T-Buffer: A Dense and Fast Storage for Rendering Order-Independent Transparency

Hui-Chen Lin Hui-Chin Yang Jyh-Jiun Shann Chung-Ping Chung
Institute of Computer Science and Engineering, National Chiao Tung University, Hsinchu, Taiwan
{huichen, huichin, jjshann, cpchung}@csie.nctu.edu.tw

ABSTRACT-*To render the transparent effect of a scene fast and in real time, hardware algorithm with additional transparent fragment storage for order-independent transparent fragments are required in graphics processing. As the scene complexity and demand for resolution are constantly increasing, the storage consumption and also its access overhead hurdle system efficiency. This paper proposes the T-buffer storage system design as a solution, which has the following design goals: To lower the demand for transparent fragment storage, we use memory only for pixel locations consisting of transparent fragments. All those fragments with the same pixel location are further grouped in memory for easy management and subsequent use. And to retrieve transparent fragments in computation fast, indexing mechanism is designed in our storage system. We compare the proposed design with two famous works, the R-buffer and the M-buffer with WF hardware oriented algorithm, in terms of storage size and access counts. Experimental results show that T-buffer uses 29% less storage than R-buffer and 67% less than M-buffer. Nevertheless, T-buffer needs to be accessed 52% less frequent than R-buffer, although 27% more frequent than M-buffer.*

Keywords: Graphics Hardware, Rendering Hardware, Graphics Processing, Transparency Rendering

1. INTRODUCTION

Transparency effect in computer graphics has recently gained much attention, due to increasing rendering quality demands. The underlying technique is alpha blending: combining a translucent foreground with a background. And all translucent fragments at the same scene coordinate must be rendered in a strict depth order, the so-called order-dependent transparency.

Mammen describes a depth sorting algorithm based on the concept of Virtual Pixel Maps [1]; Snyder and Lengyel present an application algorithm to identify and sort the mutual-occluding parts of objects [2]. These sorting algorithms are very time-consuming. Therefore, order-independent transparency, rendering transparent objects without depth sorting, becomes an important issue for high performance rendering systems. There are several different kinds of order-independent transparency algorithms. Some are single-pass rendering algorithms that sample the alpha value and interpret it as how much it covers the pixel to produce dithering-like transparent effect in images [3] [4]. However, since they are probability-measure algorithms, they also have chance to produce artificial results. Other order-independent transparency algorithms use multi-pass rendering method that process transparent fragments several times to render them in correct depth order [5]. These multi-pass rendering algorithms are some kinds of fragment-level depth sorting technique; therefore, in general cases, they have the same defect as sorting algorithms have, that is, time consuming.

For the reason of solving time-consuming problem, most order-independent transparency algorithms modified the traditional GPU architecture to achieve fast rendering. Z^3 hardware technique is one of these modified hardware architecture which only renders a fixed number of transparency layers correctly [6]. R-buffer (RB) architecture [7] implemented A-buffer software algorithm [8] into hardware by adding an extra storage system to store transparent fragments in their arrival order. WF (Weight Factor) hardware oriented algorithm [9] precomputes the contribution factor of each fragment to the final color of a pixel and sequentially stores transparent fragments based on their x-y coordinate into an additional organized storage system. As the scene complexity arises, the number of transparent fragments and the storage space for transparent fragments increase significantly. How to store these transparent fragments for lowering the demand for memory

becomes more and more important. Furthermore, current fragment storage supporting techniques for order-independent transparency [9] [7] still have some defects to be improved such as large storage requirement and high execution time during rendering.

Our objective is to design a dense and extensible transparent fragment storage system which places order-independent-arrival transparent fragments in an organized way. Desired features in this design category include:

- How to efficiently store all transparent fragments, using as less storage as possible
- How to efficiently retrieve in this storage all those fragments mapped to the same screen coordinates
- How to avoid storage overflow, such that both fragment storage and retrieve do not have to suffer from lengthened outside memory latency

2. BACKGROUND

2.1 Transparency and Alpha Blending

Translucent objects can be rendered by specifying the degree of transparency with a color. The value to represent the degree of transparency is defined as an **alpha (α) value**, which ranges from 0.0 (completely transparent) to 1.0 (completely opaque). Each fragment has its alpha value with its RGB color components. To obtain the final color of a pixel, the translucent fragments belonging to the pixel (i.e., fragments have the same x-y coordinate) are typically assumed to be rendered from back to front in visibility order, or depth order. Normally, the alpha blending equation (1) [10] is used, as shown below:

$$c = \alpha_f c_f + (1-\alpha_f)c_b \qquad \text{Eq. (1)}$$

where c is the final color of a pixel, c_f and α_f are the color and the alpha value of foreground transparent fragment, and c_b is the color of background fragment.

