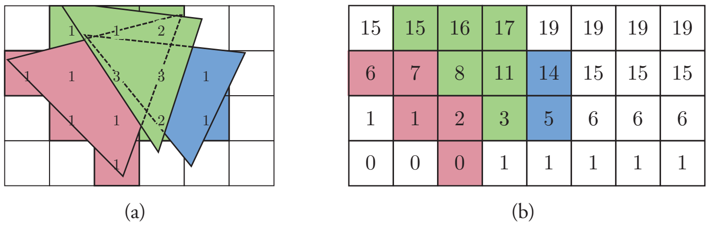

Figure 20.5. An example of (a) counts and (b) offsets for three triangles.

In step 3, offsets are computed, applying the parallel prefix sum algorithm [Ladner and Fischer 80] to the count data. The offsets are computed in place, overwriting the counts since the counts can be recalculated as the difference between consecutive offset values. This algorithm is visualized in Figure 20.4(b), and an example of the final offset table is shown in Figure 20.5(b). For simplicity, the input data for the parallel prefix sums is increased to the next power of 2, causing a maximum of double memory allocation for the offset data. Harris et al. [Harris et al. 07] describe methods for improving the speed of prefix sums using CUDA as well as handling non–power-of-2 data.

The total number of fragments is known from the prefix-sums computation. Thus, the exact memory needed is determined and allocated before packing during the main render.

In step 4, the second and main rendering pass of the scene is performed. Each incoming fragment atomically increments the offset value, giving a unique index that stores the fragment data. Figure 20.6 shows an example of the final linear LFB data. The data can be read using the difference in offsets, keeping in mind that they now mark the end of each fragment array:

```
fragOffset = 0;
if (pixel > 0)
   fragOffset = imageLoad(offsets, pixel - 1).r;
fragCount = imageLoad(offsets, pixel).r - fragOffset;

for (int i = 0; i < fragCount; ++i)
   frags[i] = imageLoad(fragmentData, fragOffset + i);
```

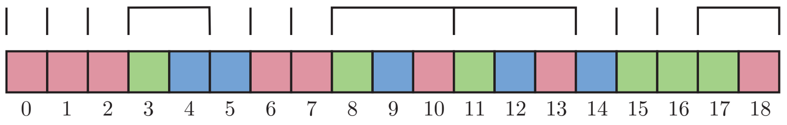

Figure 20.6. An example of linearized LFB fragment data. This, along with the offset table, Figure 20.5(b), is the algorithm's output.

20.4.1 Implementation Details

The LFB is a generic data structure, so having a cleanly accessible interface that multiple shaders can use is important. This is made more difficult with the two geometry passes of the linearized LFBs. Our approach is to implement an external preprocessor to parse the `#include` statement, which gives any shader access to the LFB interface. One could also use `ARB_shading_language_include`. The application's LFB object then sets the uniform variables for each shader that `#includes` it. This simplifies managing multiple LFB instances. Injecting `#define` values is also useful, for example, for setting constants, removing unused code, or creating permutations of shaders.

The following shows our basic linearized LFB rendering process, where `renderProgram` and `transparencyProgram` are OpenGL shader programs:

```
lfb.init();           //zero lookup tables
lfb.setUniforms(renderProgram);
render();             //fragment count pass
lfb.count();          //parallel prefix sums
lfb.setUniforms(renderProgram);
render();             //capture and store fragments
lfb.end();            //cleanup. also pre-sort if needed
...
lfb.setUniforms(transparencyProgram);
fullScreenTriangle(); //draw LFB contents
```

In this example, `renderProgram` computes each fragment's color and calls `addFragment(color, depth)`, defined in `"lfb.h"`. This call increments the fragment count in the first rendering pass and writes fragment data in the second. The transparent geometry drawn to the LFB in `render()` is then blended into the scene, rendering a full-screen triangle. The fragment shader of `transparencyProgram` calls `loadFragments()` and `sortFragments()`, defined in `"lfb.h"`, providing an array of sorted fragments to be blended.

We store the linearized LFB offset table and data in buffer objects and bind them to `ARB_shader_image_load_store` image units via `glTexBuffer` for shader access. Memory barriers must be set between each LFB algorithm step and parallel prefix sum pass with `glMemoryBarrier(GL_SHADER_IMAGE_ACCESS_BARRIER_BIT)`. Memory barriers force previous operations on memory to finish before further operations start. This stops, for example, prefix sums from being computed before the fragment-count pass has finished writing the results.

