S-buffer: Sparsity-aware multi-fragment rendering

Conference Paper · January 2012

DOI: 10.2312/conf/EG2012/short/101-104

CITATIONS
20

READS
165

2 authors:

Andreas Alexandros Vasilakis
Athens University of Economics and Business
31 PUBLICATIONS **183** CITATIONS
SEE PROFILE

Ioannis Fudos
University of Ioannina
74 PUBLICATIONS **1,161** CITATIONS
SEE PROFILE

S-buffer: Sparsity-aware Multi-fragment Rendering

Andreas A. Vasilakis † and Ioannis Fudos ‡

Department of Computer Science, University of Ioannina, Greece

Abstract

This work introduces S-buffer, an efficient and memory-friendly gpu-accelerated A-buffer architecture for multi-fragment rendering. Memory is organized into variable contiguous regions for each pixel, thus avoiding limitations set in linked-lists and fixed-array techniques. S-buffer exploits fragment distribution for precise allocation of the needed storage and pixel sparsity (empty pixel ratio) for computing the memory offsets for each pixel in a parallel fashion. An experimental comparative evaluation of our technique over previous multi-fragment rendering approaches in terms of memory and performance is provided.

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

1. Introduction

Capturing efficiently global information of a 3D scene is an important feature in many graphics applications that simulate complex rendering effects. Example effects include order independent transparency, volume and CSG rendering, trimming, shadow mapping, collision detection, voxelization and others all of which require processing of multiple fragments.

Storing multiple fragments efficiently in terms of time and space is a challenging task. A-buffer [Car84] was the first method to capture all fragments per pixel in a single pass. Fragments are stored into variable-length lists per pixel during geometry rendering, followed by a post-sorting process that correctly reorders fragments by depth. Variants have been proposed [MCTB11] that limit per-pixel storage requirements generating approximate results. Recently, [YHGT10] introduced an actual implementation of A-buffer on the gpu by performing concurrent linked list construction. The algorithm scales well and runs in linear time on the number of generated fragments, but its performance degrades rapidly in cases where heavy access on the gpu shared memory is necessary. If fragment overflow occurs, they propose to dynamically reallocate memory and then re-render the scene. Otherwise, much of the allocated memory goes unused.

We introduce *S-buffer*, an efficient and memory-friendly algorithm built on the A-buffer architecture without relying at linked-lists or fixed-array structures. Inspired by [Pee08, Lip10], we perform an additional rendering pass for counting fragments per pixel which enables us to allocate the exact amount of memory that we shall need. To optimize caching and data bus occupancy, we organize storage into variable contiguous regions (*bins*) for each pixel. Contrary to linear [Lip10] and common parallel [Pee08] prefix sum for generating per-pixel memory indices, we employ a randomized prefix sum in parallel by exploiting *pixel sparsity* (i.e. the fact that in many scenes there are many fragmentless pixels). An inverse mapping strategy is also presented to slightly improve performance. S-buffer successfully integrates into the standard graphics pipeline and can take advantage of features such as multi-sample rendering, gpu tessellation and instancing.

2. Related Work

[Pee08] computes buffer offsets for linearising A-buffer storage based on the maintenance of a fragment counter pass

† This author's work is co-funded by the European Union - European Social Fund (ESF) & National Sources,in the framework of the program "HRAKLEITOS II" of the "Operational Program Education and Life Long Learning" of the Hellenic Ministry of Education, Life Long Learning and religious affairs.

‡ fudos@cs.uoi.gr

Updated version of the final, definitive version of this paper, available at:
http://dx.doi.org/10.2312/conf/EG2012/short/101-104

and a subsequent prefix sum on the fragment counter data. However, the order of this algorithm depends on the active screen dimensions resulting in a performance downgrade even when rendering sparse scenes in higher resolutions. Closest to our work lies the *l-buffer* architecture [Lip10] that exploits pixel sparsity. However, a *serialized* process on the used pixels is performed to compute the memory offsets for each pixel.

Various techniques have been proposed to simulate the behavior of A-buffer architecture with reduced memory requirements. *F-buffer* [MP01] and *R-buffer* [Wit01] replace the linked list structure with a FIFO buffer to capture all incoming fragments. Z^3*-buffer* [JC99] sets an upper bound for the number of fragments stored per pixel. Similarly, *K-buffer* [BCL*07] and *Stencil Routed A-buffer* (or *SRAB*) [MB07] use fixed-size vectors able to capture up to k fragments of primitive pre-sorted scenes. There have been a few attempts to perform the entire rasterization process using a software graphics pipeline [LHLW10, PTO10]. However, they limit users to switch from the traditional hardware pipeline to a CUDA rasterizer. Readers may refer to a comprehensive survey [MCTB11] for a detailed description of the pros and cons in terms of memory and performance of many of the aforementioned alternatives.