2.2 Transparency Rendering Problem

The blending equation (1) is order-dependent, which means that transparent fragments require to be processed in their depth order, not in their arrival order. Thus, if we render transparent fragments in arbitrary order, it will produce an artificial result. For example, in Figure 1, fragment T_4 and T_6 come before fragment T_7, if we blend T_4 and T_6 with opaque fragment O_3 first, the blend T_7 later, according to Eq. (1), the final color c will be

$$\begin{aligned} c &= \alpha_7 c_7 + (1-\alpha_7)\{\alpha_6 c_6 + (1-\alpha_6)[\alpha_4 c_4 + (1-\alpha_4)c_3]\} \\ &= \alpha_7 c_7 + (1-\alpha_7)\alpha_6 c_6 + (1-\alpha_7)(1-\alpha_6)\alpha_4 c_4 + \\ &\quad (1-\alpha_7)(1-\alpha_6)(1-\alpha_4)c_3 \end{aligned}$$

But the correct final color should be

$$\begin{aligned} c &= \alpha_6 c_6 + (1-\alpha_6)\{\alpha_4 c_4 + (1-\alpha_4)[\alpha_7 c_7 + (1-\alpha_7)c_3]\} \\ &= \alpha_6 c_6 + (1-\alpha_6)\alpha_4 c_4 + (1-\alpha_6)(1-\alpha_4)\alpha_7 c_7 + \\ &\quad (1-\alpha_6)(1-\alpha_4)(1-\alpha_7)c_3 \end{aligned}$$

Thus, the incorrect result is produced due to the incorrect rendering order.

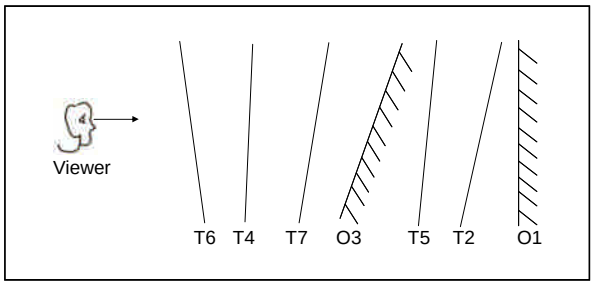


Figure1. Example of fragment blending processing

Since fragments are generated in arbitrary order at rasterization, not in depth order, several algorithms are proposed for correct transparent rendering. These algorithms can be classified as sorting based algorithms and order-independent transparency algorithms. Sorting based algorithms require the primitives (polygons) to be sorted from back to front with respect to the viewpoint. These sorting algorithms can further be classified into application sorting [1][2], hardware assistant sorting [5], and hardware sorting [9][11] algorithms based on the method they use to sort the primitives. However, for application sorting algorithms, it is difficult to do depth sorting since objects in a scene may intersect each other and intersected parts need to be divided into several polygons. Even for those hardware assistant sorting algorithms, it is very time-consuming. Therefore, it comes out order-independent transparency.

2.3 Order-independent transparency

Since our research focuses on hardware storage support for order-independent transparency, we will introduce more details of R-buffer hardware architecture [7] and WF hardware oriented algorithm [9], which are more related to our system design.

2.3.1 R-buffer hardware architecture

The R-buffer is a FIFO (first-in-first-out) memory which stores transparent fragments in the sequence that they arrive.