Fragment counts can be calculated in step 2 using either atomic increments or additive blending. Blending can be faster but is not supported with integer textures, so either the prefix sums must be performed with floats or a copy is required. Care must be taken to structure the implementation such that the fragment count between rendering passes matches exactly. For example, entire triangles intersecting the near- and far-clipping planes are rasterized. We ignore fragments outside the clipping planes by forcing the early depth test with `layout(early_fragment_tests) in;`.

If blending is used, the offset table can be zeroed with `glClear`. If blending is not used, zeroing a buffer object can be accomplished quickly using `glCopyBufferSubData` to copy a preallocated block of zeroed memory. This gives a small performance boost over writing zeros from shaders, at the cost of additional memory overhead.

The prefix sums can be computed in a vertex shader by calling `glDrawArrays(GL_POINTS, 0, n)` without client state attributes being bound. `gl_VertexID` can be used as the thread ID for the computation. Enabling `GL_RASTERIZER_DISCARD` prevents the point primitives proceeding to rasterization.

When reading the total fragment count during the prefix sums step, both `glGetBufferSubData` and `glMapBufferRange` are slow when operating directly on the offset table. As a workaround, we copy the total fragment count into a one-integer buffer and read from that instead. The same phenomenon occurs when reading the linked-list LFB atomic counter.

When rendering transparency, we sort the fragments in a local array in the shader, as access to global video memory is relatively slow. This imposes the limitation of a maximum number of fragments per pixel because the size of the local array is set at compile time. Saving the sorted data (or sorting in place for small depth complexities) may be beneficial for other applications that read fragments many times, for example, raycasting. The $O(n \log n)$ sorting algorithms perform worse than $O(n^2)$ algorithms for small n; the fastest sorting algorithm tested was insertion sort for up to 32 fragments. This is discussed further in Section 20.5.

We have found that shaders reading empty LFB fragment lists take an unexpectedly long time, so the stencil buffer is used to mask empty pixels. This provides a performance boost, especially when a significant fraction of the viewport is empty. We believe the cause is related to a slowdown from relatively large local arrays, in this case, the sorting array, for reasons about which we are uncertain.

20.5 Performance Results

We have implemented the brute force, linearized, linked-list, and linked-pages LFBs. Transparency is used as a benchmark, as other authors have done [Yang et al. 10, Crassin 10]. We compare performance and show that linearized and linked-list LFBs are competitive packing techniques. All timing experiments were performed using a Geforce GTX 460 at 1920×1080 resolution.

Updates to the OpenGL 4.2 implementation (late 2011) provide fast atomic counters. As such, there is no longer significant overhead from the atomic contention that originally hindered the linked-list approach. The basic linked-list LFB now performs better than the linked-pages variant [Crassin 10].

Two meshes, the dragon and atrium shown in Figures 20.7 and 20.8, are rendered to detail each algorithm's step times, given in Table 20.1. These scenes were chosen for their differing viewport coverage and depth complexities, shown in Figure 20.9.

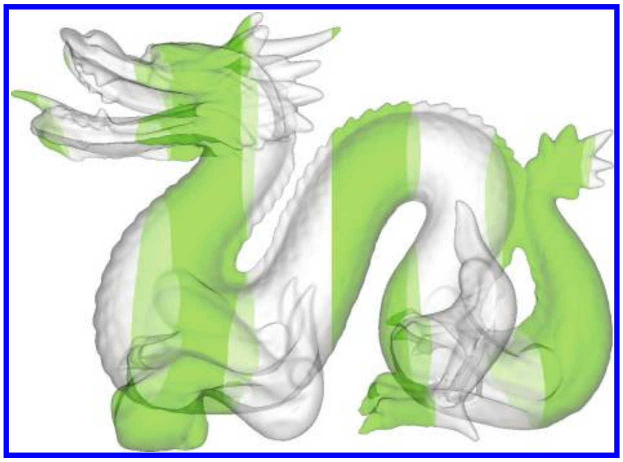

Figure 20.7. The Stanford dragon model, 871,414 triangles, striped to better show transparency. 1.3M total fragments.

Reading and sorting LFB data quickly becomes the bottleneck with more fragments. A goal in linearizing the LFB data is to give better memory access patterns; however, we observe little performance benefit from the sequential access. Both techniques perform similarly for these scenes, to within 10% of each other.

