A Cost-Effective Supersampling for Full Scene AntiAliasing

Byung-Uck Kim[1], Woo-Chan Park[2], Sung-Bong Yang[1], and Tack-Don Han[1]

[1] Dept. of Computer Science, Yonsei University, Seoul, Korea,
kimbu@yonsei.ac.kr,
[2] Dept. of Internet Engineering, Sejong University, Seoul, Korea

Abstract. We present a graphics hardware system to implement supersampling cost-effectively. Supersampling is the well-known technique to produce high quality images. However, rendering the scene at a higher resolution requires a large amount of memory size and memory bandwidth. Such costs can be alleviated by grouping subpixels into a fragment with a coverage mask which indicates which part of the pixel is covered. However, this may cause color distortion when several objects either overlap or intersect with each other within a pixel. In order to minimize such errors, we introduce an extra buffer, called the *RuF(Recently used Fragment)-buffer*, for storing the footprint of a fragment most recently used in the color manipulation. In our experiments, the proposed system can produce high quality images as good as supersampling with a smaller amount of memory size and memory bandwidth, compared with the conventional supersampling.

Keywords: Antialiasing, Supersampling, Graphics Hardware, Rendering Algorithm

1 Introduction

With growth of user demand for high quality images, the hardware-supported full scene antialiasing (FSAA) has become commonplace in 3D graphics systems. Artifacts due to aliasing are mostly caused by insufficient sampling. To attenuate such aliasing problem, supersampling has been practiced in the high-end graphics system [2] and begins to be adopted by most pc-level graphics accelerator.

In supersampling, 3D objects are rendered at a higher resolution and then are averaged down to the screen resolution [8]. Hence it requires a large amount of memory size and memory bandwidth. For example, $n \times n$ supersampling requires n^2 times bigger both memory size and memory bandwidth than one-point sampling. Some reduced versions of it have been practiced; sparse supersampling [2] that populates sample points sparsely and adaptive supersampling [1] in which the only discontinuity edges are supersampled. In multi-pass approach, the accumulation buffer [4] has been proposed in which one scene is rendered several times and these images are then accumulated, one at a time, into the accumulation buffer. When the accumulation is done, the result is copied back into the

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 271–281, 2004.

frame buffer for viewing. However, it is obvious that rendering the same scene n times takes n times longer than rendering it just once. Both supersampling and the accumulation buffer are well integrated into the Z-buffer (also called *depth buffer*) algorithm that is adopted by most rendering systems for the hidden surface removal. Moreover, Z-buffer algorithm handles correctly interpenetrating objects.

Rather than rendering each subpixel individually, A-buffer approach [3] groups subpixels into a fragment with a coverage mask that indicates which part of the pixel is covered. Such a representation is efficient in reduction of the memory and bandwidth requirements because it shares the common color value instead of having its own color value per subpixel. To apply Carpenter's blending formulation [3] for antialiasing of opaque objects, fragments should be sorted in the fragment list by their depth value. The fragment lists can be implemented by a pointer-based linked list [6] or a pointer-less approach [9]. For reducing noticeable artifacts, correct subpixel visibility calculations are more important that correct antialiasing of subpixels. Therefore, the more concise depth value representation has been practiced in [5].

This paper presents a cost-effective graphics hardware system that renders the supersampled graphics primitives with full scene antialiasing. In our approach, an area-weighted representation of a fragment using a coverage mask is adopted, as in A-buffer, to reduce the memory and bandwidth requirements. This may cause color distortion when several objects either overlap or intersect with each other within a pixel. In order to minimize such errors, we introduce an extra buffer, called the *RuF (Recently used Fragment)-buffer*, for storing the footprint of a fragment most recently used in the color manipulation. In addition, we introduce the new color blending formulation for minimizing color distortion by referencing the footprint of the RuF-buffer. In our simulation, we compared the amount of memory size and memory bandwidth of the proposed scheme with those of supersampling and investigated the per-pixel color difference of the images produced from both methods. For various 3D models with 8 sparse sample points, the proposed algorithm reduces the amount of memory size and memory bandwidth by 35.7% and 67.1%, respectively, with 1.3% per-pixel color difference as compared with supersampling.