The information of each transparent fragment —the location (x, y), the depth value (z), the color value (RGB) with alpha value(A or α)— needs to be stored in the R-buffer. Pixel state memory stores each pixel's current state. The second z-buffer stores the depth value of the furthest visible transparent fragments per pixel. The memory size of the R-buffer is proportional to the number of transparent fragments after early z test. The memory size of the second z-buffer is equivalent to the original z-buffer. In pixel state memory, each pixel needs three bits to record its current value; thus, the memory size of the pixel state memory is equal to three multiplied by the screen size. To sum up the memory requirement of R-buffer architecture, we list the R-buffer memory requirement equation as follow:

$$Memory_{total} = M_{R\text{-}buffer} + M_{2nd\text{-}z\text{-}buffer} + M_{state\text{-}memory}$$

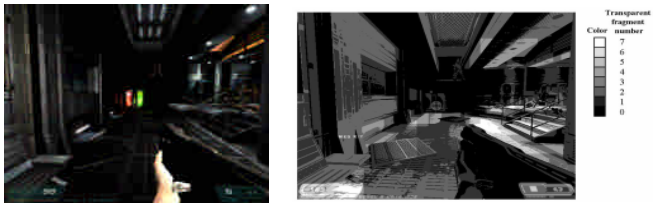
2.3.2 Hardware oriented algorithm based on weight factors computations

The organized memory scheme of WF algorithm suggests that transparent fragments belonging to the same pixel are stored sequentially and connectedly in the M-buffer. M-buffer is organized in sections of D_{avg} words, where D_{avg} is the average number of fragments per pixel. Each pixel has it corresponding storage section, with capacity for D_{avg} fragments; that is, for a system with $W \times H$ pixels, $W \times H$ sections would be required and a pixel i in a system has a corresponding section i in M-buffer. To extend the storage capabilities, a pointer memory is added so that more than one section can be dynamically assigned to a given pixel. The information stored per section of a pointer memory indicates that whether one section is sufficient (by storing a NULL pointer) or whether the following-coming fragments are stored in another section (by storing the section index). For example, if there are F transparent fragments belonging to a pixel i, where F is larger than D_{avg}, the first D_{avg} fragments are stored in section i of M-buffer, and the following F- D_{avg} fragments are stored in another section j ($j \geqq W \times H$). The section i of a pointer memory stores the section j index. If section j is still insufficient to store F- D_{avg} fragments (i.e., F-$D_{avg} > D_{avg}$), the rest F-$2xD_{avg}$ fragments are stored to another section k ($k>j$), and so on.

3. DESIGN

3.1 Statistics and Observation

Figure 2(a) shows a frame image in DOOM3. We analysis the number of transparent fragments of each pixel in this frame and obtain the result in Figure 2(b). In the gray-level image, the black color indicates that the number of transparent fragments of a pixel is 0, while white color indicates that the number of transparent fragments of a pixel is 7, that is, the maximum number of transparent fragments of a pixel in the frame. The transparent fragment numbers of a pixel between 0 and 7 and their corresponding colors are shown at right side of Figure 2(b). We find that not all pixels in a frame have transparent fragments. However, in M-buffer with WF proposal, each pixel is assigned the same size of memory space, no matter whether the pixel has transparent fragments or not. Therefore, if we use a transparent storage support proposed in M-buffer, it needs a large memory space and parts of them are unused resulting in unnecessary memory cost.



(a) (b)

Figure 2. (a) A test frame; (b) Number of transparent fragments per pixel expressed by

3.2 T-buffer Storage System

The objective of our transparent fragment storage system is to reduce the memory requirement and memory access of transparent fragments for order-independent transparency. The overview of our proposed transparent fragment storage system is shown in Figure 3.

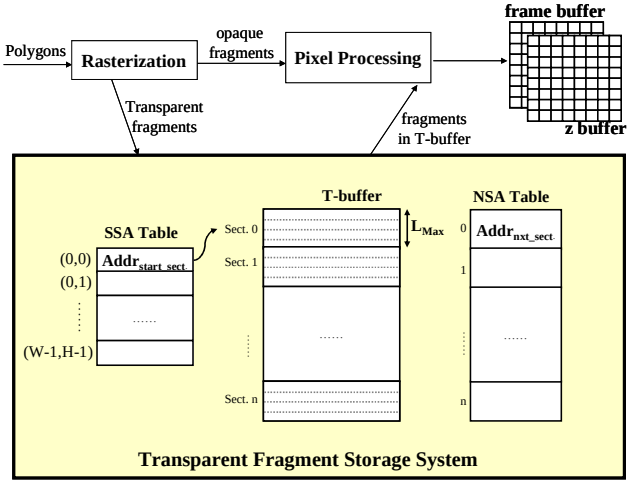


Figure 3. Design diagram of T-buffer and

After rasterization stage, the polygons are segmented into several fragments in arbitrary order. Then, opaque fragments continue the pixel processing procedure while transparent fragments are stored into our transparent fragment storage system. The storage scheme is to store transparent fragments in an organized way based on their (x, y) coordinate. Transparent fragments belonging to a same pixel location are stored together.