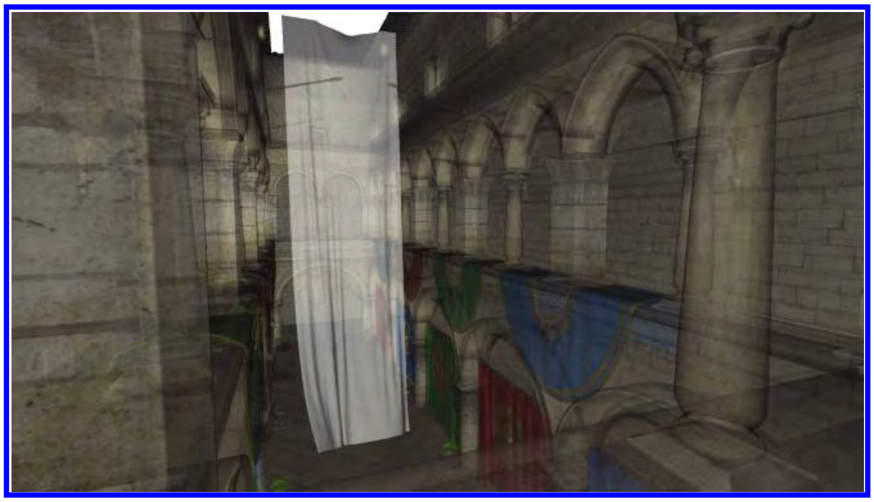

Figure 20.8. Sponza atrium by Frank Meinl, 279,095 triangles. 17.5M total fragments.

	Dragon		Atrium	
Algorithm step	L	LL	L	LL
Zero tables or pointers	0.02	3.0	0.02	3.00
Fragment count render	3.97		10.60	
Compute prefix sums	4.47		4.67	
Main LFB render	3.79	5.99	30.00	30.99
Read & blend fragments	8.52	10.16	95.05	88.08
Sort in shader	1.3	0.93	43.68	42.14
Total	22.07ms	20.10ms	177.05ms	171.24ms

Table 20.1. Linearized (L) and linked-list (LL) LFB algorithm step times.

Results vary considerably depending on resolution and viewing direction compared to typical rasterization using the depth buffer. To better investigate these variables, we use a synthetic scene of transparent, layered tessellated grids, as shown in Figure 20.10. An orthographic projection is used, and the grid size, layers, and tessellation are varied. A linear relationship between rendering time and total fragments is observed in Figure 20.11, where fragments are increased by scaling ten layers of 20K triangle grids to fill the viewport. Rendering more grid layers to increase the total fragments gives similar results. The brute force LFB is faster than other techniques, although it has much higher memory requirements. The rendering times in the synthetic scene broadly match that of the dragon and atrium for their fragment counts.

Depth complexity, or rather, fragment distribution, impacts sorting performance significantly. In Figure 20.12, the linearized LFB is used to render increasing grid layers while reducing the viewport coverage, keeping the total fragments (8M) and

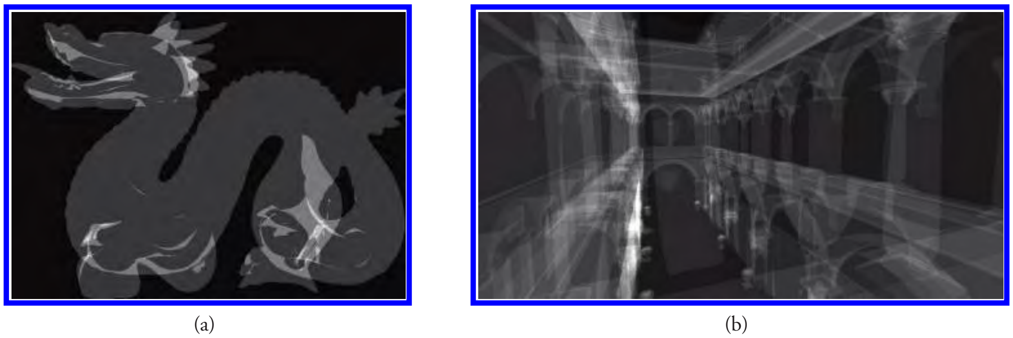

Figure 20.9. Depth complexities where black represents 0 fragments and white represents (a) 8 fragments and (b) 32 fragments.

