國 立 交 通 大 學

資訊科學與工程研究所

碩 士 論 文

H 緩衝暫存器:利用歷史記錄及溢出分享之有效率透明畫格儲存方法

H-Buffer: An Efficient History-Based and Overflow Sharing Transparent Fragment Storage Method

研 究 生：呂 東 霖

指導教授：陳 登 吉 教授

中 華 民 國 九 十 八 年 九 月

H 緩衝暫存器:利用歷史記錄及溢出分享之有效率透明畫格儲存方法

H-Buffer: An Efficient History-Based and Overflow Sharing Transparent Fragment Storage Method

研 究 生：呂東霖	Student：Tung-Lin Lu
指導教授：陳登吉	Advisor：Deng-Jyi Chen

國 立 交 通 大 學
資 訊 科 學 與 工 程 研 究 所
碩 士 論 文

A Thesis
Submitted to Institute of Computer Science and Engineering
College of Computer Science
National Chiao Tung University
in Partial Fulfillment of the Requirements
for the Degree of
Master
In

Computer Science

June 2009

Hsinchu, Taiwan, Republic of China

中華民國 九 十 八 年 九 月

H 緩衝暫存器:利用歷史記錄及溢出分享之有效率透明畫格儲存方法

學生：呂東霖　　　　　　　　　　　　　　　　　　　　指導教授：陳登吉 教授

國立交通大學資訊科學與工程研究所 碩士班

摘　要

圖形繪製往往需要大量的資料暫存空間，也因此限制了在以輕薄為特色的嵌入式裝置上之發展。為了解決此問題，此發明著重在減少暫存透明網格的空間需求。此設計是建構在以下的概念：除了極少數場景在做切換的畫面，決大多數的畫面其每一個畫面座標所擁有的透明網格數量都相當接近。因此我們提出了一個歷史根基透明網格暫存緩衝器－H暫存緩衝器。由於透明網格以任意順序進入H暫存緩衝器做儲存，因此主要面對的挑戰為如何減少其記憶體內部碎裂，並有效率儲存透明網格。在此設計中，我們會去記錄每一張畫面座標上面的透明網格數量，並且用來做為下一張畫面在繪圖時，使用透明網格暫存緩衝器空間大小的依據。一旦遇到透明網格暫存緩衝器空間不夠，我們會將透明網格存放到溢出空間，並且讓多個畫面座標可以共用此空間以減少因內部碎裂所造成的空間浪費。我們測試了連續５００張 QUAKE4 及 DOOM3 畫面，並且與目前所需空間最小的T暫存緩衝器做比較。在QUAKE4中，我們的設計比T暫存緩衝器少了約25％的透明網格儲存空間需求，而在DOOM3中，少了約20％的透明網格儲存空間需求。

H-Buffer: An Efficient History-Based and Overflow Sharing Transparent Fragment Storage Method

Student：Tung-Lin Lu Advisor：Deng-Jyi Chen

Institute of Computer Science and Engineering
National Chiao-Tung University

Abstract

Graphics rendering requires various huge amounts of temporary data storages, prohibiting this feature from being implemented on slim embedded devices. To overcome this difficulty, we focus our effort on storage of transparent fragments after rasterization stage. We base our design on the fact that: successive frames typically will have the same or very similar number of transparent fragments located at the same screen pixel location, except in the rare case of scene change. We propose a history based transparent fragment buffer called H-Buffer. Note that transparent fragments arrive in any arbitrary order, making the design challenging. And the storage pressure comes from necessary storage plus internal fragmentation, the latter being resulted from fix-size storage allocation and can be reduced. In this design, transparent fragment counts at all pixel locations are collected for every frame, and be used for storage allocation for the next frame. For the unavoidable case of insufficient storage allocation, our overflow storage allocation assigns neighbor pixel locations to share a given overflow area, in an attempt to reduce internal fragmentation. Easy management and quick access are two major concerns in our design. In evaluation, we used 500 frames from QUAKE4 and DOOM3. Storage requirements are compared against the W-buffer, and the T buffer methods. Compared with the strongest competitor, the T-buffer, results show that our method reduces storage pressure by 25% in QUAKE4 benchmark, and

20% in DOOM3 benchmark. This design idea can be extended to applications where the load change is typically mild and only occasionally abrupt.

誌謝

這兩年來的碩士研究生涯中，首先，我要感謝我的指導教授 鍾崇斌老師。感謝他嚴謹與耐心的指導，使我在我的碩士研究中，獲得許多寶貴的建議與方向，並且學習到如何以不同的角度去看待研究事物，並對各種可能性加以討論並提出質疑，而得以完成此碩士論文及順利通過畢業口試。此外，感謝我的畢業口試委員 單智君教授、邱日清教授、洪士灝教授，由於他們的建議，使得此論文研究更加完整。

同時，我也要感謝實驗室的學長姐、同學及學弟們在各方面，給我很大的幫助。特別感謝惠親學姐辛苦地帶領與指導、喬偉豪學長及翁綜禧學長的建議與幫忙，以及;感謝同樣在GPU計劃中的秀青、之傑及浤偉彼此的討論、相互打氣與支持。沒有你們的幫助，我也無法如此順利地完成此碩士論文。還有實驗室的學弟CPR跟阿Sa，如果實驗室少了你們，我們每天的歡笑也會少了很多。

此外，感謝我的家人在背後默默的支持我，即使你們對於研究上沒辦法給予我太多意見，但有你們在一旁關心我，也讓我能更堅定與堅毅地繼續我的研究之路。

呂東霖 2009.9

Contents

List of Figures

List of Tables

Chapter 1 Introduction

In recently years, a special purpose processor, **Graphics Processing Unit (GPU),** is used to render 3D computer graphics. Due to increase demand of high realistically rendering quality, the support for transparent effect becomes more important for visual reality. Transparency effect operation is implemented in GPU which called **alpha blending:** combining a translucent foreground with a background. In the process of alpha blending, all transparent fragments at the same screen coordinate must be rendered in correct depth order (from back to front with respect to the viewpoint). However, fragment stream after Z-test process is in arbitrary order. For this reason, the transparent objects are sorted to get the correct depth order in the application level in tradition. However, it is difficult for software application developers to sort transparent objects since objects may intersect each other. In addition, as the number of transparent primitives increases rapidly, the application sorting becomes more and more complicated. For fast rendering, the recent research is focus on order-independent transparency rendering algorithm and implemented on the GPU architecture with additional hardware support for temporary storing transparent fragments.

The earliest hardware oriented order-independent transparency rendering design is Z^3 hardware technique [Joup99], but it only renders a fixed number of transparency layers correctly. The first hardware rendering full transparency layers design is R-buffer. R-buffer [Witt01] implements A-buffer software algorithm [Carp84] into hardware by adding an extra storage system to stores transparent fragments in their arriving order. After all the fragments of the frame past Z-test, the alpha blending process will render the entire scene by iteratively blending furthest visible transparent fragment with background until the R-buffer is empty. Another hardware oriented order-independent transparency rendering design is M-buffer [Amor06] which based on WF (Weight Factor) algorithm. Every pixel stores transparent fragments into their corresponding section in M-buffer according to their screen coordinate. T-buffer transparent fragment storage system [Lin08] is another hardware oriented design based on the WF algorithm. T-buffer improves the utilization of transparent fragment storage of M-buffer.

Once the scene complexity arises, transparent fragment amount and the storage space for transparent fragments increase significantly. For this reason, how to minimum the demand for memory becomes important. In addition, the past design of transparent fragment storage is not efficiency for storing transparent fragments. T-buffer, the highest utilization of transparent fragment storage still have nearly thirty percent memory which be wasted by internal fragmentation in QUAKE4 benchmark. Our objective is to design a flexible and economic transparent fragment storage system based on WF algorithm which can provide the memory size of transparent fragment storage system which needs to be least.

Chapter 2 Background and Related work

In this chapter, we will give an overview of graphics pipeline. Then, we will introduce the definition of transparency, alpha blending operation, explain the transparency rendering problem, and expatiate on order-independent transparency. At the end of this chapter, we will present the details of three previous works related to hardware support techniques for order-independent transparency.

2.1 Graphics pipeline

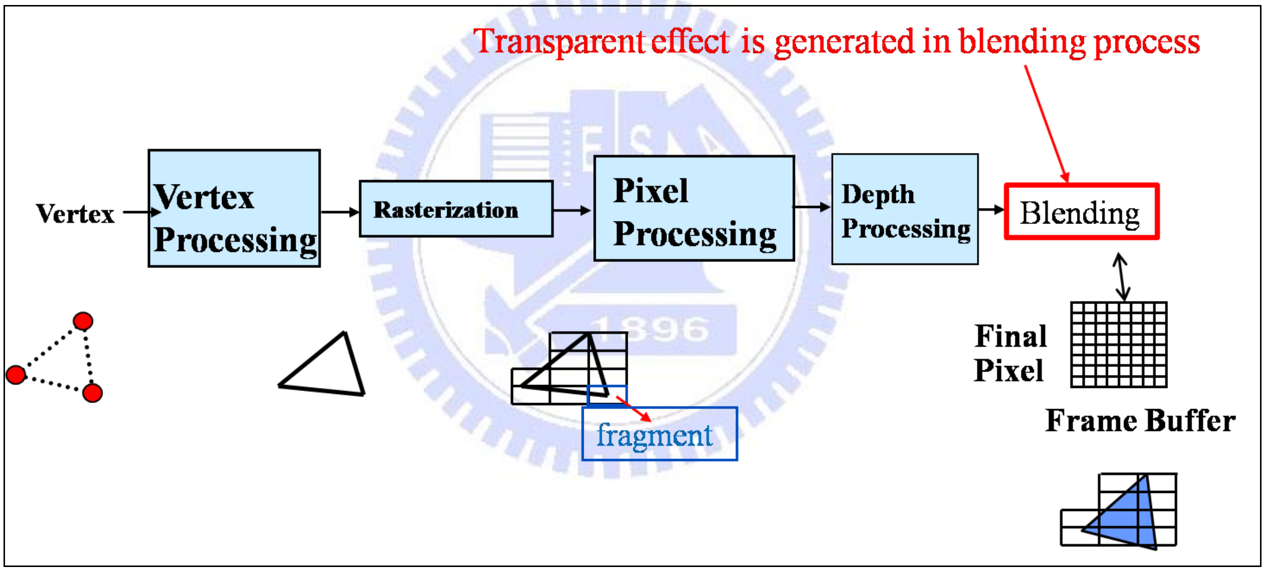


Figure 2-1 3D graphics pipeline

Graphics pipeline can be roughly divided into five stages: **vertex processing**, **rasterization**, **pixel (fragment) processing**, **depth processing**, and **blending processing**, as shown in Figure 2-1. At vertex processing stage, coordinate transformation, lighting, assemble vertex into primitive, clipping, culling operations will be processed, and output primitive stream. After vertex processing stage, these primitives are sent into rasterization stage. Rasterization stage is to determine which squares of an integer grid in screen coordinate are occupied by the triangle and to assign a color and a depth value to each such square. Such generated image square is called **fragment.** Fragments are then sent into pixel processing stage. At pixel processing stage, the interpolation and texture mapping process generate fragment color [Watt00]. And the next stage, depth process, will cut those fragments which occluded by other fragment. Until all the fragments are pass the depth processing stage, the blending stage will process. Some backend operation like alpha blending, anti-Aliasing, fog will process at this stage.

Noticed that transparency effect is generated in blending stage, and our system is designed for storing transparent fragments, which pass the depth processing stage; therefore, we are only concerned about the process between depth processing stage and blending stage in graphics pipeline in this thesis.

2.2 Transparency and alpha blending

All the fragments have alpha (a) value attribute to represent the degree of transparency, which range is from 0.0 to 1.0. The alpha value 0.0 represent the completely transparent, and 1.0 represent the completely opaque. To obtain the final color of a pixel, the transparent fragments belonging to the pixel (i.e., transparent fragments have the same x-y coordinate) are typically assumed to be rendered from back to front in visibility order, or depth order. The process of blending a translucent foreground with a background color to generate the effect of transparency is called **alpha blending**. The alpha blending equation [Watt84] is used for alpha blending, 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.

To clarify, consider an example of fragment blending processing shown in Figure 2-2. There are two fragments at the same screen coordinate. The front of fragment is a transparent fragment, and the back of fragment is a opaque fragment. Assume each color attributes (R,G,B,A) are (1,0,0,0.5) and(1,1,0,1.0).When alpha blending process, the front fragment will blending with the background fragment. According to Eq.(1), the final color *c* of the pixel is equal to:

C(R,G,B)=0.5x(1,0,0)+(1-0.5)(1,1,0)=(1,0.5,0).

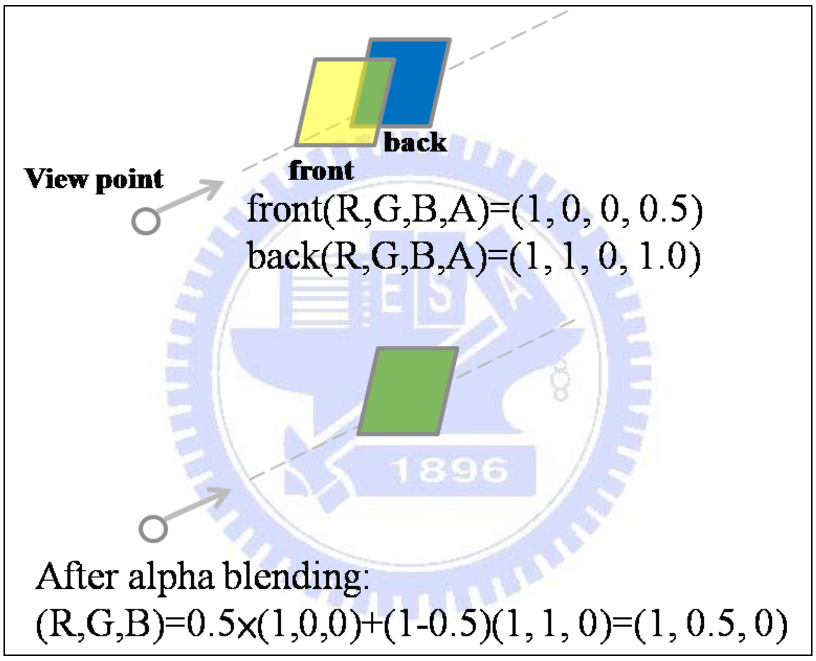

back
front
View point
front(R,G,B,A)=(1, 0, 0, 0.5)
back(R,G,B,A)=(1, 1, 0, 1.0)
After alpha blending:
(R,G,B)=0.5x(1,0,0)+(1-0.5)(1, 1, 0)=(1, 0.5, 0)

Figure 2-2 Example of fragment blending processing

2.3 Transparency rendering problem

The blending equation Eq.(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. Figure 2-3 is an example of processing alpha blending in depth order and arbitrary order. There are two transparent fragments and one opaque fragment at the same screen coordinate. We can found that if we did not blend transparent fragment from back to front in depth order, the final pixel color may different.