The rest of this paper is organized as follows. In Section 2, we describe the proposed graphics architecture. Section 3 explains fragment processing algorithm for antialiasing. Section 4 provides the experimental results of image quality, memory and bandwidth requirement. Finally, the conclusions are given in Section 5.

2 The Proposed Graphics Architecture

In this section, we present the data structure and memory organization for processing a pixel with subpixels individually or a fragment. We also describe the proposed graphics hardware with the newly developed RuF-buffer.

2.1 Data Structure and Memory Organization for a Pixel

Figure 1 shows the data structure and memory organization for representing a pixel with subpixels individually and a fragment. Here we assumed that each pixel has 8 sparse sample points. In supersampling method, each subpixel is processed individually; the pair of color and depth value per subpixel is stored into color buffers and depth buffers in the frame buffer. Hence, the required memory size per pixel is $m \times (C + Z)$ bits where C and Z is color (32 bits) and depth value (24 bits), respectively and m is the number of sample points. In this example, $8 \times (32 + 24)$ bits $= 56$ bytes per pixel is required.

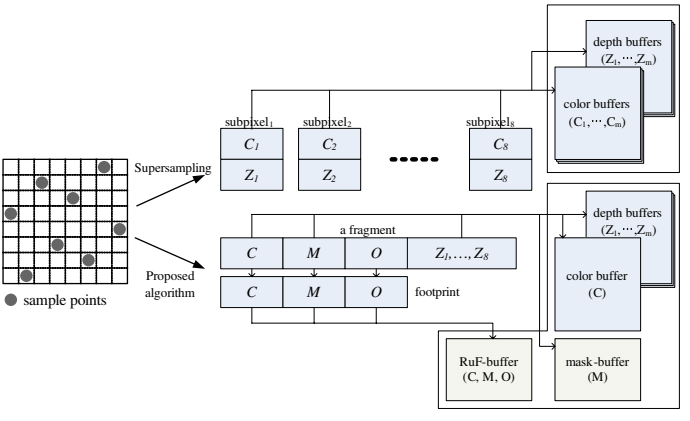

Fig. 1. Data structure and memory organization for a pixel.

In the proposed scheme, the data structure for a fragment is basically originated from the one of the A-buffer. Subpixels within a pixel are grouped into a fragment that shares the common color value (C) with a coverage mask (M). Moreover, we can easily compute the color contribution of a fragment within a pixel since a coverage mask represents an area-weighted value. For handling subpixel visibility correctly, depth value per subpixel $(Z_1, \cdots, Z_m)$ is kept individually. An object tag (O) is the unique identifier per object and can be generated sequentially by the rendering hardware incorporated with modelling software [6]. It is used for post-merging fragments; if two fragments in a pixel have the same object tag value then the footprint of both fragments can be merged into the RuF-buffer. The RuF-buffer holds the footprint of a fragment that is recently used in the color manipulation phase. The footprint of a fragment consists of color, coverage mask and object tag of a fragment and it will be used for correct handling the hidden surface removal. The required memory size per a pixel is $(2 \times (C + M) + m \times Z + O)$ bits where M is the coverage mask (m bits) and O is the object tag (16 bits). Here, 2^{16} objects are assumed to be enough for representing 3D model in a scene. Therefore, the memory size

Table 1. Memory requirement

m	Supersampling	Our approach	Reduction ratio
4	28 bytes	24 bytes	14.3%
8	56 bytes	36 bytes	35.7%
16	112 bytes	62 bytes	44.6%
64	448 bytes	218 bytes	51.3%

of $(2 \times (32+8)+8 \times 24+16)$ bits = 36 bytes per pixel is required when 8 sparse sample points are used.

Table 1 shows the comparison of the memory requirement between supersampling and the proposed algorithm as the number of sample points increases. As shown in the results, the reduction ratio of the memory requirement begins to be larger as the number of sample points increases since our approach can save the memory requirement for representing individual color value per subpixel by sharing the common color value.

2.2 RuF-Buffer Graphics Architecture

Figure 2 shows the proposed graphics architecture with the conventional geometric-processing and rasterizer-processing. We add the mask-buffer and the RuF-buffer into the conventional architecture. Generally, 3D data are geometric-processed with rotating, scaling and translation. The processed results are fed into the rasterizer-processing. In rasterizer-processing, the fragments of each polygon are generated by scan-conversion and then passed through occlusion test such as Z-buffer algorithm and various image mapping such as texture mapping or bump mapping. Finally, the color value of each pixel is manipulated and stored into the color buffer in the frame buffer. When all the fragments are processed, the color values in the frame buffer are sent to a display device.