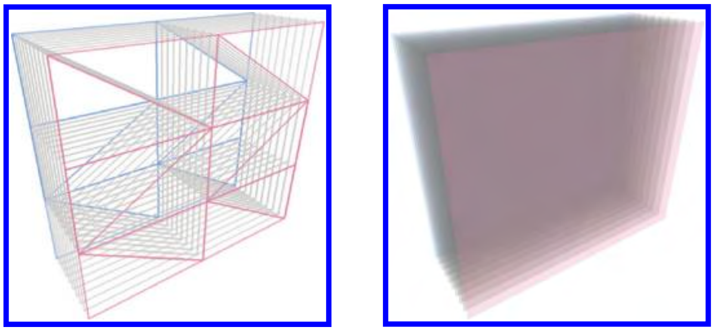

Figure 20.10. Transparent layered grids of polygons.

polygons (500–1000) approximately constant. Rendering times are similar for both linearized and linked-list LFBs, as sorting is a common operation. The sorting time becomes dominant after approximately 50 layers. Simply declaring and populating the sorting array (no sorting) with 256 `vec4` elements causes a 3–4× slowdown, compared to blending unsorted fragments directly from video memory (no local array). As expected, $O(n^2)$ insertion sort is faster for small n, for example, in the dragon and atrium scenes. We expect most scenes to have similar depth complexities; however, more complex scenes will benefit from $O(n \log n)$ sorting algorithms.

In terms of memory requirements, the overhead for the linearized LFB is the offset table, whereas the overhead for the linked-list LFB is both *head* and *next* pointers. In general, the linked-list LFB uses ≈ 25% more memory than the linearized LFB from the addition of `next` pointers, assuming 16 bytes of data per fragment.

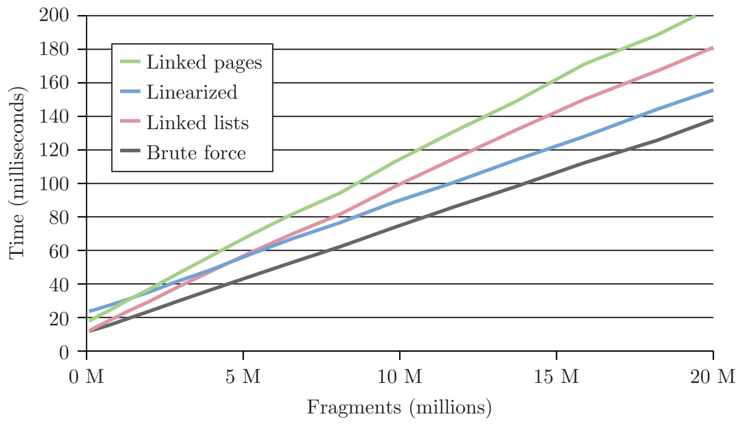

Figure 20.11. Comparing rendering times for different LFB techniques.

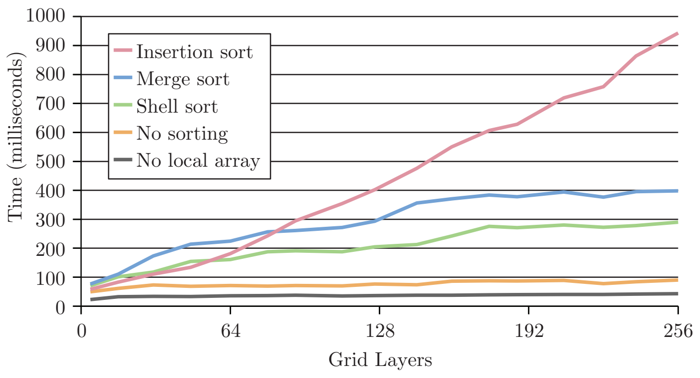

Figure 20.12. Varying depth complexity.

For 1920 × 1080 resolution, offsets are 8MB—in this case, 92KB more than `head` pointers, but potentially up to two times larger.

20.6 Conclusion

We have presented a comparison of linearized and linked-list LFBs, and results show both perform similarly for transparency. An expectation regarding the linearized LFB was that the sequential data layout would provide faster memory access. At this stage, this has not been observed to significantly affect the performance of transparency rendering.

The concept of capturing all output during rasterization, for example, in REYES [Carpenter 84], is well known, and the ability to do so in real time is becoming practical. In this chapter, we have focused on transparency; however, there are many applications that become possible or could be improved with the LFB. Screen-space effects such as ambient occlusion, indirect illumination [Yang et al. 10], motion blur, and depth of field suffer inaccuracies from missing data behind the front layer. Correct refraction and reflection [Davis and Wyman 07] require raycasting through multilayer data. Relief imposters [Hardy and Venter 10] produce incorrect results for concave objects, a problem that could be solved with the LFB.