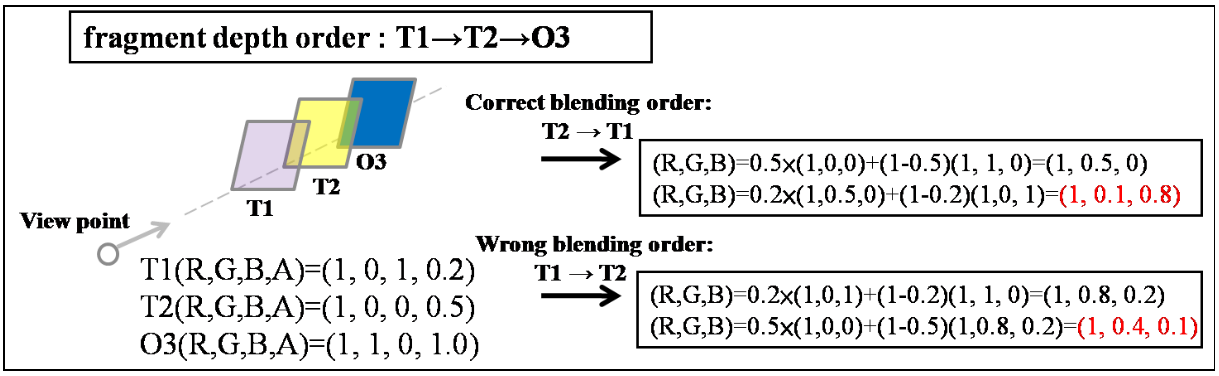

fragment depth order : T1→T2→O3
View point
T1
T2
O3
T1(R,G,B,A)=(1, 0, 1, 0.2)
T2(R,G,B,A)=(1, 0, 0, 0.5)
O3(R,G,B,A)=(1, 1, 0, 1.0)
Correct blending order:
T2 → T1
(R,G,B)=0.5×(1,0,0)+(1-0.5)(1, 1, 0)=(1, 0.5, 0)
(R,G,B)=0.2×(1,0.5,0)+(1-0.2)(1,0, 1)=(1, 0.1, 0.8)
Wrong blending order:
T1 → T2
(R,G,B)=0.2×(1,0,1)+(1-0.2)(1, 1, 0)=(1, 0.8, 0.2)
(R,G,B)=0.5×(1,0,0)+(1-0.5)(1,0.8, 0.2)=(1, 0.4, 0.1)

Figure 2-3 Example of processing alpha blending in depth order and arbitrary order.

However, fragments are generated in arbitrary order at rasterization, not in depth order. several algorithms [Mamm89][Snyd98][Ever01] 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. However, for application sorting algorithms, it is time-consuming for depth sorting since objects in a scene may intersect each other and intersected parts need to be divided into several polygons. Therefore, it comes out order-independent transparency.

2.4 Order-independent transparency

Order-independent transparency is defined as a process which renders transparent fragment in arbitrary order instead of sorting them in advance. There are several different kinds of order-independent transparency algorithms.

Most order-independent transparency algorithms modified the traditional GPU architecture to solve time-consuming problem. Z^3 hardware technique [Joup99] is one of these modified hardware architecture which only renders a fixed number of transparency layers correctly. R-buffer [Witt01] is a modified hardware architecture which implements A-buffer [Carp84] software algorithm into hardware by adding an extra storage system to store transparent fragments associated with each pixel. WF (Weight Factor) hardware oriented algorithms [Amor06] precomputes the contribution factor of each fragment to the final color of pixel and propose an organized strategy to sequentially store transparent fragments corresponding to the same pixel. T-buffer transparent fragment storage system [Lin08] has modified the WF hardware oriented algorithm and improve the utilization of transparent fragment storage. Since our research focuses on hardware storage support for order-independent transparency, we will introduce more details of R-buffer hardware architecture, WF hardware oriented algorithm, and T-buffer hardware architecture which are more related to our system design.

2.5 Related works

2.5.1 R-Buffer hardware architecture

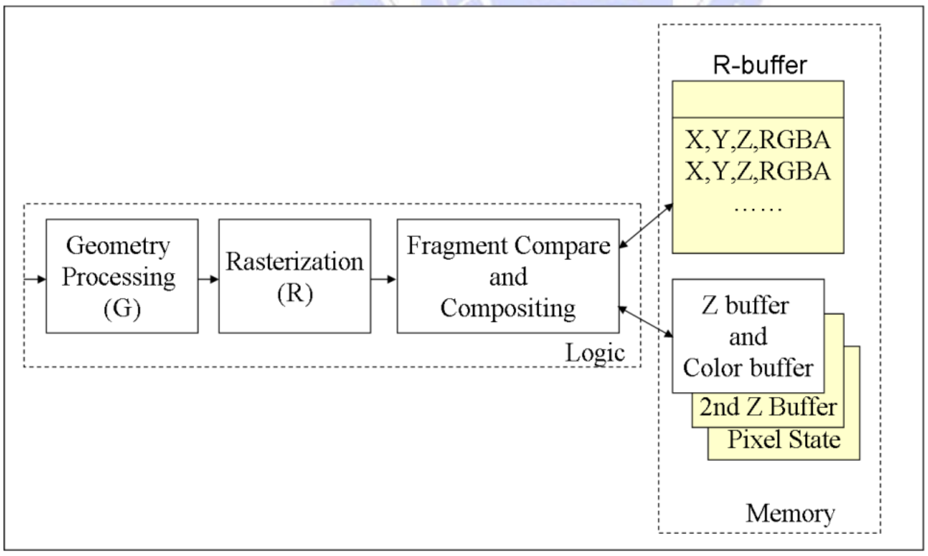
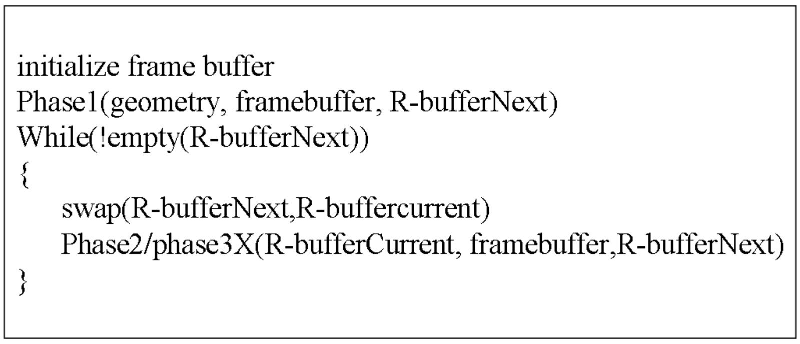


Figure 2-4 R-buffer graphics architecture scheme [Witt01]

R-buffer [Witt01] is a graphics hardware architecture which implements A-buffer software algorithm [Carp84]. Figure 2-4 shows the R-buffer graphics architecture. The

R-buffer architecture is a standard graphics pipeline with additional hardware support: a proposed recirculating fragment buffer, called R-buffer, pixel state memory, and a second z-buffer. In rasterization stage, the objects are rasterized into fragments in arbitrary depth-order. After rasterization, a transparent fragment is sent to the R-buffer, and the depth value of an opaque fragment is compared with the depth value in z-buffer to find the closest opaque fragment which needs to be placed into frame buffer. The transparent fragments behind the closest opaque one are discarded. Then, each transparent fragment in R-buffer is read out iteratively to find the furthest one to be blended with the fragment in frame buffer.

Figure 2-5 shows the high level R-buffer algorithm. Phase 1 rasterizes the primitives into fragments and places the closest opaque fragment into frame buffer, the furthest transparent fragment's depth value into second z-buffer. Phase 1 is equivalent to early z test with the exception that unoccluded transparent fragments are sent into R-buffer and second z-buffer is updated with the depth value of the furthest visible transparent fragment. After all fragments are generated, in phase2, the transparent fragments in R-buffer are discarded if they are occluded by the opaque fragments in frame buffer. If the R-buffer is not empty, the phase3 is processed iteratively to find the transparent fragment whose depth value matches the depth in the second z-buffer from R-buffer and blend that transparent fragment with the fragment in frame buffer, and then, drop that transparent fragment from R-buffer. When the R-buffer is empty, the whole process is finished.



```
initialize frame buffer
Phase1(geometry, framebuffer, R-bufferNext)
While(!empty(R-bufferNext))
{
    swap(R-bufferNext,R-buffercurrent)
    Phase2/phase3X(R-bufferCurrent, framebuffer,R-bufferNext)
}
```

Figure 2-5 R-buffer high level algorithm [Witt01]

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:

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

2.5.2 Hardware oriented algorithm based on weight factors computations

For the convenience of explaining this algorithm [Amor06], we called it WF (Weight Factor) hardware oriented algorithm in brief. WF hardware oriented algorithm is based on the precomputation of the contribution of each fragment to the final color of the pixel with the specialized storage scheme. Figure 2-6 shows the generic structure of WF hardware oriented algorithm. Phase 1 and phase 2 of WF hardware oriented algorithm are similar to those of R-buffer high level algorithm, shown in Figure 2-5. In phase 1, fragments are sequentially generated and the current closest opaque transparent is placed into frame buffer while the transparent fragments are stored into another buffer, called M_{buffer}. In phase 2, all transparent fragments stored in M_{buffer} are analyzed and discarded if they are occluded by the closest opaque fragment stored in frame buffer. In phase 3, each transparent fragment in M_{buffer} is compared with other fragments belonging to the same pixel in order to compute its weight factor and the blending of the fragment is performed.

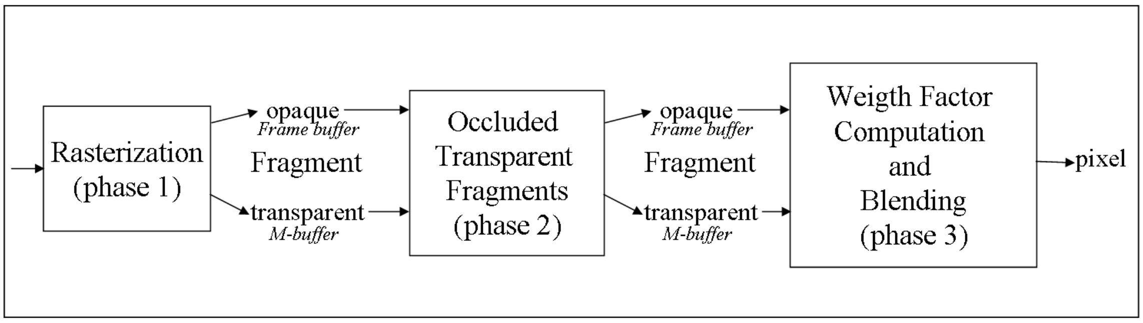


Figure 2-6 Generic structure of WF hardware oriented algorithm

The weight factor computation is based on the analysis of the blending equation (1). By breaking the recursivity of the blending equation (1), the equation can be revised as:

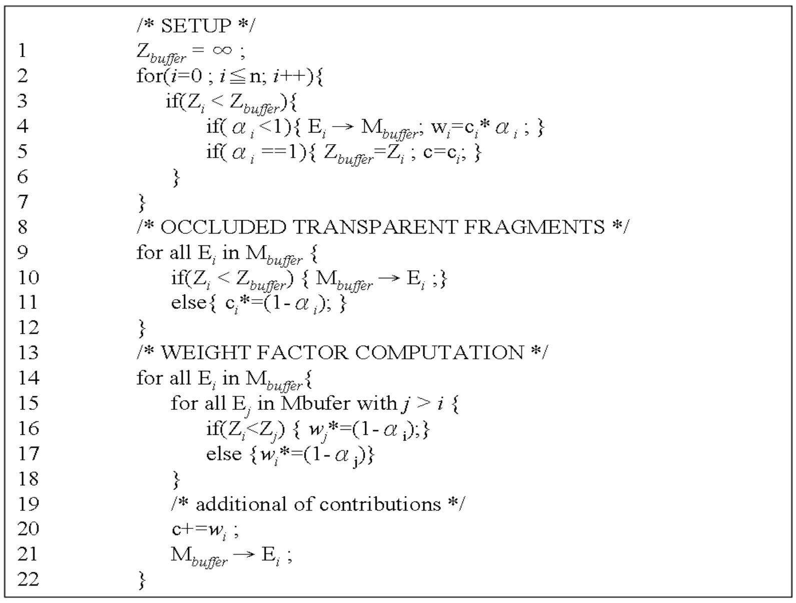
$$c = \sum_{i=0}^{n} w_i \alpha_i c_i \qquad \text{Eq. (2)}$$

where there are n transparent fragments and one opaque fragment belonging to the pixel which has the final color *c*, c_i is the color of the transparent fragment *i*, α_i is the alpha value of fragment *i*, and w_i is the weight factor of the transparent fragment *i*. The weight factor w_i is computed by the accumulative contribution of all transparent fragments *j* in front of the transparent fragment *i* ($Z_j < Z_i$). The equation of w_i can be written as:

$$w_i = \prod_{j=0}^{n} a_j \qquad \text{Eq. (3)}$$

with

$$a_j = \begin{cases} 1-\alpha_j & \text{if } Z_j < Z_i \\ 1 & \text{otherwise.} \end{cases} \qquad \text{Eq. (4)}$$



```
     /* SETUP */
1    $Z_{buffer} = \infty$ ;
2    for($i$=0 ; $i \leqq$ n; $i$++){
3        if($Z_i < Z_{buffer}$){
4            if( $\alpha_i$ <1){ $E_i \rightarrow M_{buffer}$; $w_i$=$c_i$* $\alpha_i$ ; }
5            if( $\alpha_i$ ==1){ $Z_{buffer}$=$Z_i$ ; c=$c_i$; }
6        }
7    }
8    /* OCCLUDED TRANSPARENT FRAGMENTS */
9    for all $E_i$ in $M_{buffer}$ {
10       if($Z_i < Z_{buffer}$) { $M_{buffer} \rightarrow E_i$ ;}
11       else{ $c_i$*=(1- $\alpha_i$); }
12   }
13   /* WEIGHT FACTOR COMPUTATION */
14   for all $E_i$ in $M_{buffer}${
15       for all $E_j$ in Mbufer with $j > i$ {
16           if($Z_i$<$Z_j$) { $w_j$*=(1- $\alpha_i$);}
17           else {$w_i$*=(1- $\alpha_j$)}
18       }
19       /* additional of contributions */
20       c+=$w_i$ ;
21       $M_{buffer} \rightarrow E_i$ ;
22   }
```

Figure 2-7 WF hardware oriented algorithm

The WF hardware oriented algorithm is outlined in Figure 2-7. It can be basically divided into three stages: SETUP (line 1-6), OCCLUDED TRANSPARENT FRAGMENTS (line 9-12), WEIGHT FACTOR COMPUTATION (line 14-22). Assume that there are n+1 fragments are processed sequentially to the same pixel. In SETUP stage, if a fragment is transparent, it is placed into M_{buffer}; otherwise, if a fragment is opaque and closest to the view point at the time, it is stored into frame buffer and Z-buffer is updated by its depth value. Note that some transparent fragments are visible when they are compared to the front-most opaque fragment at the time they arrive, but a closer opaque fragment may arrive later and occlude them. Therefore, in the second stage, OCCLUDED TRANSPARENT FRAGMENTS, those transparent fragments in M_{buffer} are discarded for the reason that they are occluded by the closest opaque fragment. In the last stage, WEIGHT FACTOR COMPUTATION, each fragment is compared with all those following in the M_{buffer} in order to compute its weight factor. Obviously, these three stages in Figure 2-7 are the same as the three phases in Figure 2-6.