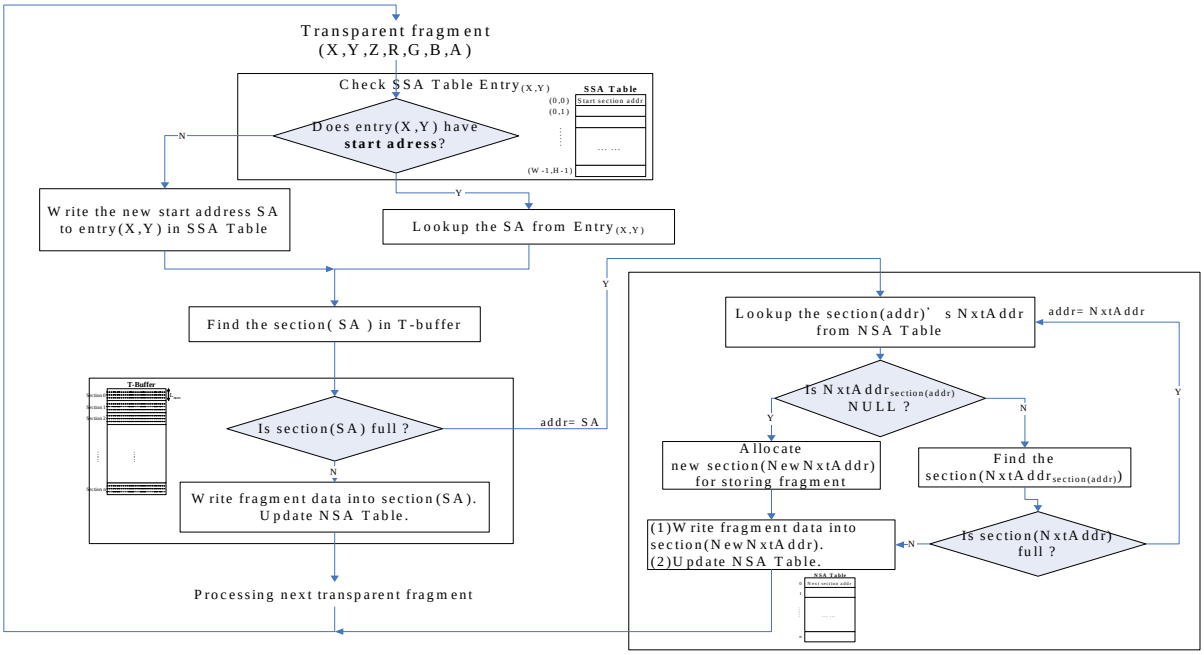


Figure 4. Flowchart of the process for storing fragments into T-buffer storage system

3.2.1 SSA(Start-Section Address) Table

Start-section address table (SSA Table) has W×H entries, where W×H is defined as the display resolution. Each pixel p has a corresponding entry e_p in SSA Table that stores the address of start section within T-buffer for pixel p. If a pixel does not have any transparent fragments, a nullified address is stored in the corresponding entry in SSA Table. T-buffer is a storage space for transparent fragments.

3.2.2 T-buffer

T-buffer is organized in sections and each section has the capability for storing L_{Max} transparent fragments. Fragments belonging to the same pixel location are grouped together and stored within one section in T-buffer. There might be more than L_{Max} transparent fragments belonging to the same pixel; thus, more than one section should be assigned to a pixel to extend the capability for storing variable number of fragments.

3.2.3 NSA(Next-Section Address) Table

If the number of transparent fragments is more than the number that one section is capable of storing, the excess fragments will be stored in another section. Then, the address of the section is recorded as the next-section address in NSA Table. The number of entries in NSA Table is equivalent to the number of sections in T-buffer and there is a one-to-one correspondence between entries in NSA table and sections in T-buffer. However, if one section is sufficient to store transparent fragments of a pixel, there is no need to assign another section, and thus, the NULL pointer is stored instead of the section address.