Bibliography

[Bavoil and Myers 08] Louis Bavoil and Kevin Myers. "Order Independent Transparency with Dual Depth Peeling." Technical report, NVIDIA Corporation, 2008.

[Bavoil et al. 07] Louis Bavoil, Steven P. Callahan, Aaron Lefohn, João L. D. Comba, and Cláudio T. Silva. "Multi-Fragment Effects on the GPU Using the k-Buffer." In *Proceed-*

ings of the 2007 Symposium on Interactive 3D Graphics and Games, *I3D '07*, pp. 97–104. New York: ACM, 2007.

[Carpenter 84] Loren Carpenter. "The A-Buffer, an Antialiased Hidden Surface Method." *SIGGRAPH Computer Graphics* 18 (1984), 103–108.

[Crassin 10] Cyril Crassin. "Icare3D Blog: Linked Lists of Fragment Pages." http://blog.icare3d.org/2010/07/opengl-40-abuffer-v20-linked-lists-of.html, 2010.

[Davis and Wyman 07] Scott T Davis and Chris Wyman. "Interactive refractions with total internal reflection." In *Proceedings of Graphics Interface 2007*, *GI '07*, pp. 185–190. New York: ACM, 2007. Available online (http://doi.acm.org.ezproxy.lib.rmit.edu.au/10.1145/1268517.1268548).

[Everitt 01] Cass Everitt. "Interactive Order-Independent Transparency." Technical report, NVIDIA Corporation, 2001.

[Hardy and Venter 10] Alexandre Hardy and Johannes Venter. "3-View Impostors." In *Proceedings of the 7th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*, *AFRIGRAPH '10*, pp. 129–138. New York: ACM, 2010. Available online (http://doi.acm.org/10.1145/1811158.1811180).

[Harris et al. 07] Mark Harris, Shubhabrata Sengupta, and John D. Owens. "Parallel Prefix Sum (Scan) with CUDA." In *GPU Gems 3*, edited by Hubert Nguyen, Chapter 39, pp. 851–876. Reading, MA: Addison Wesley, 2007.

[Korostelev 10] Eugene Korostelev. "Order-Independent Transparency on the GPU Using Dynamic Lists." UralDev Programming Contest Articles 4, http://www.uraldev.ru/articles/id/36, 2010.

[Ladner and Fischer 80] Richard E. Ladner and Michael J. Fischer. "Parallel Prefix Computation." *Journal of the ACM* 27 (1980), 831–838.

[Lipowski 10] Jarosław Konrad Lipowski. "Multi-Layered Framebuffer Condensation: The l-Buffer Concept." In *Proceedings of the 2010 International Conference on Computer Vision and Graphics: Part II*, *ICCVG'10*, pp. 89–97. Berlin, Heidelberg: Springer-Verlag, 2010.

[Lipowski 11] Jarosław Konrad Lipowski. "d-Buffer: Letting a Third Dimension Back In..." http://jkl.name/~jkl/rnd/, 2011.

[Liu et al. 09a] Fang Liu, Meng-Cheng Huang, Xue-Hui Liu, and En-Hua Wu. "Efficient Depth Peeling via Bucket Sort." In *Proceedings of the Conference on High Performance Graphics 2009*, *HPG '09*, pp. 51–57. New York: ACM, 2009.

[Liu et al. 09b] Fang Liu, Meng-Cheng Huang, Xue-Hui Liu, and En-Hua Wu. "Single Pass Depth Peeling via CUDA Rasterizer." In *SIGGRAPH 2009: Talks*, *SIGGRAPH '09*, pp. 79:1–79:1. New York: ACM, 2009. Available online (http://doi.acm.org/10.1145/1597990.1598069).

[Mammen 89] A. Mammen. "Transparency and Antialiasing Algorithms Implemented with the Virtual Pixel Maps Technique." *Computer Graphics and Applications, IEEE* 9:4 (1989), 43–55.

[Microsoft Corporation 10] Microsoft Corporation. "DirectX Software Development Kit Sample." http://msdn.microsoft.com, 2010.

[Wei and Xu 06] Li-Yi Wei and Ying-Qing Xu. "Multi-Layer Depth Peeling via Fragment Sort." Technical report, Microsoft Research, 2006.

[Yang et al. 10] Jason C. Yang, Justin Hensley, Holger Grün, and Nicolas Thibieroz. "Real-Time Concurrent Linked List Construction on the GPU." *Computer Graphics Forum* 29:4 (2010), 1297–1304. Available online (http://dblp.uni-trier.de/db/journals/cgf/cgf29.html#YangHGT10).