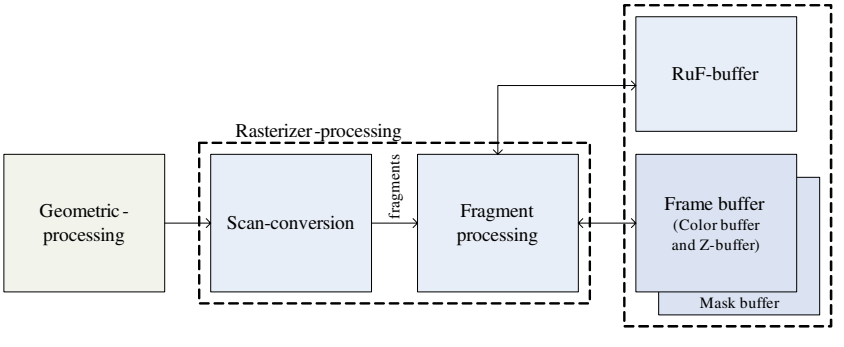

Fig. 2. The proposed graphics architecture.

3 Fragment Processing for Antialiasing

Figure 3 shows three phases of the newly introduced functional unit of fragment processing: the *occlusion test*, the *color manipulation*, and the *RuF-buffer recording*. Roughly speaking, the newly fragment incoming into the graphics pipeline is tested with Z-buffer algorithm per subpixel. If it is totally occluded by the one previously stored in the frame buffer, called a *prepixel*, then it will be discarded and the next fragment will be processing.

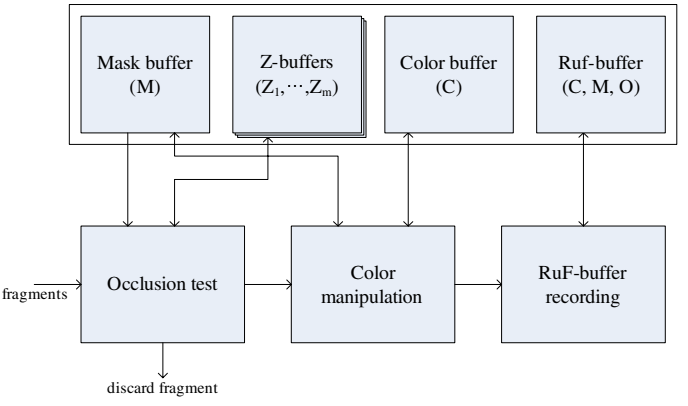

Fig. 3. Functional units for fragment processing.

Otherwise, we calculate the visible fraction of an incoming fragment, called a *survived surface*, and the hidden surface of the prepixel occluded by it. For calculating color value of a pixel, the survived surface will be added into and the hidden surface will be subtracted from the color buffer. In this phase, we look up the RuF-buffer for investigating the color value of the hidden surface. Finally, the survived fragment is merged into the RuF-buffer to allow more opportunity by covering the larger portion within a pixel. In describing each stage, the subscripts, 'i', 'p', and 'r' are used for denoting the attribute of an incoming fragment, the prepixel in the frame buffer, and the footprint in the RuF buffer. For instance, M_i is for the coverage mask of an incoming fragment. For simplicity, we assume that a coverage mask used in formulation returns the area-weighted value; for instance, if the number of sample points is eight and M_i covers three subpixels then M_i in formulation denotes the value of 3/8.

Detail descriptions of each stage in the fragment processing are presented as follows:

The occlusion test : The depth comparison per subpixel between an incoming fragment and a prepixel are tested with the conventional Z-buffer algorithm. Then the mask composite for the survived surface (M_s) and the hidden surface

(M_h) are processed. Z-buffers are updated with new depth values of the survived surface.