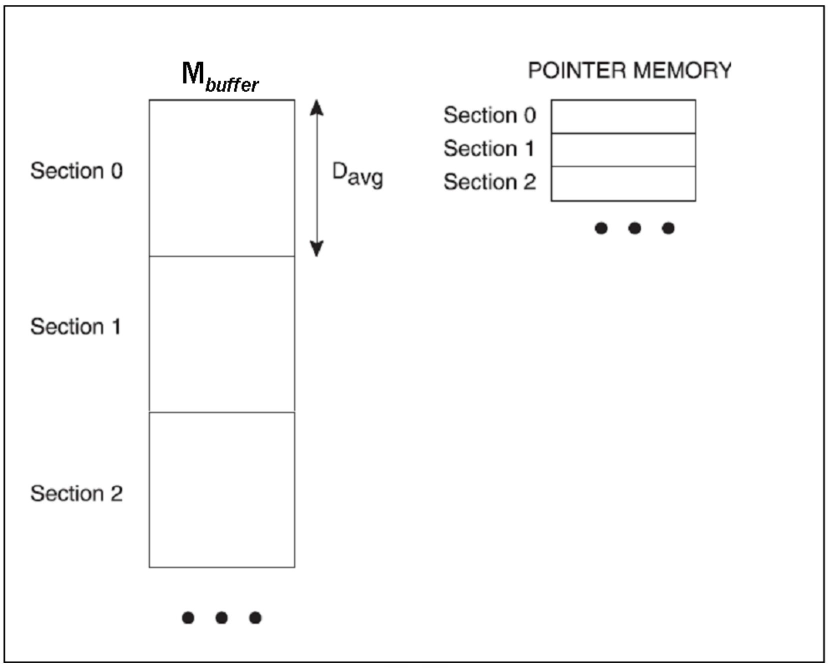


Figure 2-8 Organized memory scheme of WF algorithm

The organized memory scheme of WF algorithm is shown in Figure 2-8. It 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 $F-2 \times D_{avg}$ fragments are stored to another section k ($k>j$), and so on.

2.5.3 T-buffer transparent fragment storage system

T-buffer transparent fragment storage system is based on WF hardware oriented algorithm. In WF hardware oriented algorithm, each pixel is assigned the same size of memory space, no matter whether the pixel has transparent fragments or not. Thus, the main idea of T-buffer is only pixel with transparent fragment(s) will assign memory space in T-buffer. Figure 2-9 is the diagram of transparent fragment storage system, and it consists of SSA table, T-buffer, and NSA table.

SSA Table has W times H entries, where W is defined as the width of a screen, and H is defined as the height of a screen. Each pixel p in a screen has a corresponding entry ep in SSA Table and each entry in SSA Table stores the address of start section for pixel p. Namely, pixel p has assigned the entry ep in SSA Table. If a pixel does not have the start section— the pixel does not have transparent fragments— a nullified address is stored in the corresponding entry in SSA Table.

T-buffer is a storage space for transparent fragments which organized in sections of Lnum, where Lnum represents the maximum number of transparent fragments that can be stored in a section. Each section stores transparent fragments with the same x-y coordinate; that is, fragments belonging to the same pixel are stored gregarious within one section in T-buffer. There might be more than Lnum transparent fragments which have the same x-y coordinate. Thus, more than one section should be assigned to a pixel to extend the capability for storing variable number of fragments. We use NSA Table to record the address of next section which is assigned to store the following fragments.

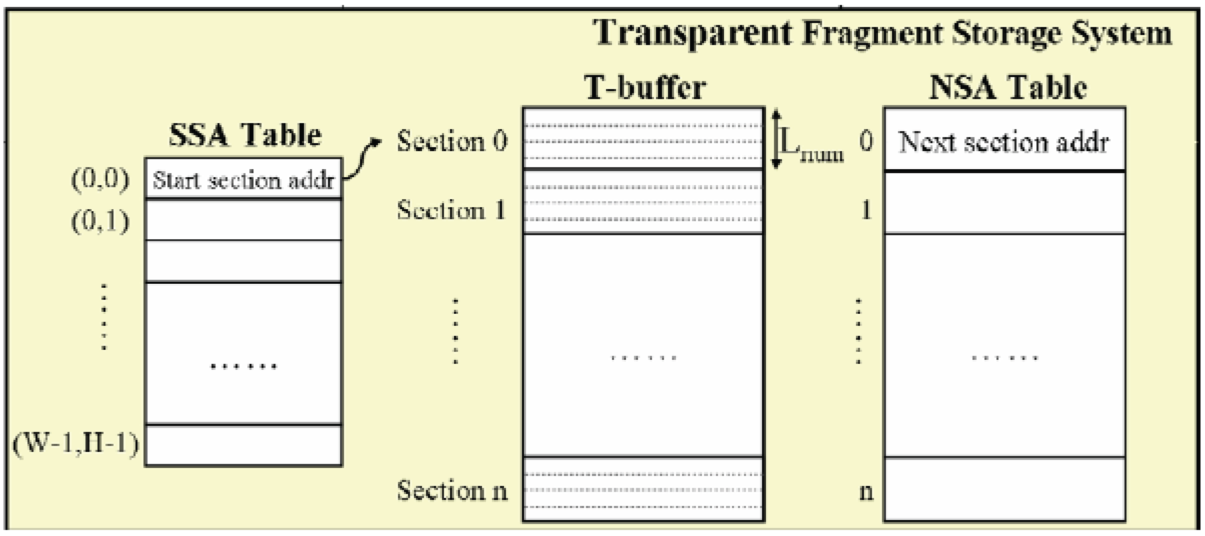


Figure 2-9 The design diagram of transparent fragment storage system

Chapter 3 Design

In this chapter, our design, H-buffer transparent fragment storage system is proposed. The objective of our design is to provide memory size of storage system which needs to be least. This chapter is organized as follows: in section 3.1, the statistics and observation from T-buffer transparent fragment storage system is introduced; in section 3.2, the system design overview is introduced; in the last section of this chapter (section 3.3), we present the H-buffer transparent fragment storage system.

3.1 Statistics and observations

3.1.1Statistics of transparent fragment storage internal fragmentation

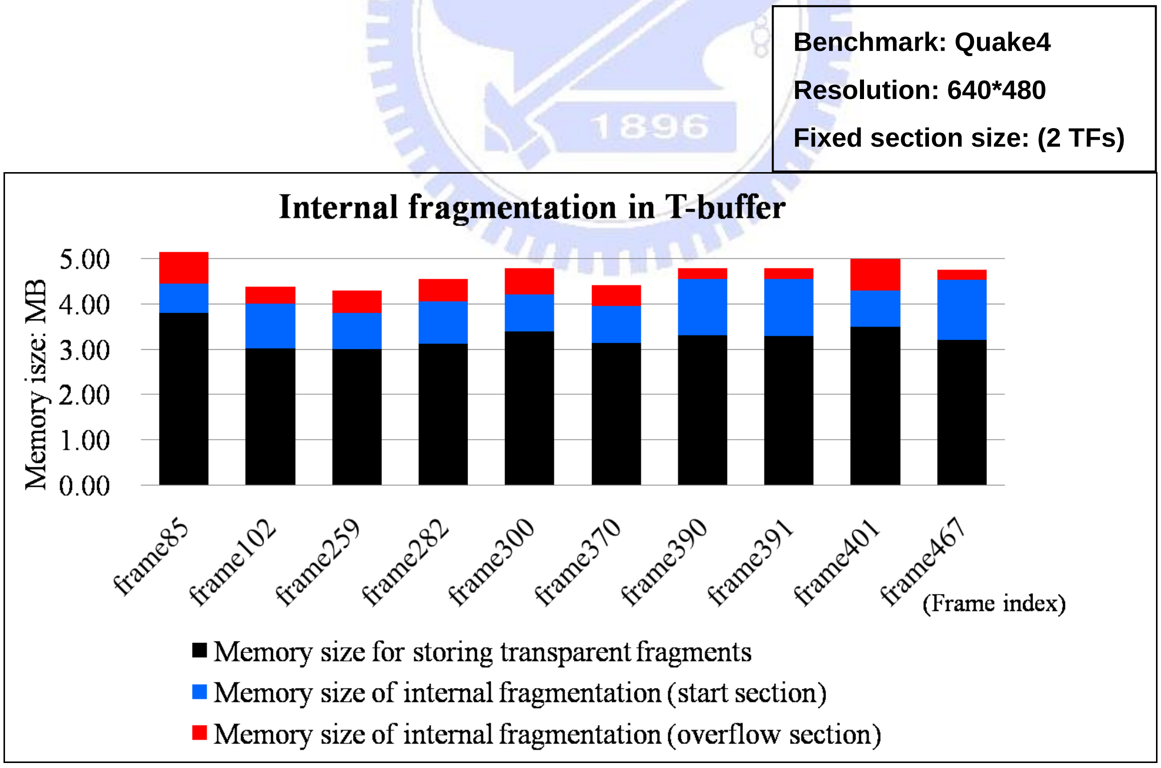


Figure 3-1 Ratio of internal fragmentation size to section size in T-buffer

Due to T-buffer is the lowest memory requirement in recently design, we have made some statistics and observation. The top 10 frames of T-buffer memory requirement is shown in Figure 3-1, and it also shows the internal fragmentation memory size in each frame, where internal fragmentation means the fixed section has been assigned in T-buffer for storing a screen coordinate's transparent fragments, but the entries of section are not fully used. The X axis is frame index, and the Y axis is the memory requirement. The black is the actually memory size for storing transparent fragment size, blue is the memory size of internal fragmentation in start section, and the red is the memory size of internal fragmentation in overflow section. The start section is defined as the first section in T-buffer which screen coordinate used, and overflow section is defined as the section used to store transparent fragments which overflow from start section. We can find the T-buffer size is 5.2MB at least, but there are about 30% memory size is wasted by internal fragmentation. If we can reduce the internal fragmentation of start section and overflow section, the requirement of storage memory can be lower.

3.1.2 Statistics of transparent fragment amount similarity

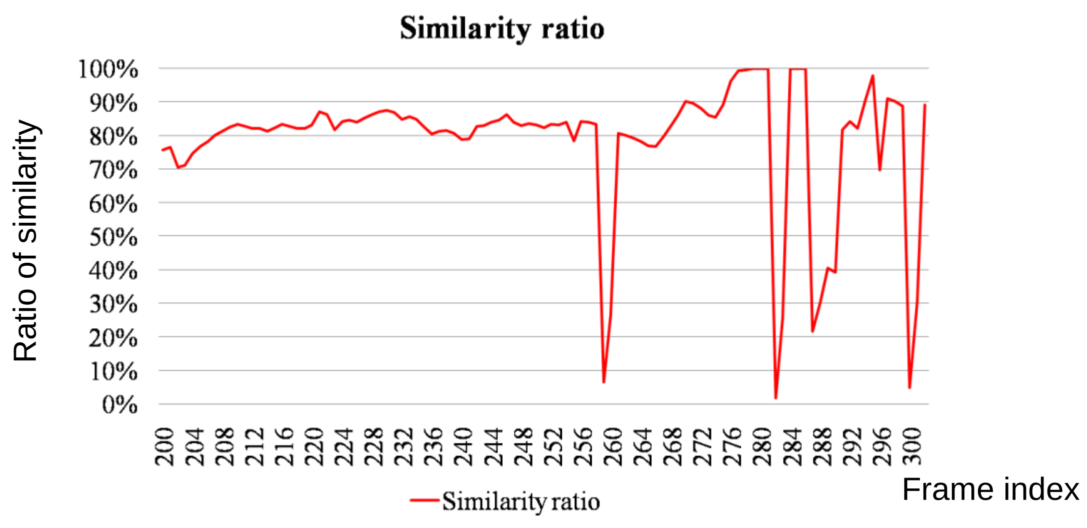


Figure 3-2 Ratio of transparent fragment amount similarity

Figure 3-2 shows frame-similarity of 500 continuous Quake4 frames. The X axis is the frame number, and the Y axis is the transparent fragment amount similarity ratio. The equation of transparent fragment amount similarity ratio is shown below:

$$\text{Transparent fragment amount similarity ratio: } \frac{P_{\text{same}}}{P_{\text{transparent}}} \qquad \text{Eq.(5)}$$

Where $P_{\text{transparent}}$ is the number of screen coordinates which have transparent fragments and the number of screen coordinate do not have transparent fragment but the same screen coordinate in previous one frame have. P_{same} is the number of screen coordinates which have T.F. amount error in one with previous one frame. In 500 continuous frames, the frame-similarity higher than 70% has 417 frames. It means most of pixels have the same transparent fragment amount with the same screen coordinate in previous one frame. Due to frame-similarity, for start section, if we statistic every pixel's transparent fragment amount in current frame, and use this information to assign start section size for next frame, the internal fragmentation by start section may be reduced.