3.3 The Access Process of T-buffer Storage System

Figure 4 shows the process of storing fragments into T-buffer. Before a transparent fragment with the location (x_i,y_j) is stored into T-buffer, the address of start section $s_{(i,j)}$ is read from the corresponding entry $e_{(i,j)}$ of SSA table. If there is no start section for a given pixel, the new empty section in T-buffer is assigned for that pixel, and the address of the new section is recorded in SSA table as the start-section address. When the section $s_{(i,j)}$ of T-buffer is available, the fragment is stored into the section $s_{(i,j)}$; if the section $s_{(i,j)}$ is full, the address of next section n_s is read from the entry e_s of NSA table, and the fragment is eventually stored into the section n_s of T-buffer.

The process for reading transparent fragments from T-buffer is shown is Figure 5. For each pixel $p_{(i,j)}$, the start-section address is read from its corresponding entry $e_{(i,j)}$ of SSA Table. If there is no start-section address for a pixel, the process is continued to read the start-section address for the next pixel. Otherwise, transparent fragments belonging to $p_{(i,j)}$ are accessed from the start section $s_{(i,j)}$ of T-buffer and the address of next section n_s can be read simultaneously from the entry e_s of NSA Table. Fragments are accessed from the next section n_s of T-buffer until there is no more next sections for pixel $p_{(i,j)}$.

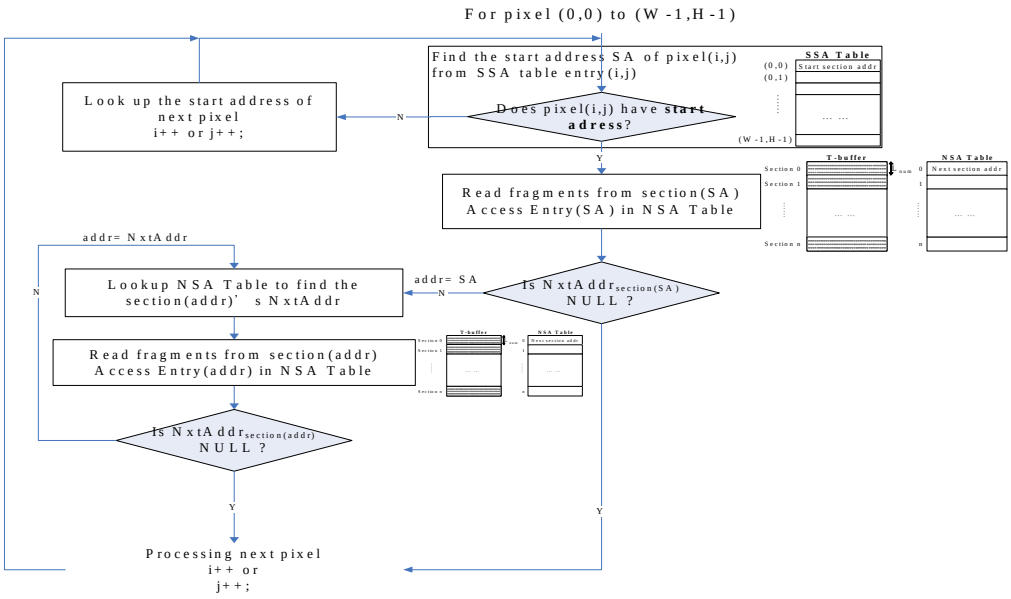


Figure 5. Flowchart of the process for reading fragments from T-buffer storage system

4. EVALUATION RESULTS

We implemented a behavioral simulator of the architecture with the T-buffer and the storage system in C++, and modified ATTILA simulator [11] to output fragment information to a tracefile. The benchmark used in ATTILA simulator is QUAKE4 [12], a modern graphics application. The tracefile outputted from ATTILA simulator contains the coordinates and RGBA color components of fragments in frames. Our simulator reads the tracefile and evaluates the memory requirement and access frequency time of transparent fragment storage system.