3. Algorithm Overview

We introduce an efficient and memory-aware A-buffer implementation on the gpu based on real-time concurrent construction of per-pixel variable-length fragment bins. The idea is to accumulate the fragments which influence a pixel into a *counter buffer* by performing a fast rendering prepass (Section 3.1), followed by a *memory offset buffer* computation aimed at packing fragments for each pixel in adjacent position of memory (Section 3.2). Capturing the total number of generated fragments provides for dynamic and precise allocation of the required storage space. Memory referencing process is accomplished using a parallelised prefix sum on the randomly arriving non-empty pixels. Then, a subsequent rendering of the scene is performed to store the out-of-order fragments per-pixel starting from the memory location captured at the address buffer. Finally, a sorting mechanism is employed to reorder the fragments for each pixel before generating the final image (Section 3.3).

3.1. Fragment Count Pass

First, a rendering pass is employed to simultaneously extract the number of fragments affecting each pixel and the total number of fragments generated for all pixels. More specifically, the fragment accumulation can by implemented by turning off depth test and performing for each rendered fragment per pixel either ADD blending *one* into a 32-bit floating pixel format texture (R_32F) or thread-safe increment operations on a 32-bit unsigned integer buffer (R_32UI). The total number of rasterized fragments is computed by hardware occlusion queries and used to precisely estimate the size of the *node buffer* that will store the information for all fragments (RG_32F, R: color, G: depth).

3.2. Memory Referencing

Prefix sums on the counter buffer have been used in [Pee08] to generate the access location of *all* pixels in the node buffer. To avoid overheads for pixels with zero fragments, [Lip10] perform a prefix sum only on the non-empty pixels in a linear fashion, regardless of the order pixels are processed. This can be implemented using one *shared counter* (32UI) in the gpu memory which can be updated via atomic memory operations provided by the recent APIs. For each pixel processed, the current shared counter value is written out to the pixel local *address buffer* location, followed by an increment of the shared counter value by the pixel fragment count.

To alleviate congestion from all pixels trying to update the same memory location, we propose to apply S *multiple* gpu-accelerated shared counters: $C = \{C(0), \cdots, C(S-1)\}$. More specifically, non-empty pixels are decomposed into non-uniform groups using a simple hash function: $H(P) = (P.x + width * P.y)\%S$, where $P.x$ and $P.y$ denote the pixel position. We associate one shared counter to each group and perform in parallel the linear prefix sums for all groups.

After the completion of this process, each group of pixels maps to its own memory space by performing a prefix sum on the final values of the shared counters: $C_{pr}(i) = \sum_0^{i-1} C(i)$, where $C_{pr}(i)$ is the i-th resulting memory reference value. An inverse mapping technique is applied to boost by a factor of two the latter prefix sum process using information from the total number of the rendered fragments. We accomplish that by splitting the shared counters into two groups, $G_1 = \{C(0), \cdots, C(\lfloor \frac{S}{2} \rfloor)\}$ and $G_2 = \{C(\lfloor \frac{S}{2} \rfloor + 1), \cdots, C(S-1)\}$. The key idea is to perform forward prefix sum for the G_1 group and inverse prefix sum for the G_2 group. We define as *inverse*, the prefix sum that starts accumulating from the end of the processing set towards the start. Then, the memory offset for each pixel P is computed using the following equation,

$$offset(P) = \begin{cases} A(P), & \text{if } P \in G_1 \\ totalFragments\text{-}1\text{-}A(P), & \text{otherwise} \end{cases} \quad (1)$$

where $A(P) = address(P) + C_{pr}(H(P))$

Figure 1 illustrates a simple example of creating memory offsets applying 3 shared counters with forward mapping ($\{C(0), C(1), C(2)\} \in G_1$). A rendering pre-pass calculates the per-pixel fragment counters. We illustrate pixels with the same hash value by painting them with the same color. A sequential prefix sum is applied for each pixel group via atomically updating the associated shared counter. Without loss of

generality, we assume that pixels are processed from the top row to the bottom row. Forward prefix sums are performed to compute $C_{pr}(i)$ mapping each group of pixels to its own memory space. The head memory location for each pixel is finally computed using equation 1.

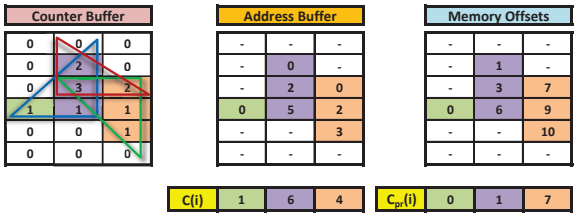

Figure 1: *S-buffer workflow when rendering a red, a blue and a green triangle.*

3.3. Fragment Storing - Resolve

In this phase, we perform an additional rendering pass to store pixel fragment data to each bin indicated by the location information generated in the previous phase. The associated fragment information is then written out to the given buffer index. The address buffer for the current pixel is then adjusted to the next free space.