The color manipulation : The survived surface is visible fraction of an incoming fragment. Hence, its area-weighted color value should be added into the color buffer. In addition, the hidden surface of a prepixel occluded by the survived surface should be subtracted from the color buffer. To look up the color value of the hidden surface, we investigate the match between the hidden surface and the footprint of the RuF-buffer through the mask comparison of ($M_k = M_h \cap M_r$). If M_k is a subset of M_r then we can totally remove the color contribution of the hidden surface from the frame buffer using the formulation of Eq. 1.

$$\text{new } C_p = C_p + C_i \times M_s - C_r \times M_k \tag{1}$$

However, since the footprint of the RuF-buffer may not provide any information about some parts of the hidden surface we expand the formulation of Eq. 1 to compensate color value with a slight error. The fourth term of formation in Eq. 2 compensates color value by subtracting the area-weighted color value for the blind parts ($M_b = M_h - M_k$) of the hidden surface from the frame buffer.

$$\text{new } C_p = C_p + C_i \times M_s - C_r \times M_k - C_p \times M_b \tag{2}$$

The RuF-buffer recording : Generally, the polygonal surfaces of an object exist in a coplanar space. Therefore, each neighbored surface generates fragments that share the same pixel on their boundary [3],[6]. So, they can be merged into one in the post-processing. Fragments that come from the same object will be merged into one since the same tagged object has same property. The merging process can be computed as follows:

$$\text{new } M_r = M_r \cup M_s; \text{new } C_r = C_r \times \frac{M_r}{\text{new } M_r} + C_r \times \frac{M_s}{\text{new } M_r} \tag{3}$$

However, if the survived fragment has the different object tag then the RuF-buffer is reset with the survived surface. The new object now begins to be drawn.

Figure 4 and Table 2 shows an example of fragment processing for each event and its associated color manipulation. In this example, subpixels are located on 3×3 grid sample points and three consecutive fragments (f_1, f_2, f_3) are incoming into the graphics pipeline. In Figure 4, we assume that a fragment f_1 was already processed in the previous phase; the frame buffer was initialized and then filled with f_1, where f_1 of object one (O_1) covers four subpixels with a color value C_0. Hence, the color value of a prepixel in the frame buffer (C_1) was computed as an area-weighted value of f_1, and then the footprint of f_1 was stored in the RuF-buffer. Now two fragments, f_2 and f_3, are newly fed into the graphics pipeline sequentially.

The left on the figure shows the processing of a fragment f_2 which of object 2 (O_2) has C_2 as a color value and covers four subpixels. The occlusion test is

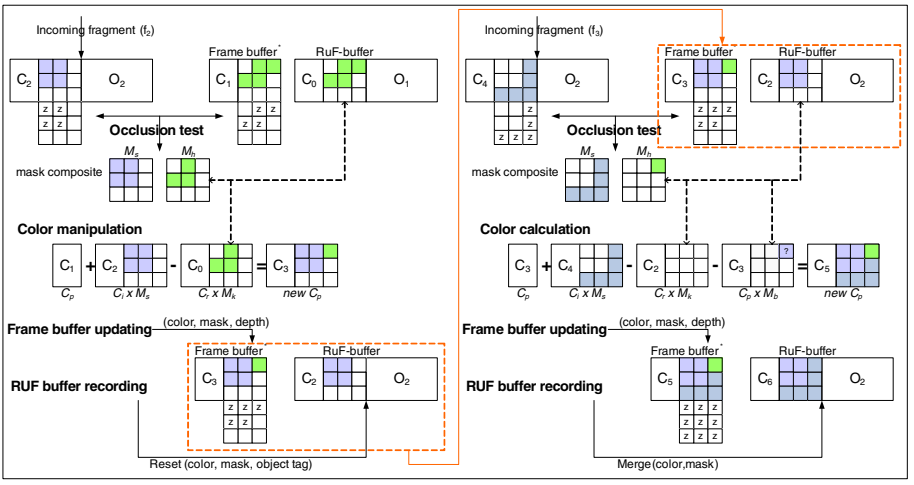

Fig. 4. Example of fragment processing

Table 2. The color manipulation process

Events	Color manipulation	Color buffer
Frame buffer initialize	–	$\emptyset$
f_1 incoming	$C_0 \times \frac{4}{9}$	C_1
f_2 incoming	$C_1 + C_2 \times \frac{4}{9} - C_0 \times \frac{3}{9}$	C_3
f_3 incoming	$C_3 + C_4 \times \frac{5}{9} - C_3 \times \frac{1}{9}$(instead of $C_0 \times \frac{1}{9}$)	C_5

first processed, and then the survived and hidden masks are composited by Z-buffer algorithm per subpixel. In this example, four subpixels of f_2 are survived and three subpixels in a prepixel are occluded. Moreover, the information of the hidden surface can be referenced through the RuF-buffer. Thus the new color value (C_3) can be computed by Eq. 1 without any color distortion. Finally, the RuF-buffer is reset with the footprint of f_2 since the object tag of f_2 is different to the one of the RuF buffer stored in the previous phase.