Table 1. Statistics of test frames

Frame N	No. of fragments	No. of transparent fragments	No. of transparent fragments per transparent pixel	Transparency density	No. of pixels whose no. of transparent fragments = N				
					1	2	3	4	5
Frame60	2,377,518	37,684	2.37	5%	5,812	956	6,633	2,279	189
Frame120	746,943	37,594	2.38	5%	5,723	956	6,634	2,278	189
Frame180	516,464	40,584	2.16	6%	8,713	956	6,634	2,278	189
Frame240	488,112	88,434	1.63	18%	32,588	12,040	7,235	2,279	189
Frame300	623,531	60,769	1.61	12%	26,082	2,729	6,636	2,179	121
Frame360	587,617	69,768	1.61	14%	27,421	7,094	7,006	1,569	173
Frame420	530,258	47,877	1.77	9%	15,728	3,290	6,770	1,296	15
Frame480	605,927	135,475	1.22	36%	95,226	8,716	6,711	671	0

We provided statistics of eight test frames, which were dumped at the interval of 60 frames, as shown in Table 1. In Table 1, the second column indicates the number of all fragments in each frame; the third column shows the number of transparent fragments in each frame; the fourth column shows the number of transparent fragments per transparent pixel; the fifth column shows the transparency density, which is defined as the percentage of transparent pixels in a frame (i.e. #transparent pixel/display resolution); the last column shows the distribution of transparent fragment layers per pixel in each frame.

Table 2. Memory requirements of each of transparent storage systems

techniques	Transparent fragment storage system			
	transparent fragment memory		other storage supports	
	Name	size	Name	Size
R-buffer	R-buffer	$N_f \times S_f$	2nd Z buffer	= Z buffer size ($W \times H \times Z$'s bytes)
			state memory	3(bits) $\times W \times H$
WF algorithm	M-buffer	$(W \times H + N_A) \times M_{WF}$	pointer memory	$S_A \times (W \times H + N_A)$
T-Buffer	T-buffer	$M_{TFSS} \times N_S$	SSA Table	$S_A \times W \times H$
			NSA Table	$S_A \times N_S$

In order to evaluate T-buffer storage system, we compare our design with two related works: R-buffer [7] and WF hardware oriented algorithm [9]. We list the memory requirements of each method in Table 2. The second column shows the name and the size of a transparent fragment storage space in each technique and the second column shows the name and the size of other storage supports in each technique. In Table 2, N_f is defined as the number of transparent fragments, N_A is defined as the number of dynamic allocated sections in WF algorithm, and N_S is defined as the number of sections of T-buffer; S_f is defined as the size of a fragment data (XYZ,RGBA), S_A is defined as the size of an address; M_{WF} is defined as the memory size per section within M-buffer in WF algorithm, and $M_{T\text{-}buffer}$ is defined as the memory size per section within T-buffer; $W \times H$ are defined as the display resolution. The simulation result of memory requirement of each technique is shown in Figure 6.

We find that our T-buffer has the lowest memory requirement than two related works in average.

Notice that in frame480, the memory requirements of T-buffer with L_{Max}=3 are larger than those of R-buffer technique. This result is occurred by the weakly utilization of a section, which has the capability for storing three transparent fragments but only stores fewer than three fragments. In frame480, the number of transparent fragments per transparent pixel is 1.22, which is lower than that in other frames, and about 34% of sections in T-buffer are weakly utilized. Therefore, in frame480, it results in unnecessary memory cost and larger memory requirement of T-buffer with L_{Max}=3 than of R-buffer.

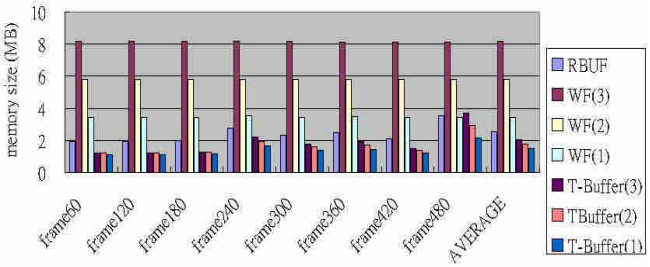


Figure 6. Memory requirement comparison; WF(num) and T-Buffer(num) : the number in parens. is defined as the parameter L_{Max}

Furthermore, we analyzed the memory access frequency for order-independent transparency of each technique. Suppose N_i pixels have i transparent fragments, where i =0,1,2,3,…..n_{max}, n_{max} is the maximum number of transparent fragments that a pixel have in the frame. It infers that the number of pixels which have at least one transparent fragment is equal to $\sum_{i=1}^{n_{max}} N_i$, and the total number of transparent fragments is equal to $\sum_{i=1}^{n_{max}} (N_i \times i)$.