Finally, we use *insertion sort* to correct the ordering of the captured sample fragments since it performs well when the number of generated fragments per pixel remains small (see also [YHGT10]). A large repertoire of multi fragment effects can be supported after sorting. Figure 2 illustrates transparency effects and CSG operations using S-buffer.

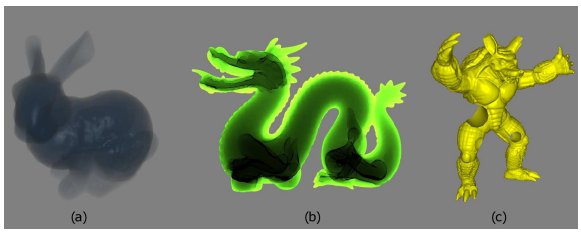

Figure 2: *(a) Transparency, (b) translucency and (c) CSG rendering on different models.*

4. Results

We present an experimental analysis of our S-buffer approach versus the other A-buffer realizations. We have measured performance in terms of frames per second (fps) and milliseconds (ms) and memory requirements in terms of MBytes for a set of different testing conditions. For the purposes of comparative time and space complexity evaluation, we have developed *PreCalc_OpenCL*, a faster variation of *PreCalc* [Pee08] which handles memory offsetting using an OpenCL-accelerated parallel prefix sum (provided by NVIDIA Corporation). Moreover, we have implemented *PreCalc_Fixed*, the fastest A-buffer which exploits a one-pass scheme by adapting per-pixel fixed-size arrays based on [Pee08]. This allows prefix sums to be efficiently obtained using a full-screen pass ($address(P) = (P.x + width * P.y) * array\_size$). Instead of using the software rasterizer of FreePipe [LHLW10], we have used an OpenGL-based implementation [Cra10] which performs similarly with the CUDA one. Finally, our variation that uses only one shared counter may be consider as an *advanced l-buffer* implementation. All methods are implemented using OpenGL 4.2 API and were tested on NVIDIA GTX 480 hardware.

Figure 3 shows how the performance of the memory-friendly A-buffer variants scales by moving from a sparse to a dense rendering of the Stanford Bunny positioned inside a cube (69463 faces, 12 depth layers) under a 1024×1024 viewport. l-buffer exhibits performance downgrade due to the linearisation of prefix sum which leads to $O(n)$ time complexity, where n is the number of the non-empty pixels. Performance is significantly boosted by increasing the number of S-buffer shared counters. Even with two global counters we match the PreCalc_OpenCL performance when the pixel sparsity remains high. Observe that our buffer exhibits its performance peak using about 30 counters. Since, final memory mapping is obtained through a linear prefix sum on the shared counters, performance starts downgrading when the number increases out of proportion. The Linked Lists technique, using only one rendering pass, has the worst behavior since it suffers from an $O(m)$ complexity, where $m(\gg n)$ is the number of generated fragments. Finally, an interesting observation is that the performance of PreCalc_OpenCL converges to S-buffer when the number of used pixels increases rapidly. Even when the rendering scene covers all pixels, our buffer performance is slightly better (7% faster) than the full parallel prefix sum solver of PreCalc_OpenCL.

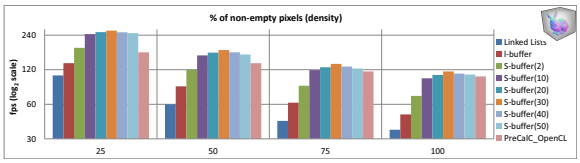

Figure 3: *Performance evaluation in fps ($\log_2$ scale) for rendering Stanford Bunny positioned inside a Cube at different clipping stages. S-buffer exhibits its performance peak using about 30 counters.*

Figure 4 illustrates performance evaluation of all A-Buffer variants on rendering Minoan Palace of the Knossos model (109168 faces, 25 depth layers, 45% used pixels) for a set of different screen resolutions. In general, we observe that fixed-size FreePipe and PreCalc_Fixed solutions outperform memory-aware variants. But this comes with the burden of

memory limitations which is discussed later on. KB and SRAB have the worst behavior since they have to carry out multiple iterations for capturing the entire scene information. PreCalc_OpenCL appears to perform quite well despite the synchronization penalties of OpenGL/OpenCL interoperability. S-buffer using 30 counters outperforms the other memory-friendly A-buffer variants, rendering at a 85% to 90% of the optimal frame rate (PreCalc_Fixed). Note that inverse memory mapping boosts S-buffer performance by $(13\%, 10\%, 6\%)$, where percentages in brackets denote of the acceleration for each of three testing resolutions: 640×480, 1024×768, and 1600×1200.