The right on the figure shows the processing of a fragment f_3, which of object 2 (O_2) has C_4 as a color value and covers five subpixels. Similarly as in f_2 processing, the hidden and survived surfaces are computed in the occlusion test; in this example, five subpixels are survived and one subpixel is occluded. However, in the color manipulation, the footprint of the RuF-buffer cannot provide any information of the hidden surface (M_b). So, the color value is compensated by subtracting it from the prepixel; for instance, the color value of $C_3 \times \frac{1}{9}$ are used in formulation instead of $C_0 \times \frac{1}{9}$. This causes color distortion with the mere color difference of $(C_3 \times \frac{1}{9} - C_0 \times \frac{1}{9})$. In RuF-buffer recording, the footprint of two fragments f_2 and f_3 are merged into one since they have the same object tag, and then it covers the entire portion of a pixel.

4 Empirical Results

In our experiments, the 3D models described with OpenGL functions are geometric-processed and passed through scan-conversion in the Mesa library, which is the OpenGL-clone implementation and can be accessed in public domain [7]. We modified the Mesa library to output the tracefile of a fragment with a coverage mask. The resulted tracefile is fed into the simulator that implements the proposed architecture in C. Then the final image of 200×150 resolution is produced as shown in Figure 5.

Table 3 describes the characteristics of 3D models used in our experiments where the number of vertices (V), triangles (T), fragments (F), and objects (O) are provided. In our experiment, we decided to use eight-sparse sample point ($8\times RuF$) for the antialiasing architecture because it has been successfully practiced in high-end graphics systems [2]. To provide an indication of the performance in our approach, various supersampling methods are also simulated; one-point sampling ($1 \times S$, 1 subpixel per pixel), 8 sparse supersampling ($8 \times S$, 8 subpixels per pixel), 4 by 4 supersampling ($4 \times 4S$, 16 subpixels per pixel), and 8 by 8 supersampling ($8 \times 8S$, 64 subpixels per pixel).

Table 3. The characteristics of 3D models used in our experiments

Model Name	V	T	F	O
Al	3618	7124	11975	35
Castle	6620	13114	17444	16
Dolphins	885	1692	4570	3
Pig	3522	7040	7499	3
Rose+vase	4028	3360	5425	5
Teapot	3644	6320	6807	1
Venus	711	1418	5464	1

4.1 Image Quality

We present the image quality with respect to the number of sample points. To observe the quality of final scenes, the error metric of per-pixel color difference is used as shown in Eq. 4.

$$\text{Per-pixel color difference} = \sum_{\forall i,j} \sum_{c=r,g,b} (p_{ijc} - q_{ijc})^2, \tag{4}$$

where p_{ij} and q_{ij} are the pixels from the same location of a reference image and a test image, respectively.

In order to make a small number of pixels with large difference more noticeable, the square of the difference is made [5]. We compute the per-pixel color difference for each 3D model where the reference image is produced by $8 \times 8S$,