3.1.3 Observation on T-buffer overflow section using mechanism

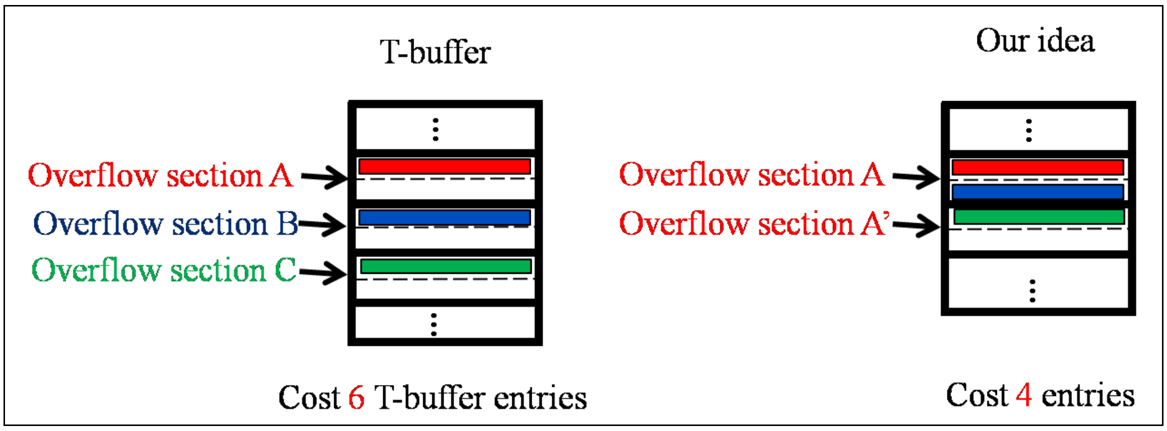


Figure 3-3 Example of overflow section sharing

We also observation on T-buffer overflow section. In T-buffer, each screen coordinate use their own overflow section(s) to store overflow transparent fragments, so that, each screen coordinate's last overflow section may face internal fragmentation problem. For reducing internal fragmentation in overflow section, our idea is let multiple screen coordinates sharing (a) fixed size overflow section(s), in order to reduce the overflow section amount which may face internal fragmentation. Figure 3-3 is an example of overflow section sharing. The different color of rectangles represent different screen coordinate's transparent fragment.

3.2 Design overview

our objective is to reduce the memory requirement by eliminating internal fragmentation. The overview of our proposed transparent fragment storage system is shown in figure 3-4 (a), and figure 3-5 is shown the diagram of T-buffer correspond to H-buffer transparent fragment storage system.

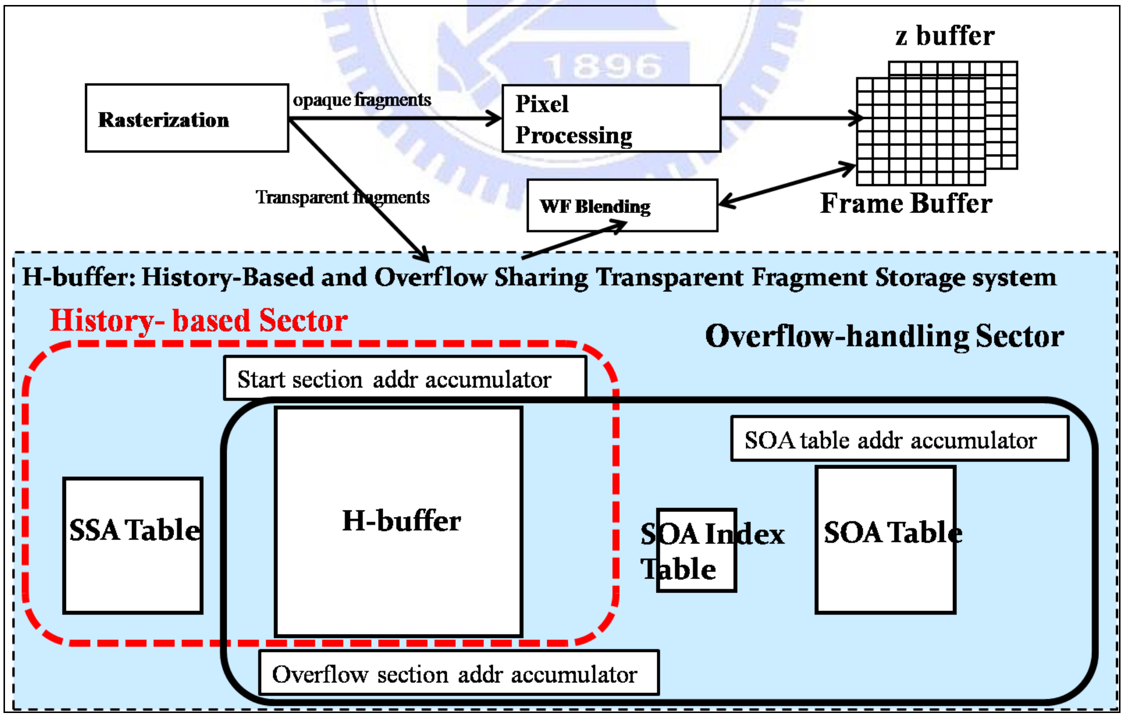


Figure 3-4-1 Design diagram of H-buffer system and the location in pipeline.

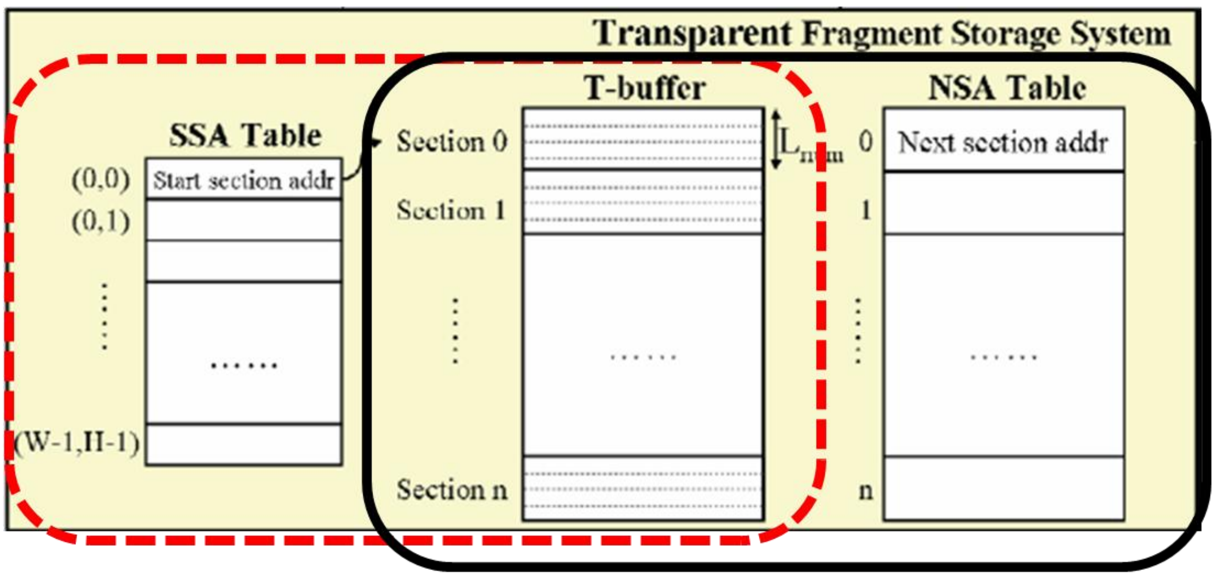


Figure 3-4-1 The diagram of T-buffer system correspond to H-buffer system.

After rasterization stage, fragments are output in arbitrary order. The opaque fragments are sending to pixel shader, and the transparent fragments are send to our transparent fragment storage system. There are two sectors in our system: The first sector is called “history-based” which includes start-section address table (SSA table), start section address accumulator, and H-buffer. Another sector is called “overflow-handling”. It includes share overflow section address table (SOA table), SOA table address accumulator, SOA index table, overflow section address accumulator, and H-buffer. Notice that H-buffer is shared among the two sectors. The detail of the two parts will be introduced in section 3.3 and section 3.4

3.3 History-based sector

3.3.1 Structure of history-based sector

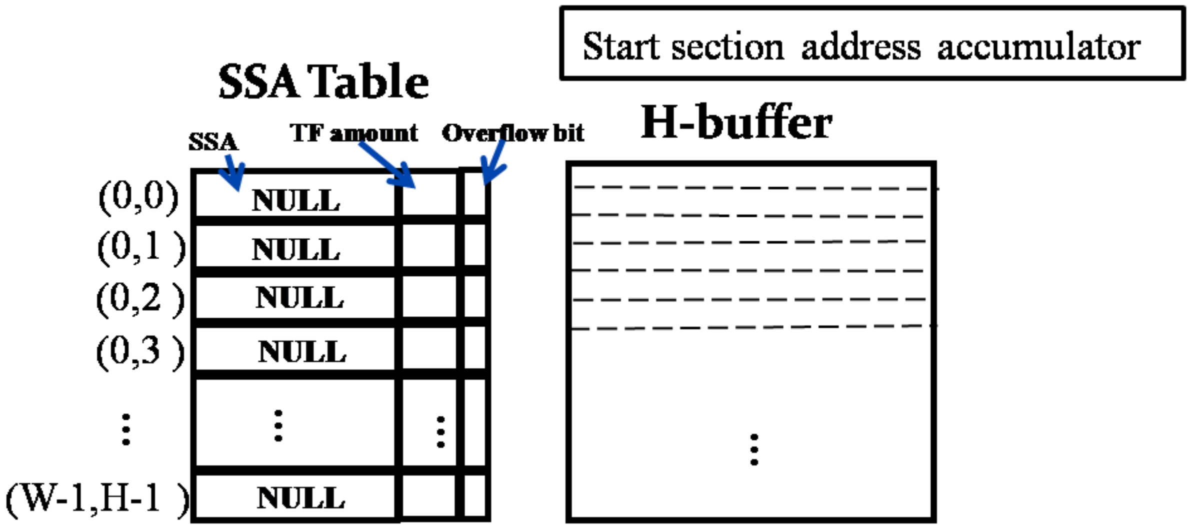


Figure 3-5 Diagram of history-based sector

As shown in the Figure 3-5, the SSA table has W X H entries, and each entry has three fields: SSA (start-section-address) field, TF (transparent fragment) amount field, and overflow bit field, where W is defined as the width of screen, and H is defined as the height of a screen. Each pixel p has a corresponding entry Ep in SSA table, and the SSA field is store the start address of start section within H-buffer for pixel p. If a pixel does not have any transparent fragments, a nullified address is stored in the corresponding entry of SSA field. Another field is TF amount field, and it stores the transparent fragment amount for pixel p. When pixel p have an incoming transparent fragment and its corresponding entry of SSA field is null, then the TF amount field will be used as a hint to allocate the section size in H-buffer. If not, it will be used as a counter to record how many transparent fragments appear on this pixel coordinate. The overflow bit is used to indicate the corresponding start section is full or not. The reason we add overflow bit is to reduce timing for searching empty entry in start section. For example, once a screen coordinate corresponding SSA table entry of SSA field has stored a start section start address, if there is no overflow bit field, we may take multiple cycles to search the empty entry in start section. If there are overflow bit field and the overflow bit indicate the section not full, so that, we can sure the start section is not full, then use the start section start address plus TF amout field value us the incoming transparent fragment storing address.

H-buffer is organized by numbers of entry, and each entry can store a transparent fragment. When a screen coordinate’s first transparent fragment incoming, it has to assign a section in H-buffer for storing transparent fragment. The H-buffer address accumulator always indicates the first none assigned entry in H-buffer, and we will use the accumulator value as the start address of new section, and section size is decided by TF amount field value in SSA table. Noticed that the new start section assign direction is always from H-buffer top to bottom.

3.3.2 An example of history-based sector

Figure 3-6 are examples to show how to assign a start section in H-buffer by SSA table. In figure 3-6 (a), suppose the first incoming transparent fragment with screen coordinate (0,2), and its corresponding SSA field in SSA table is NULL. Hence we will use the H-buffer address accumulator and the TF-amount value to assign a new start section in H-buffer, after that, the address accumulator will accumulate the TF-amount value. After storing the transparent fragment data into start section, the corresponding SSA field will record the start section start address, and TF-amount field will start counting the transparent fragments on this screen coordinate.

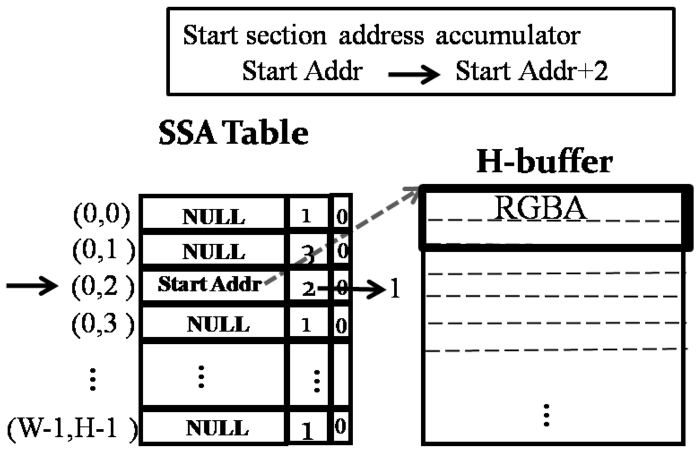


Figure 3-6-1 Upon first appearance of a (0,2) transparent fragment.

In figure 3-6 (b), suppose the second incoming transparent fragment with screen coordinate (0,1), and the process is same with figure 3-6 (a). In figure 3-6 (c), suppose the third incoming transparent fragment with screen coordinate (0,2). The corresponding SSA field has stored start section address, and the overflow bit is “0”, so that, we can use SSA field value plus TF amount field value as the transparent fragment storing address. Finally, update the TF amount field and overflow bit.