In R-buffer architecture, $\sum_{i=1}^{n_{max}} (N_i \times i)$ transparent fragments are stored in R-buffer. In the first pass of reading R-buffer, there are $\sum_{i=1}^{n_{max}} (N_i \times i)$ transparent fragments to be accessed and $\sum_{i=1}^{n_{max}} N_i$ transparent fragments are removed from R-buffer. In the second pass, $\sum_{i=1}^{n_{max}} N_i \times i - \sum_{i=1}^{n_{max}} N_i$ transparent fragments need to be accessed and $\sum_{i=2}^{n_{max}} N_i$ fragments are removed from R-buffer. In the p_{th} pass, $\sum_{i=1}^{n_{max}} N_i \times i - \sum_{j=1}^{p-1} \sum_{i=j}^{n_{max}} N_i$ need to be accessed and $\sum_{i=p}^{n_{max}} N_i$ fragments are removed from R-buffer. Thus, the memory access frequency of R-buffer is equal to $f_{R\text{-}buffer}$ =

$$\sum_{i=1}^{n_{max}} N_i \times i + (\sum_{i=1}^{n_{max}} N_i \times i - \sum_{i=1}^{n_{max}} N_i) + (\sum_{i=1}^{n_{max}} N_i \times i - \sum_{i=1}^{n_{max}} N_i - \sum_{i=2}^{n_{max}} N_i) + ... + (\sum_{i=1}^{n_{max}} N_i \times i - \sum_{j=1}^{n_{max}} \sum_{i=j}^{n_{max}} N_i)$$

In addition, after a fragment accessed from R-buffer, its depth value needs to be compared with the depth value stored in second z-buffer, and thus, the access frequency of second z-buffer, denoted as $f_{2nd\text{-}zbuffer}$, is equal to that of R-buffer. Moreover, each transparent fragment is eventually blended with the background fragment stored in frame buffer, and thus, the access frequency of frame buffer, denoted as $f_{frame\text{-}buffer}$, is equal to $\sum_{i=1}^{n_{max}} (N_i \times i)$. The total memory access frequency of storage system in R-buffer architecture is equal to $f_{R\text{-}buffer}$ + $f_{2nd\text{-}zbuffer}$+ $f_{frame\text{-}buffer}$. It implies that the more transparent fragments in a frame or the more transparent fragments per pixel, the more memory access frequency of R-buffer architecture.

In WF algorithms, assume that the pointer memory can be accessed simultaneously with M-buffer, and does not wait for being accessed after M-buffer access. Therefore, the access frequency of the pointer memory will not affect the time requirement of memory access and can be ignored. Since transparent fragments with the same x-y coordinate are stored sequentially and connectedly, we can access these transparent fragments at one pass from M-buffer and blend these fragments based on weight factor computation proposed in [9]. Thus, the access frequency of M-buffer is equal to $\sum_{i=1}^{n_{max}} N_i \times i$.

In our design, the access frequency of T-buffer is the same as the access frequency of M-buffer in WF algorithms. However, in our approach, SSA Table needs to be accessed before accessing T-buffer in order to find the start section in T-buffer. Thus, SSA Table cannot be accessed simultaneously with T-buffer and the access frequency of SSA Table should be considered. Figure 7 shows the memory access frequency of transparent storage system of each technique. As we expect, the memory access frequency of R-buffer proposal is higher than WF proposal and our approach; WF proposal has the lowest access frequency; the access frequency of our approach is just a little higher than WF proposal and much lower than R-buffer proposal.