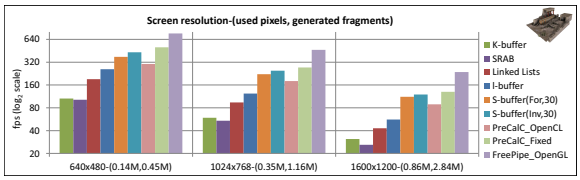

Figure 4: *Performance evaluation in fps ($\log_2$ scale) for rendering the Knossos model at different rendering dimensions. S-buffer with inverse mapping outperforms the other memory-friendly A-buffer variants.*

We further provide a time comparison of the memory referencing step for the buffers that include this step. S-buffer with 30 counters needs $(0.215ms, 0.425ms, 1.08ms)$ to compute memory offsets which is $\approx 10\times$ slower than the fastest PreCalc_Fixed $(0.027ms, 0.05ms, 0.11ms)$. Moving from inverse to forward mapping results at an extra $0.05ms$ cost for all resolutions which explains why inverse mapping boost is decreasing when moving to higher resolutions. PreCalc_OpenCL takes $(1.5ms, 2.45ms, 4.85ms)$ to compute the parallel prefix sum regardless of the pixel sparsity, which is $5\times$ to $7\times$ slower than our method. Finally, fragment-aware Linked Lists exhibits the worst performance by taking $(4.66ms, 10.4ms, 21.98ms)$.

In the context of storage requirements for the latter scenario, FreePipe and PreCalc_Fixed lead to increased memory requirements (60.94MB, 159MB, 380.86MB) most of which is not actually used (88%) due to their strategy to allocate the same memory for each pixel. K-Buffer (21.09MB, 54MB, 131.84MB) and SRAB (23.44MB, 60MB, 146.48MB) due to their nature, capture up to 8 fragments per pass and therefore need 30% less memory resources than previous bounded buffers. Conversely, PreCalc_OpenCL (8.02MB, 20.53 MB, 50.10MB) and S-buffer (6.99MB, 17.90 MB, 43.69MB) allocate the exact amount of memory needed since the number of fragment insertions is known apriori. Linked Lists needs slightly more memory resources for storing memory pointers with an extra linked list (8.73MB, 22.35 MB, 51.7MB). However, in cases where the number of fragments varies (camera or mesh animation) overflows may occur.

5. Conclusions

We have presented S-buffer, a two-pass A-buffer implementation on the gpu designed so as to take advantage of the fragment distribution and the sparsity of the pixel-space. S-buffer exhibits improved combined memory usage and performance behavior even in low pixel sparsity rasterizations. However, the need of an additional rendering step results in performance downgrade when compared to FreePipe.

References

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

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

[Cra10] CRASSIN C.: Fast and accurate single-pass a-buffer using opengl 4.0+, icare3d blog, 2010. 3

[JC99] JOUPPI N. P., CHANG C.-F.: Z3: an economical hardware technique for high-quality antialiasing and transparency. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (New York, NY, USA, 1999), HWWS '99, ACM, pp. 85–93. 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 *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2010), I3D '10, ACM, pp. 75–82. 2, 3

[Lip10] LIPOWSKI J. K.: Multi-layered framebuffer condensation: the l-buffer concept. In *Proceedings of the 2010 international conference on Computer vision and graphics: Part II* (Berlin, Heidelberg, 2010), ICCVG'10, Springer-Verlag, pp. 89–97. 1, 2

[MB07] MYERS K., BAVOIL L.: Stencil routed a-buffer. In *ACM SIGGRAPH 2007 sketches* (New York, NY, USA, 2007), SIGGRAPH '07, ACM. 2

[MCTB11] MAULE M., COMBA J. L., TORCHELSEN R. P., BASTOS R.: A survey of raster-based transparency techniques. *Computers & Graphics 35*, 6 (2011), 1023 – 1034. 1, 2

[MP01] MARK W. R., PROUDFOOT K.: The f-buffer: a rasterization-order fifo buffer for multi-pass rendering. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (New York, NY, USA, 2001), HWWS '01, ACM, pp. 57–64. 2

[Pee08] PEEPER C.: Prefix sum pass to linearize a-buffer storage. *U.S. Patent*, 2008/0316214 (2008). 1, 2, 3

[PTO10] PATNEY A., TZENG S., OWENS J. D.: Fragment-parallel composite and filter. *Computer Graphics Forum 29*, 4 (2010), 1251–1258. 2

[Wit01] WITTENBRINK C. M.: R-buffer: a pointerless a-buffer hardware architecture. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (New York, NY, USA, 2001), HWWS '01, ACM, pp. 73–80. 2

[YHGT10] YANG J. C., HENSLEY J., GRÃIJN H., THIBIEROZ N.: Real-time concurrent linked list construction on the gpu. *Computer Graphics Forum 29*, 4 (2010), 1297–1304. 1, 3