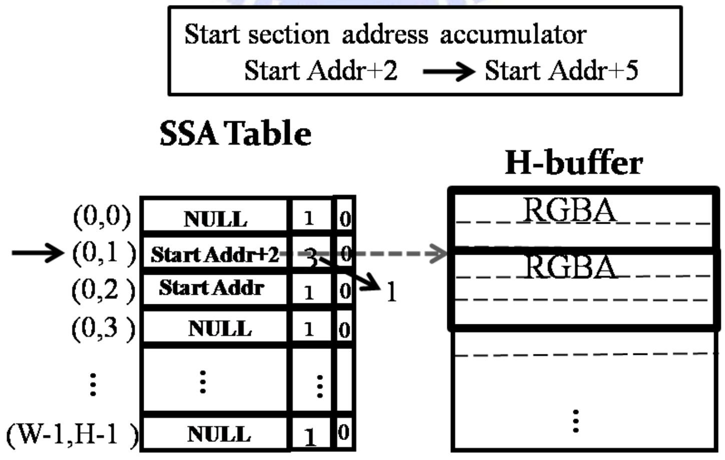


Figure 3-6-2 Upon second appearance of (0,1) transparent fragment.

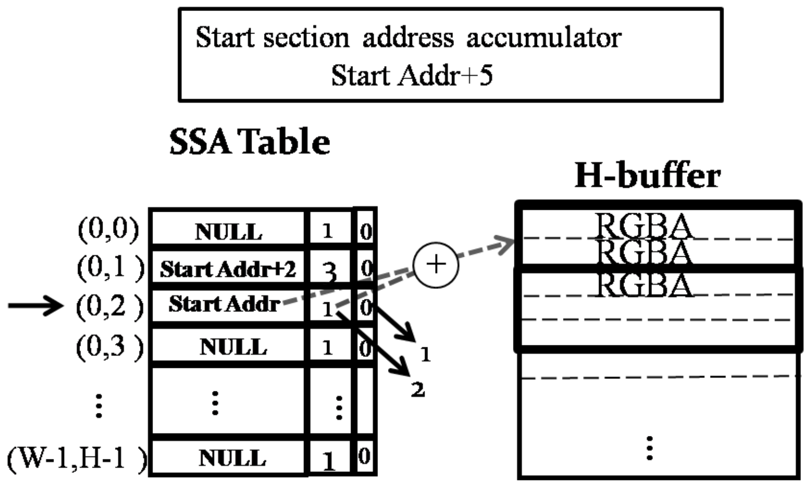


Figure 3-6-3 Upon third appearance of (0,2) transparent fragment.

3.4 Overflow-handling sector

Before present the overflow-handling sector, we will make some discussion about our sharing overflow section mechanism.

3.4.1 Overflow section size

Due to fix size overflow section, we have to assign a sensible value as section size. We estimate the overflow section size by using the average value of overflow transparent fragment amount per overflow pixel and standard deviation. Assume the average value is α, standard deviation is σ, and shared pixel amount is β. Therefore, the overflow transparent fragment amount per pixel is between (α-σ) and (α+σ), and the overflow section size is between $\beta(\alpha-\sigma)$ and $\beta(\alpha+\sigma)$.

3.4.2 Overflow section sharing meahanism

We have two mechanisms: (a) Choose separate pixels (b) Chose neighbor pixels. For mechanism (a), the advantage is each group of share pixels’ overflow transparent amount is close, but the disadvantage is hardware hard to implement. The advantage of mechanism (b) is to exploit the characteristic of neighbor pixel will have same transparent fragment amount, but still have chance cause internal fragmentation in overflow section. For this reason, the mechanism (b) seems to be easily implement, and the internal fragmentation in overflow section may reduce. The Figure 3-7 has shown an example for multiple pixels share overflow sections. Each pixel in the same block will share the same overflow section. In this example, pixels in 2x2 block A share one or more overflow sections in H-buffer, and the block B is also.

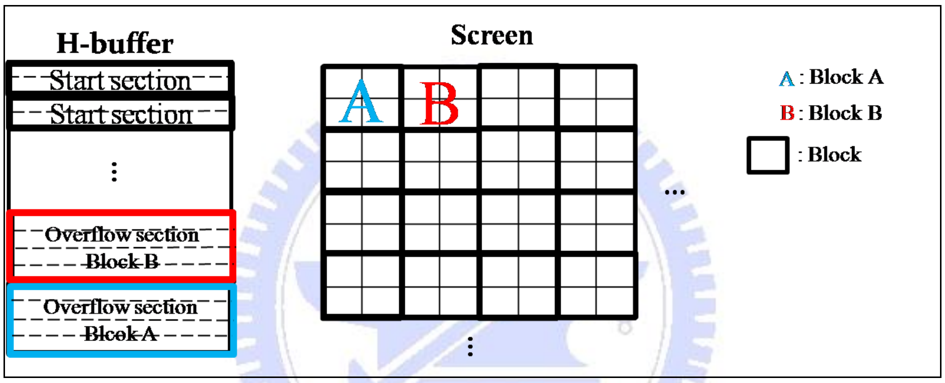


Figure 3-7 An example of sharing overflow sections

Due to sharing, we have to indicate which pixel and which block use which entry in overflow section. As shown in figure 3-8, we will use the X coordinate last M bits, and Y coordinate last N bits as block pixels' coordinate, where M and N is the block width and height. And remain bits of screen coordinate is used to index block.

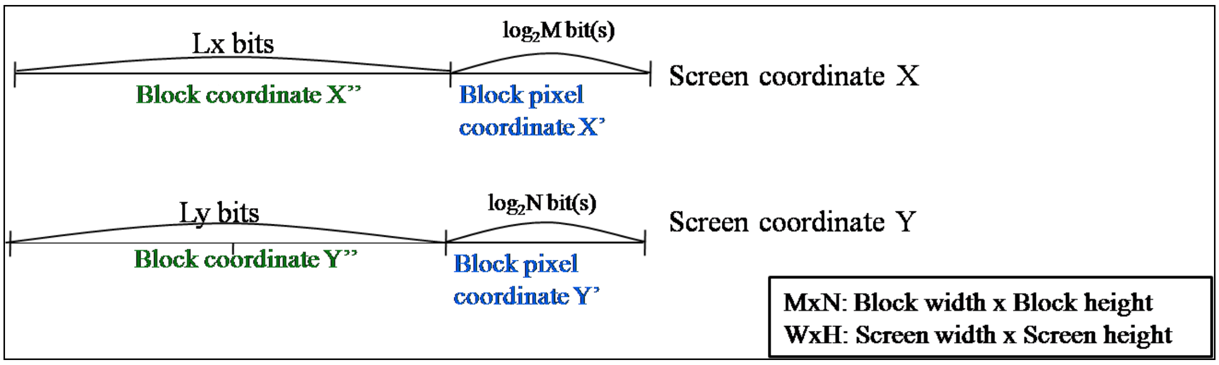


Figure 3-8　　Index information

3.4.3 Structure of overflow-handling sector

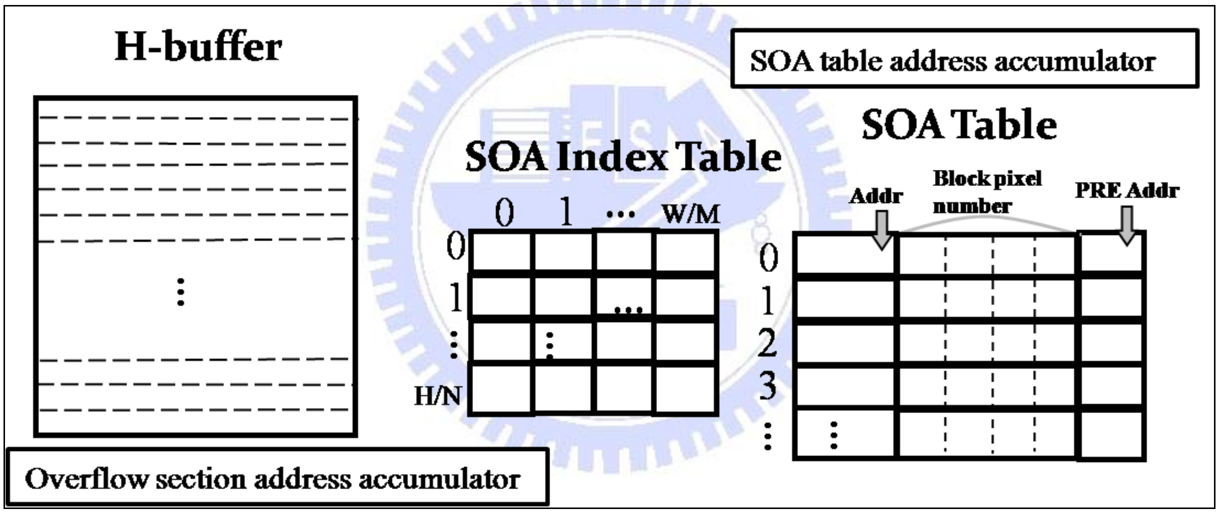


Figure 3-9　　Diagram of overflow handling sector

As shown in Figure 3-9, the SOA table has (WxH / MxN) entries, and we use the block number to index this table, where WxH is defined as screen resolution, and MxN is defined as the block size. Each SOA index table entry records the latest use of SOA table entry address. So that, any screen coordinate can fast to get the corresponding SOA table entry.

The SOA table records the overflow section using information. Each used SOA table entry will correspond to a overflow section. The Addr field is used to record the overflow section start address. The block pixel number field amount is equal to overflow section size. Each block pixel number field record the block number to identify which pixel use the corresponding entry in overflow section. The PRE(previous) field is to record the previous one SOA table entry address which used by same block pixels. If a section in H-buffer face overflow condition, a new fixed size overflow section in H-buffer will be assigned to store overflow transparent fragments. In addition, we will use the SOA table address accumulator to find the first unused SOA table entry to record overflow section information. Once the SOA table entry E corresponding overflow section also face overflow situation, we will assign another overflow section in H-buffer and a new SOA table entry E’. After that, the PRE addr field of SOA table entry E’ will record the address of SOA table entry E. Finally, the corresponding SOA index table entry will update to E’. Thus we can retrieve transparent fragments fast by this recording information.

Noticed that new overflow section assign direction is from H-buffer bottom to top. The benefit is to reduce the overflow section start address recording overhead. Due to fixed size overflow section, we can eliminate the last few bits of overflow section start address, so that, the Addr field requirement bits in SOA table can be reduced.

The Figure 3-10 is an example for overflow-handling sector process. Suppose the block size is 2x2; the overflow section size is 2; the start sections of screen coordinate (2,2), (2,3) are full, and the incoming transparent fragments screen coordinate sequence are (2,3), (2,2), and (2,2). Each overflow handling process are figure 3-10 (a), (b), and (c).