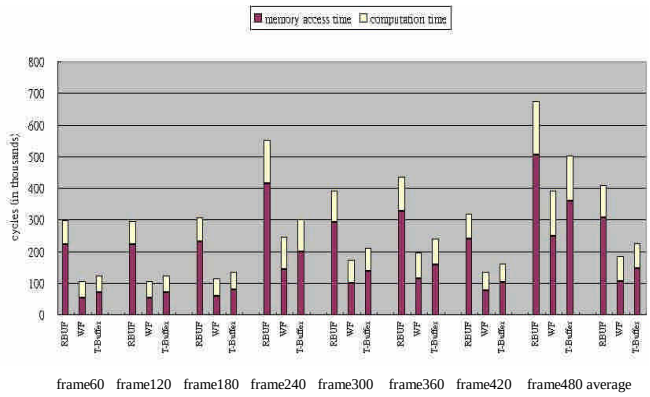


Figure 7. Memory access frequency

5. CONCLUSIONS

In this paper, we proposed a dense and flexible T-buffer storage system for order-independent transparency. For QUAKE4 benchmark, our transparent fragment storage system, in comparison with R-buffer architecture, reduces 29% memory requirement in average, and reduces 52% memory access frequency. Even ignoring long external memory latency when overflow occurs, in comparison with M-buffer, although the memory access frequency of our storage system is 27% higher than M-buffer, it reduces 67% memory requirement in average.

T-buffer design has the following advantages over the M-buffer:

- Sections in T-buffer are allocated to screen coordinates flexibly. Any section can be freely allocated to any screen (x,y), with the help of the SSA table. This reduces the memory pressure for transparent fragment storage.
- Furthermore, sections in T-buffer can be allocated to other overflow sections, with the help of the NSA table. This overflow handling can recursively go on, until all T-buffer sections are used up. Transparent fragment overflow to outside memory can hence be reduced to the minimum.
- With above two advantages, storage efficiency of the T-buffer is expected to be very high. So a much smaller T-buffer may perform comparably with a full-size M-buffer.
- Better yet, T-buffer size (both section count and section capacity are design parameters) can be trimmed to best fit any given application domain.

REFERENCES

[1] A. Mammen. Transparency and Antialiasing Algorithms Implemented with the Virtual Pixel Maps Technique. In IEEE Computer Graphics and Applications, Vol. 9, No. 4, pp. 43–55, 1989.

[2] John Snyder and Jed Lengyel. Visibility Sorting and Compositing without Splitting for Image Layer Decompositions. In Proceedings of the 25th annual conference on Computer graphics and interactive techniques, pp. 219–230, 1998.

[3] Peter L. Williams, Randall J. Frank, and Eric C. LaMar. Alpha Dithering. Lawrence Livermore National Laboratory, Livermore, CA.

[4] Jurriaan D. Mulder, Frans C. A. Groen, and Jarke J. van Wijk. Pixel masks for screen-door transparency. In Proceedings of the conference on Visualization '98, pp. 351–358. IEEE Computer Society Press, 1998.

[5] Cass Everitt. Interactive Order-Independent Transparency. In Technical report, NVIDIA Corporation. ACM Press, 2001.

[6] N. P. Jouppi and C.-F. Chang. Z^3 : an Economical hardware Technique Rendering for High-quality Antialising and Transparency. In Proceesings of Graphics Hardware, pp. 85-93, ACM/Eurographics, 1999.

[7] Craig M. Wittenbrink. R-buffer: a Pointerless A-buffer Hardware Architecture. In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, pages 73–80. ACM Press, 2001.

[8] L. Carpenter. The A-buffer, an Antialiased Hidden Surface Method. In Proceedings of ACM SIGGRAPH, pp.103-108, 1984.

[9] M. Amor, M. Boo, E.J. Padron and D. Bartz. Hardware Oriented Algorithms for Rendering Order-Independent Transparency. The Computer Journal, vol. 49, issue 2, 2006.

[10] T. Porter and T. Duff. Compositing Digital Images. ACM Computer Graphics (Siggraph 84 Proc.), pp. 253–259, 1984.

[11] Víctor Moya, Carlos González, Jordi Roca, Agustín Fernández and Roger Espasa. ATTILA: A Cycle-Level Execution-Driven Simulator for Modern GPU Architectures. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2006), March 2006

[12] http://www.quake4game.com/

[13] Stephanie Winner, Mike Kelley, Brent Pease, Bill, Rivard, and Alex Yen. Hardware Accelerated Rendering of Antialiasing Using a Modified A-buffer algorithm. In Proceedings of the 24th annual conference on Computer graphics and interactive techniques, pp. 307–316, 1997