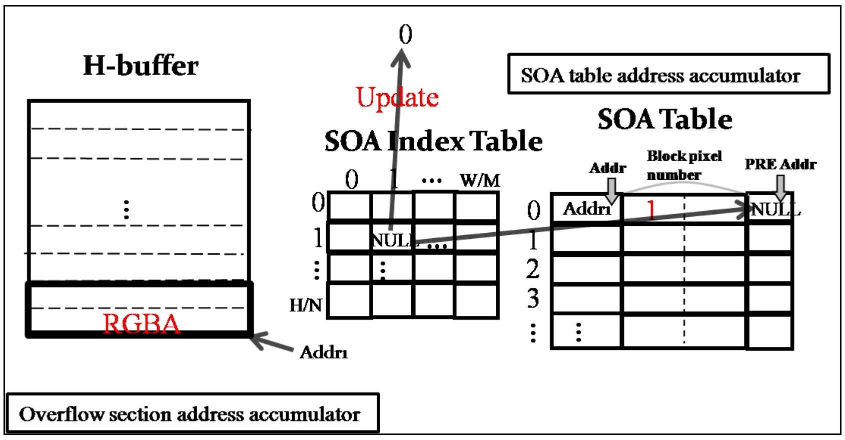


Figure 3-10-1 Screen coordinate (2,3) transparent fragment incoming

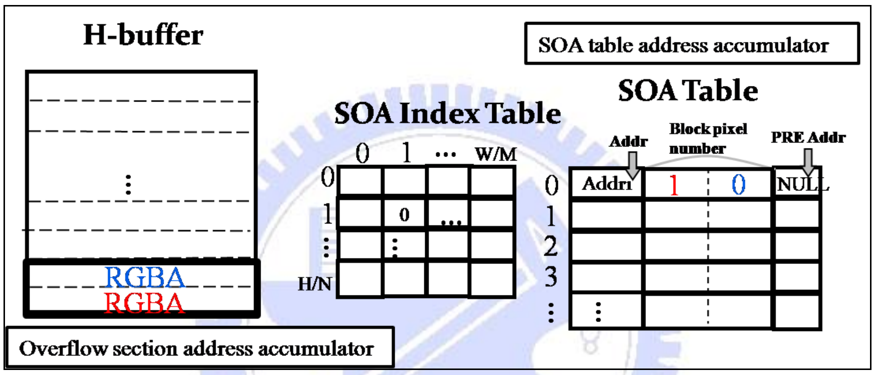


Figure 3-10-2 Screen coordinate (2,2) transparent fragment incoming

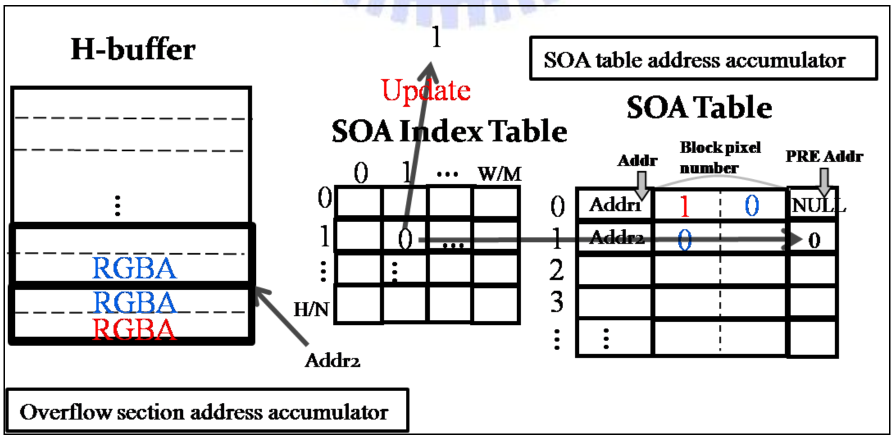


Figure 3-10-3 Screen coordinate (2,2) transparent fragment incoming

3.5 Access preocess and disscussion on timing requirement

3.5.1 Storing process

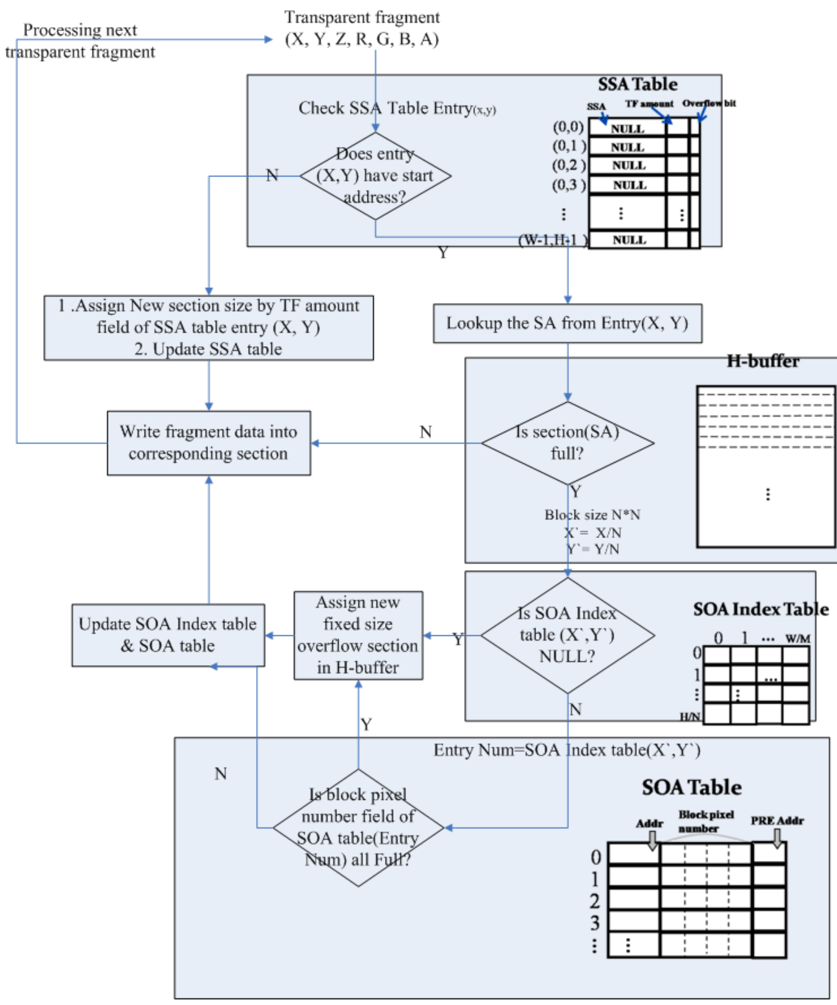

Processing next transparent fragment
Transparent fragment (X, Y, Z, R, G, B, A)
Check SSA Table Entry(x,y)
SSA Table
SSA
TF amount
Overflow bit
Does entry (X,Y) have start address?
N
Y
1 .Assign New section size by TF amount field of SSA table entry (X, Y) 2. Update SSA table
Lookup the SA from Entry(X, Y)
H-buffer
Write fragment data into corresponding section
Is section(SA) full?
Block size N*N X`= X/N Y`= Y/N
Is SOA Index table (X`,Y`) NULL?
SOA Index Table
Assign new fixed size overflow section in H-buffer
Update SOA Index table & SOA table
Entry Num=SOA Index table(X`,Y`)
Is block pixel number field of SOA table(Entry Num) all Full?
SOA Table
Addr
Block pixel number
PRE Addr

Figure 3-11 The flowchart of storing fragments into H-buffer

Figure 3-11 shows the process of storing transparent fragments into H-buffer. Before a transparent fragment with the location (X,Y) is stored into H-buffer, the address of start-section S(X,Y) is read from the SSA table entry(X,Y). If the start-section address field is NULL, the new empty section in H-buffer is assigned for that pixel to store transparent fragments, and the section size is decided by TF amount field. Otherwise, we will check whether the section S(X,Y) is full. If NO, the fragment is stored into section S(X,Y); if YES, it means the start section overflow, so that, we will check the SOA index table and SOA table to find a H-buffer overflow section, and store transparent fragment into that overflow section. Due to storing process timing can be hidden in pixel processing timing, so we do not discussion on storing process timing here.

3.5.2 Retrieving process

Due to overflow section sharing, we cannot retrieve overflow transparent fragments from overflow-handling directly, and cause many cycles to search corresponding pixel's transparent fragments. For balance retrieving time, we need temporary buffer to keep each block pixel's transparent fragment H-buffer memory address. After that, alpha blending process will capture transparent fragments from each blending queue.

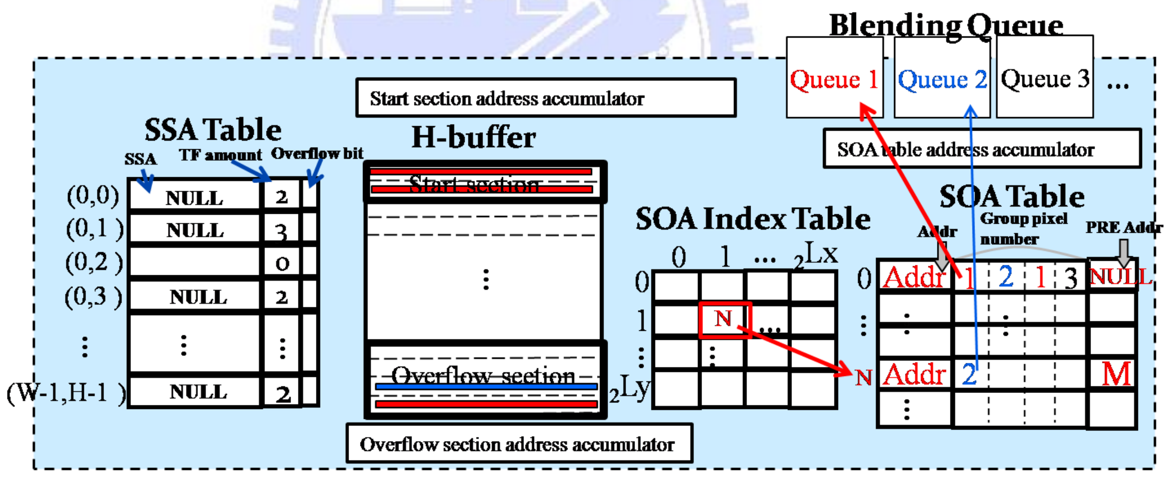
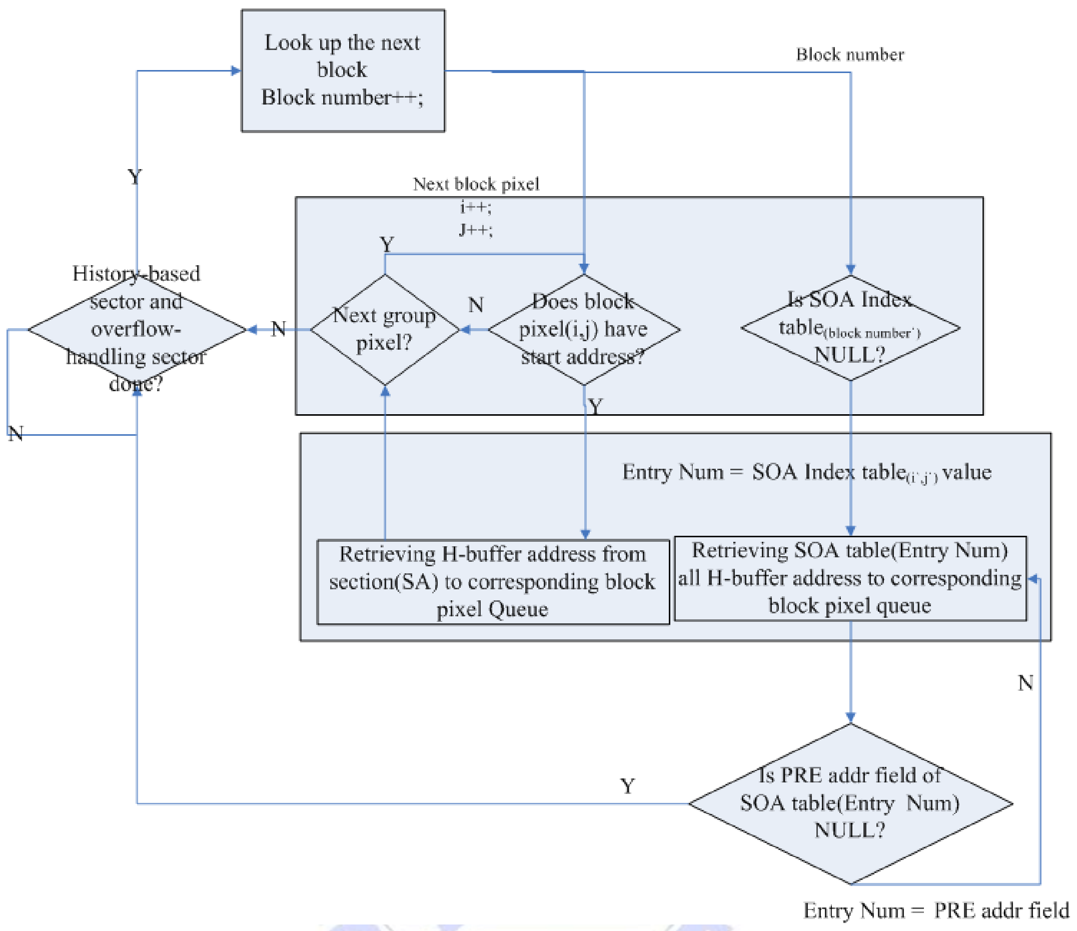


Figure 3-12 Concept of blending queue.

In figure 3-12, alpha blending process is block by block, and the blending queue amount is equal to block pixel amount. When retrieving, the process block will look up SOA table for generating all the overflow transparent fragments' H-buffer address in the block. In the same time, the generating H-buffer address will depend on their block pixel number to decide sending which blending queue. Until the process block finish alpha blending, then the next block may start the retrieving process.



Look up the next block Block number++;
Block number
Next block pixel i++; J++;
Y
History-based sector and overflow-handling sector done?
N
Next group pixel?
Does block pixel(i,j) have start address?
Is SOA Index table(block number') NULL?
Y
Entry Num = SOA Index table(i',j') value
Retrieving H-buffer address from section(SA) to corresponding block pixel Queue
Retrieving SOA table(Entry Num) all H-buffer address to corresponding block pixel queue
Is PRE addr field of SOA table(Entry Num) NULL?
Entry Num = PRE addr field

Figure 3-13 The flowchart of retrieving fragments from H-buffer

Figure 3-13 is the flowchart of retrieving process. When a block start retrieving transparent fragments, the history-based sector and the overflow-handling sector will process at same time. History-based sector will look up SSA table, and generate transparent fragment H-buffer address which in its start section. History-based sector process will sent their start section transparent fragment H-buffer address to correspond bending queue until all the block pixels done. The overflow-handling sector will look up SOA Index table to find the last used SOA table entry. After that, we will use PRE field to continuously look up SOA table for generating overflow transparent fragment H-buffer address, and sent them to the corresponding queue.

Chapter 4 Evaluation Results

In this chapter, we first show our evaluation environment and the characteristic of input frame data (in section 4.1 and 4.2). Then, we show and analyze simulation results of memory requirement and execution time during rendering of each method: R-buffer, WF hardware oriented algorithm, and our TFSS design in section 4.3. In the last section, we briefly summarize our conclusion from the results.

4.1 Evaluation environment

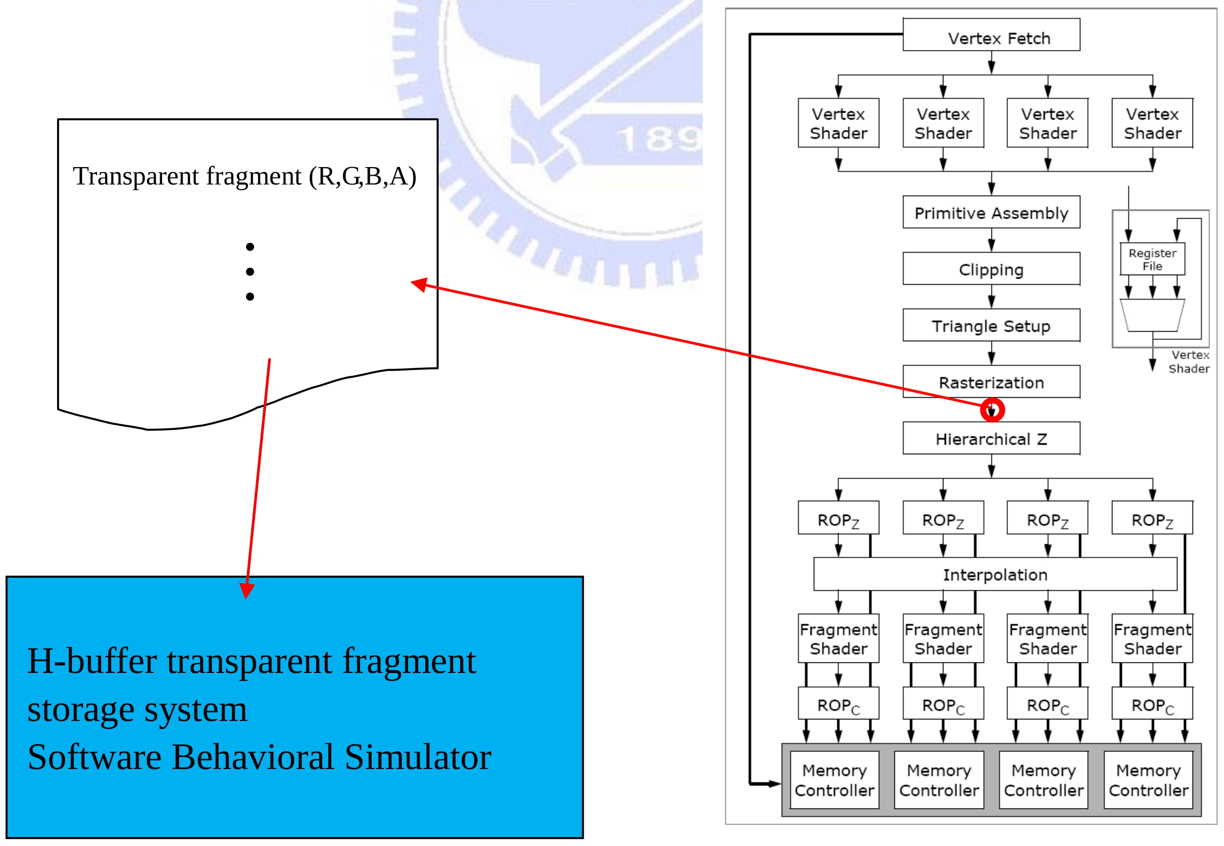


Figure 4-1 Simulation flow and ATTILA architecture [Moya06]

Figure 4-1 shows the architecture of ATTILA simulator and we dump trace of fragment data from ATTILA simulator rasterization for our software behavioral simulator. We implemented a behavioral simulator of the architecture with the transparent fragment storage system in C, and modified ATTILA simulator [Moya06] to output fragment information to a tracefile. The benchmark used in ATTILA simulator is QUAKE4 and DOOM3, two modern graphics applications. The tracefile outputted from ATTILA simulator contains the coordinates and RGBA color components of fragments in frames. Our simulator reads the tracefile and evaluates each sector memory requirement of H-buffer transparent fragment storage system.

The simulation parameters are listed below:

- Display resolution: the number of distinct pixels in each dimension that can be displayed.
- Color component bit-width: the bit-width of each of the RGBA color components
- Block size : block width, block height; Number of pixels share overflow section
- Overflow section size: the memory size of a overflow section in H-buffer

In our simulator, the display resolution are 640×480, the bit-width of each of RGBA color components is 8 bits, and modulate the value of Block size to observe the memory requirements. The overflow section sizes will analysis at next section.

4.2 Overflow section size analysis

As we mention in section 3.4.1, w will statistic number of overflow transparent fragment, average, and standard deviation in QUAKE4 and DOOM3 graphic applications.

Table 4-1-1 Avg. and S.D statistics

	Avg.	S.D.	(Avg.-S.D.)	(Avg.+S.D.)
QUAKE4(640x480)	1.10	0.25	0.85	1.35
DOOM3(640x480)	1.19	0.49	0.7	1.68

Table 4-1-2 Overflow section size in different Num.

Block size	2x2	2x2	4x4	4x4	8x8	8x8	16x16	16x16
Num.(Avg.-S.D.)	3.4	2.8	13.6	11.2	54.4	44.8	217.6	179.2
Num.(Avg.+S.D.)	5.4	6.7	21.6	26.4	86.4	107.5	345.6	430.0
Probably overflow section size	2,4,8		8,16,32		32,64,128		128,256,512	

Table 4-1 (a) is 500 frames of QUAKE4 and DOOM3 statistics. The “Avg.” is average of overflow transparent fragment amount per overflow pixel, and “S.D.” is standard deviation.

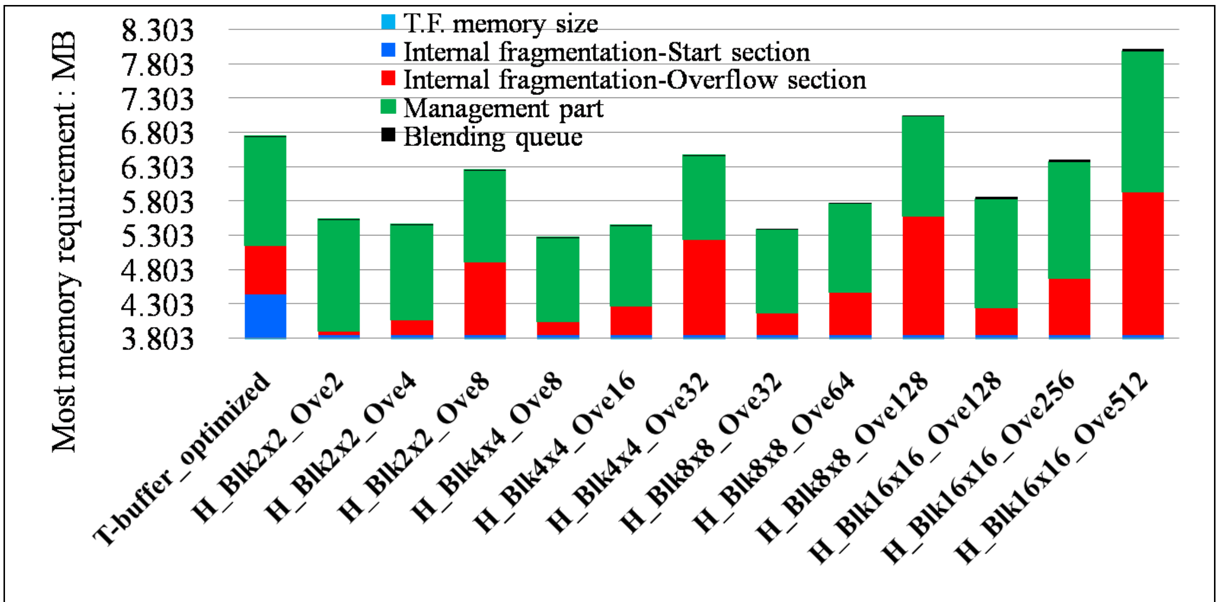
The statistic result shows the overflow transparent fragment is about 1 to 2, and the variation in DOOM3 is bigger than QUAKE4. Table 4-1 (b) is a rough estimate about overflow section size. We observe the distribution of screen coordinates' overflow transparent fragment amount in average of 600 frames. In QUAKE4 benchmark, with a standard deviation, there are 90 percent screen coordinates locate in; In DOOM3 benchmark, with a standard deviation, there are 87 percent screen coordinates locate in. So that, the average value with a standard deviation value might close the overflow transparent fragment amount per screen coordinate in application. After that, we will get the probably overflow section size. The two smaller size in each column has possibility to be the optimization value. The reason is the biggest one would cause more than 60% block with internal fragmentation. Even if the overflow handling sector memory requirement may smaller, but after tradeoff, it is not benefit. In the next section, we will show the simulation result to prove our viewpoint.

4.3 Simulation results

In this section, we show the simulation results of memory requirements of H-buffer transparent fragments storage system and compare with the strongest competitor: T-buffer transparent fragment storage system.

4.3.1 Memory requirement

We obtained the simulation result of memory requirement of our design and T-buffer, as shown in Figure 4-2 and Figure 4-3. The X-axis is the memory requirement at least in 600 frames; the Y-axis indicates different systems, and different sharing mechanism. Notice that the management part in T-buffer system includes SSA table and NSA table; in H-buffer system includes SSA table and overflow-handling part's table.



•Blending queue: (Block width x Block height) x (Index bits) x 20
•T-buffer: the highest transparent storage utilization design, and it's memory requirement has been optimized
•H_MxN_Num.: H-buffer system_Block width x Block height_Overflow section size

Figure 4-2 Memory requirement comparison (QUAKE4)

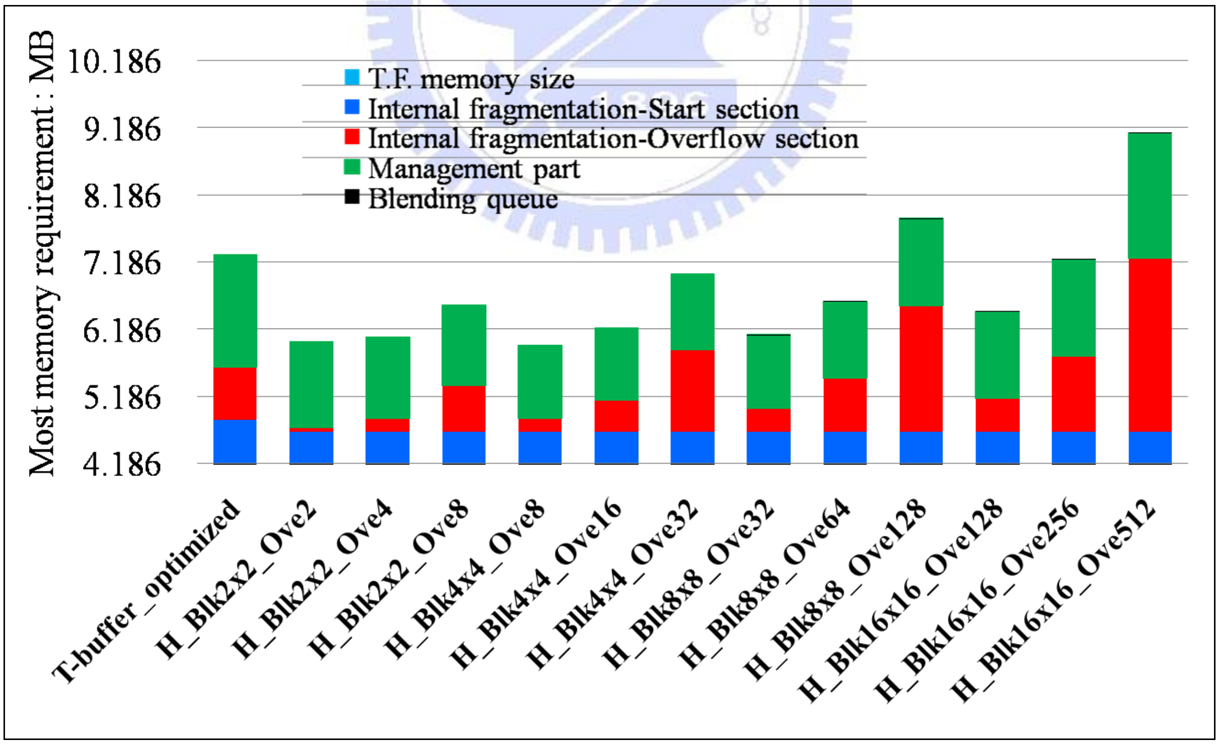


Figure 4-3 Memory requirement comparison (DOOM3)

Due to simulation result have same number of transparent fragments, so we ignore the T.F memory size in Figure 4-2 and Figure 4-3. Compare with the T-buffer system, our H-buffer transparent fragment storage system has lower the memory requirement about 25% in Quake4, and 20% in DOOM3. Furthermore, the overhead of blending queue is small even use bigger block. Notice that in smaller block, like 2x2 and 4x4, the internal fragmentation increase obviously with the overflow section size also increase. Otherwise, due to sharing, the bigger block and overflow section size use less overflow-handling part. In another side, the small block cause many overflow, so that, the overflow-handling sector memory requirement has increased. From figure 4-2 and figure 4-3, we can find H-buffer system not only reduce internal fragmentation but also decrease the management part memory requirement.

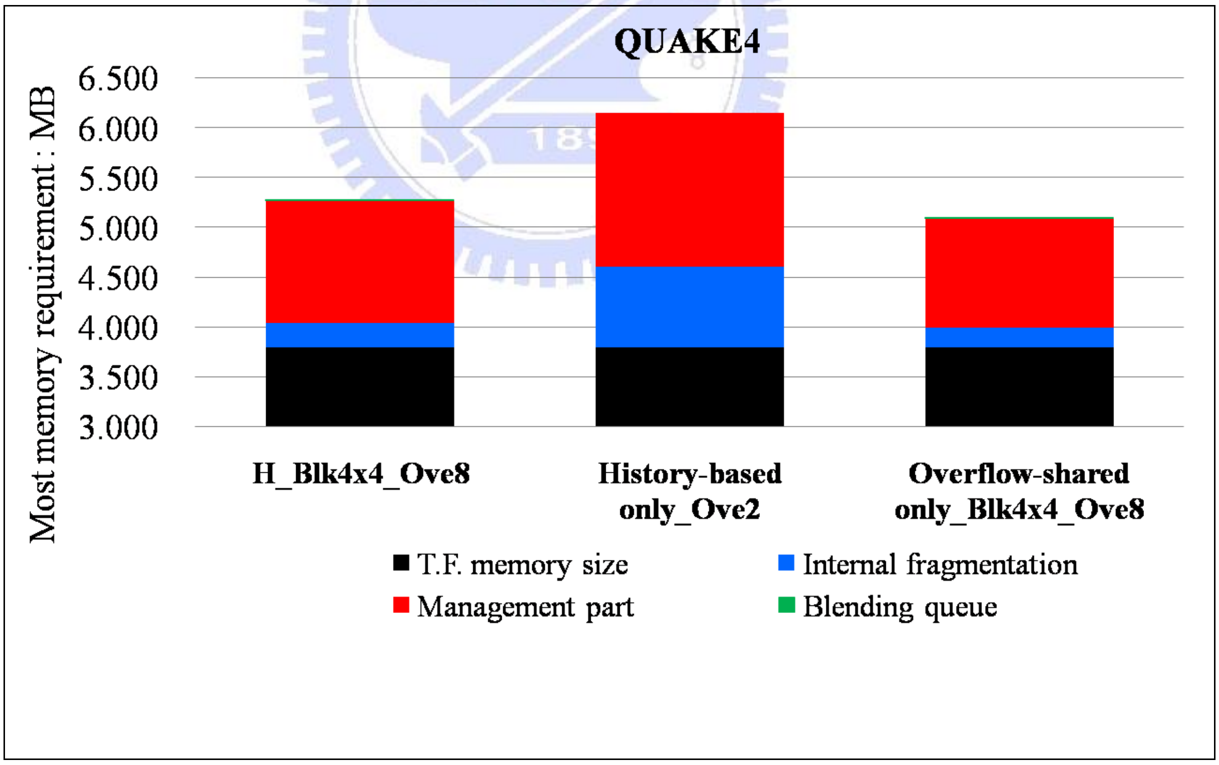


Figure 4-4 Memory requirement comparison (Quake4)

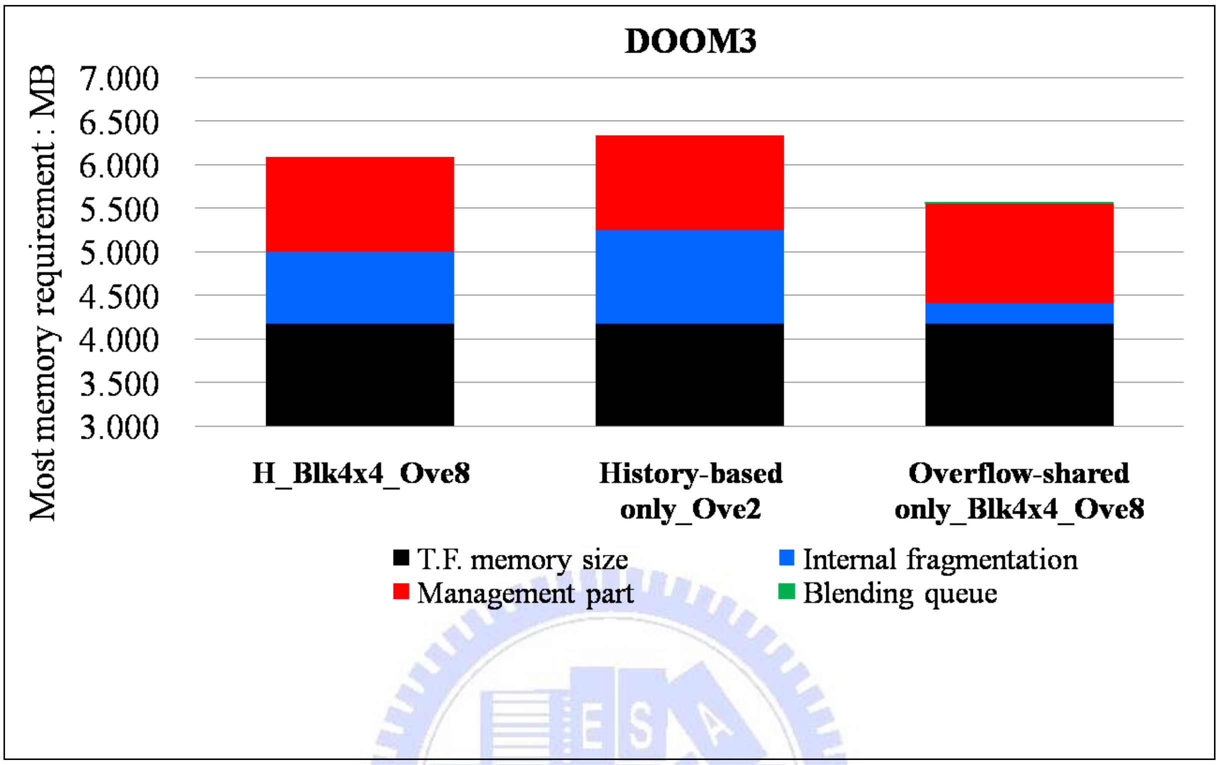


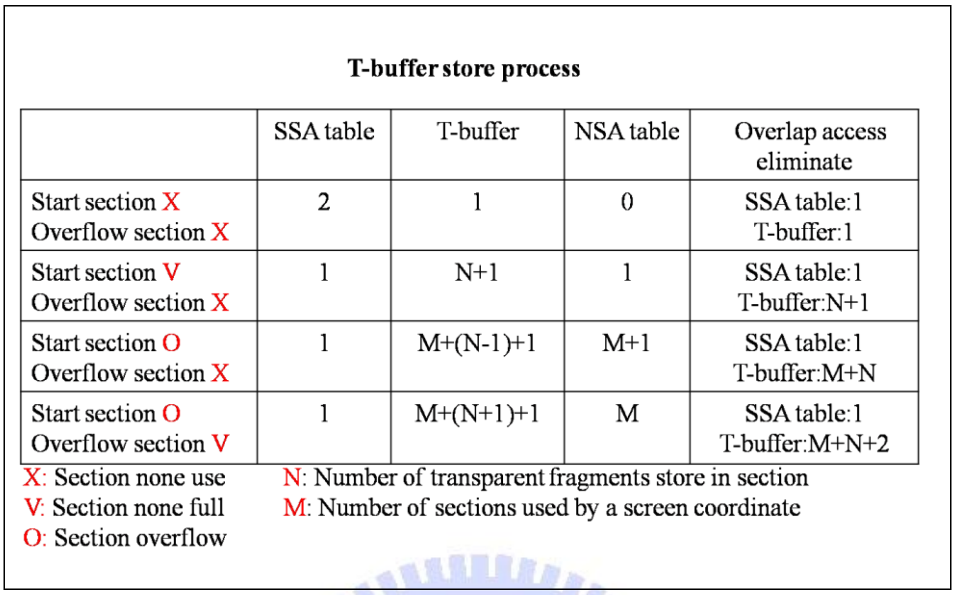
Figure 4-5 Memory requirement comparison (DOOM3)

We also consider the memory requirement of separate history-based and overflow-shared. Figure 4-4 and Figure 4-5 show the history-based only and overflow-shared memory requirement. It shows the history-based method reduce internal fragmentation less than overflow-shared method. The probably reason is history-based method still face the view change problem. We can find the memory requirement of overflow-shared method is better even use the two methods together.

4.3.2 Timing requirement

For emulating the timing requirement, we list all storing process condition of each table and transparent fragment buffer access. Table 4-2 shows the storing process condition of T-buffer system and H-buffer system.

Table 4-2　Storing process condition

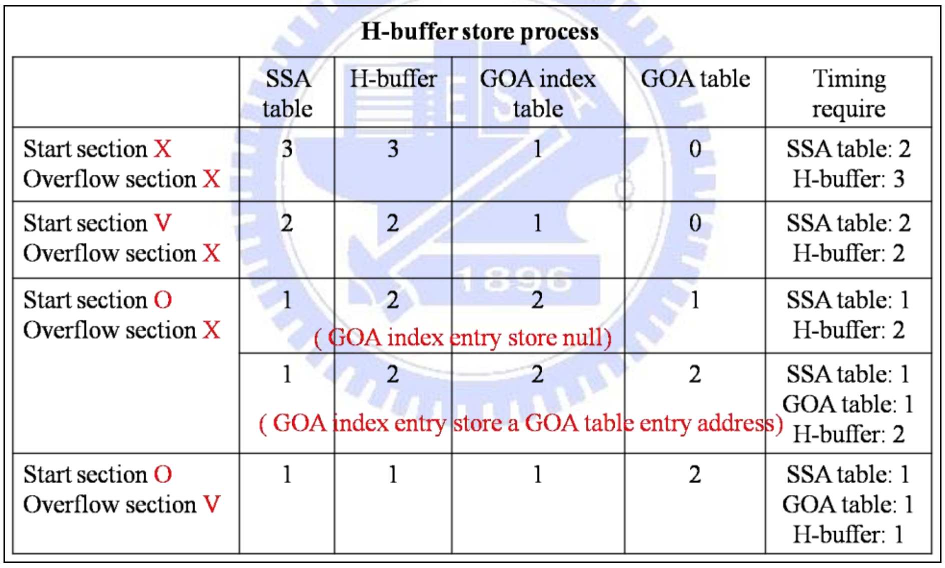


T-buffer store process

	SSA table	T-buffer	NSA table	Overlap access eliminate
Start section X Overflow section X	2	1	0	SSA table:1 T-buffer:1
Start section V Overflow section X	1	N+1	1	SSA table:1 T-buffer:N+1
Start section O Overflow section X	1	M+(N-1)+1	M+1	SSA table:1 T-buffer:M+N
Start section O Overflow section V	1	M+(N+1)+1	M	SSA table:1 T-buffer:M+N+2

X: Section none use
V: Section none full
O: Section overflow

N: Number of transparent fragments store in section
M: Number of sections used by a screen coordinate

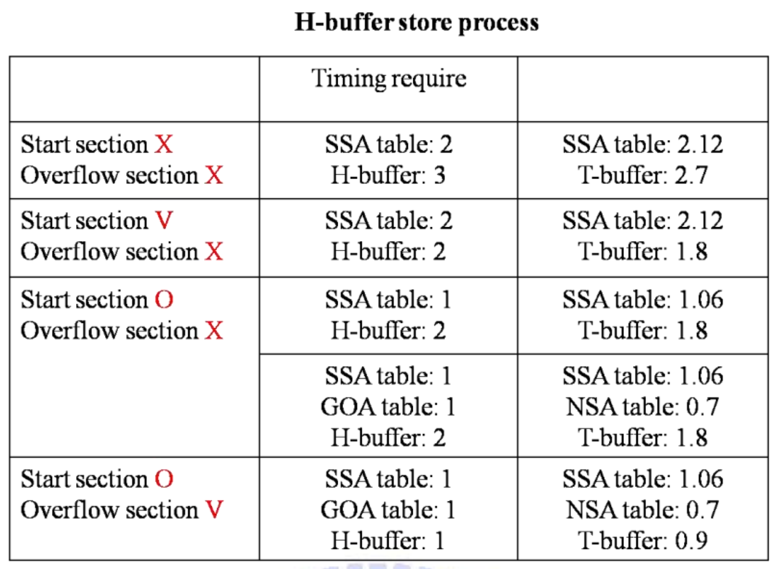


H-buffer store process

	SSA table	H-buffer	GOA index table	GOA table	Timing require
Start section X Overflow section X	3	3	1	0	SSA table: 2 H-buffer: 3
Start section V Overflow section X	2	2	1	0	SSA table: 2 H-buffer: 2
Start section O Overflow section X	1	2 (GOA index entry store null)	2	1	SSA table: 1 H-buffer: 2
	1	2 (GOA index entry store a GOA table entry address)	2	2	SSA table: 1 GOA table: 1 H-buffer: 2
Start section O Overflow section V	1	1	1	2	SSA table: 1 GOA table: 1 H-buffer: 1

For comparing the timing of two different systems, we change the H-buffer timing require to T-buffer timing requirement by memory access time ratio. For example, the T-buffer access time is 2ns, and H-buffer access time is 1ns, and the original H-buffer access times should multiply 0.5. Notice that the memory access time was estimated by CACTI. Table 4-3 is H-buffer system storing timing requirement and turn to T-buffer system.

Table 4-3 H-buffer system storing timing requirement turn to T-buffer system

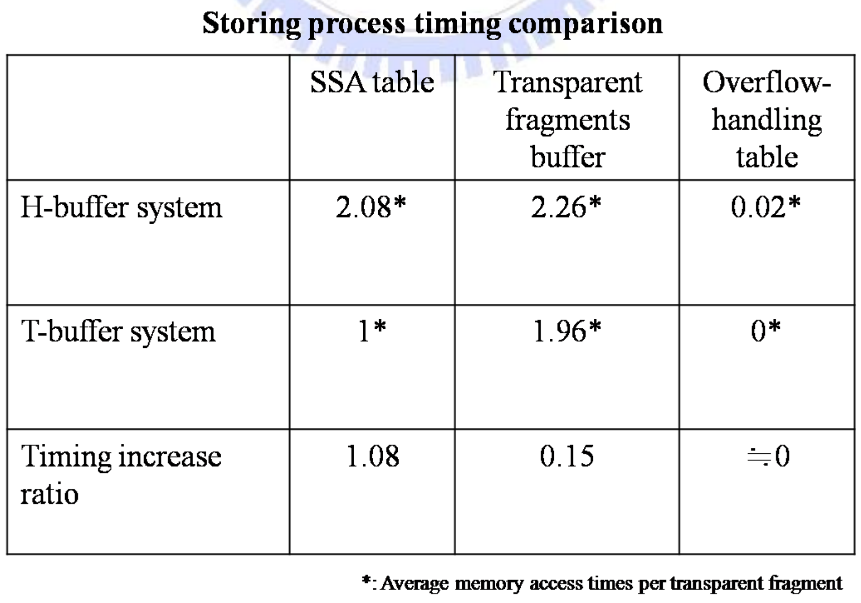


H-buffer store process

	Timing require	
Start section X Overflow section X	SSA table: 2 H-buffer: 3	SSA table: 2.12 T-buffer: 2.7
Start section V Overflow section X	SSA table: 2 H-buffer: 2	SSA table: 2.12 T-buffer: 1.8
Start section O Overflow section X	SSA table: 1 H-buffer: 2	SSA table: 1.06 T-buffer: 1.8
	SSA table: 1 GOA table: 1 H-buffer: 2	SSA table: 1.06 NSA table: 0.7 T-buffer: 1.8
Start section O Overflow section V	SSA table: 1 GOA table: 1 H-buffer: 1	SSA table: 1.06 NSA table: 0.7 T-buffer: 0.9

We use timing requirement as our simulation parameter, and simulate each table and transparent fragment buffer access times requirement per pixel. Table 4-3 shows the two different system timing requirement and H-buffer timing increase ration.

Table 4-3 Systems storing timing requirement comparison



Storing process timing comparison

	SSA table	Transparent fragments buffer	Overflow-handling table
H-buffer system	2.08*	2.26*	0.02*
T-buffer system	1*	1.96*	0*
Timing increase ratio	1.08	0.15	≒0

*: Average memory access times per transparent fragment

,

We can found the H-buffer system memory access time is more than T-buffer system. But in fact, the overhead is not serious. The overhead is about 10 thousand ns, and it is more smaller than ten millions ns of blending a frame.

Chapter 5 Conclusion and Future Work

5.1 Conclusion

In this thesis, we propose a transparent fragment storage system for order-independent transparency. For reducing internal fragmentation in past design, the main ideas are transparent fragment amount similarity and overflow section sharing.

For QUAKE4 benchmark, our transparent fragment storage system, in comparison with T-buffer transparent fragment storage system, reduces 22% memory requirement in average, From the evaluation result, we find that as the number of transparent fragments per pixel increases, our storage system has more advantages on memory requirements.

5.2 Future work

In the overflow-handing sector, we propose a blending queue for temporary storing retrieving transparent fragment. The reason is we want to reduce searching corresponding pixel's transparent fragment time. But once the blending queue full, retrieving time may increase. Therefore, the blending queue overflow–handling mechanism is worth to discuss.

References

[Lin08] Hui-Chen Lin. Transparent Fragment Storage System for Order-Independent Transparency in GPU. Institute of Computer Science and Engineering, National Chiao-Tung University, 2008.

[Amor06] 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.

[Witt01] 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

[Carp84] L. Carpenter. The A-buffer, an Antialiased Hidden Surface Method. In Proceedings of ACM SIGGRAPH, pp.103-108,1984.

[Moya06] 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.

[Watt00] Alan Watt. 3D Computer Graphics. 3rd edition. Pearson Addison-Wesley publishing. 2000

[Watt84] T. Porter and T. Duff. Compositing Digital Images. ACM Computer Graphics (Siggraph 84 Proc.), pp. 253–259, 1984.

[Mamm89] 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.

[Snyd98] John Snyder and Jed Lengyel. Visibility Sorting and Compositing without Splitting for Image Layer Decompositions. In Proceedings of the 25th annual conferenceon Computer graphics and interactive techniques, pp. 219–230, 1998.

[Ever01] Cass Everitt. Interactive Order-Independent Transparency. In Technical report, NVIDIA Corporation. ACM Press, 2001.

[Joup99] N. P. Jouppi and C.-F. Chang. Z3 : an Economical hardware Technique Rendering for High-quality Antialising and Transparency. In Proceesings of Graphics Hardware, pp. 85-93, ACM/Eurographics, 1999.