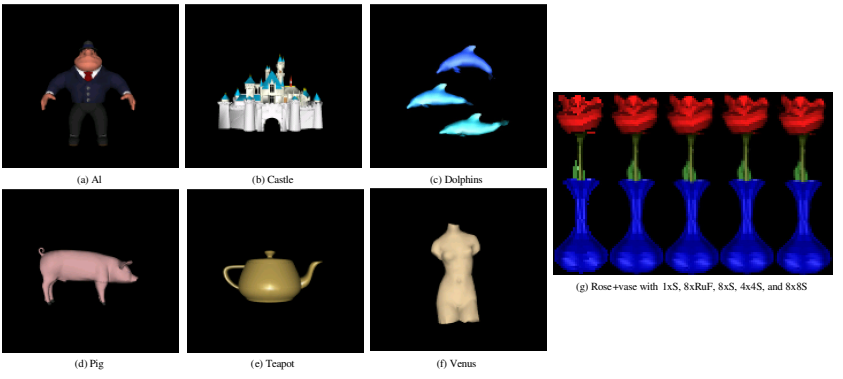

Fig. 5. The final images for each 3D model

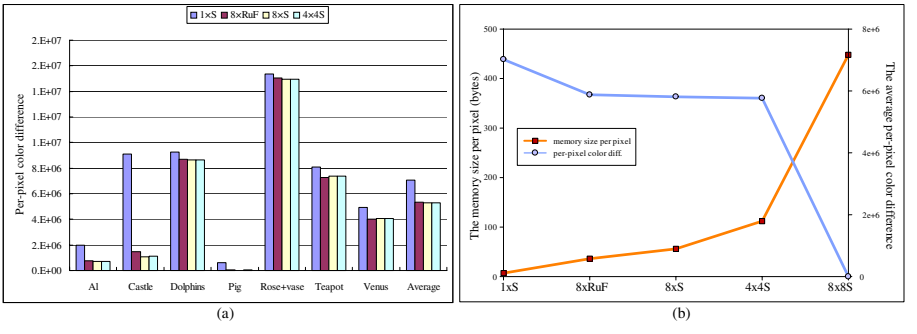

Fig. 6. Performance of color difference and memory size.

it is regarded as to be an ideal image, and each test image is produced by $1 \times S$, $8 \times S$, $8 \times RuF$ and $4 \times 4S$, respectively.

Figure 6(a) shows the results of the per-pixel color difference for each 3D dataset with various sample points. As can be seen from the results, the per-pixel color difference becomes to be smaller as the number of sampling points increase. Moreover, the final images of $8 \times RuF$ are almost as good quality as $8 \times S$; both of them have the same number of sample points.

4.2 Trade-Off Between Image Quality and Memory Requirement

In order to show the cost efficiency of the proposed architecture, two graphs for memory size per pixel and per-pixel color difference, respectively, are plotted together in Figure 6(b). The per-pixel color difference between $8 \times RuF$ and $8 \times S$ is 1.3% but the memory size per pixel is 35.7%. That is, our approach provides almost as good quality as supersampling with a less hardware cost.

4.3 Memory Bandwidth Requirement

Figure 7 shows the memory bandwidth requirements where two bar graphs for supersampling (left) and the proposed scheme (right) are plotted as a pair for each model. Arbitrary scenes for each 3D model are produced with both methods where 8 sparse sample points are used. As shown in the results, the proposed architecture can reduce the memory bandwidth requirement by 53.6% $\sim$ 75.5% for Castle and Rose+vase. The internal bandwidth is required for pixel processing between the graphics pipeline and the frame buffer (includes the RuF-buffer and mask-buffer). The external bandwidth is for swapping the front and back buffer or for average-down filtering. In supersampling, the external bandwidth dominates the memory bandwidth requirement. In other words, it implies that the screen-size color buffer is very efficient in reducing the bandwidth requirement since it does not require the overhead for average-down filtering process.

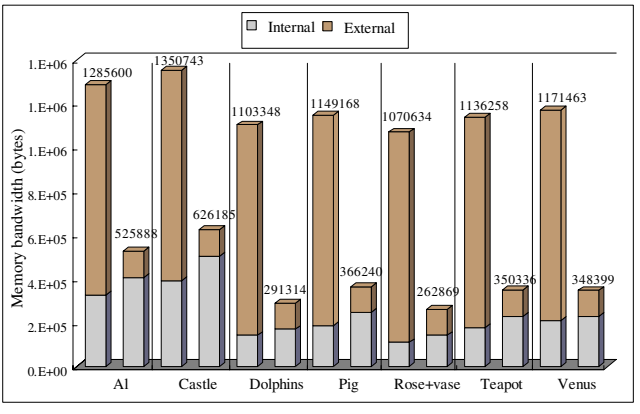

Fig. 7. The memory bandwidth requirement

5 Conclusion

In this paper, we present a graphics hardware system to implement supersampling in cost-effective manner. For hardware-implementation aspect, our graphics architecture uses same programming model as in Z-buffer algorithm for the hidden surface removal and adds only small additions to the conventional rendering process such as mask comparison and composite. In addition, mask comparison and composite can be simply processed with bitwise operations. In the color manipulation, computing the color contribution of a fragment can be processed through look-up tables, each entry of which holds the predefined floating point number divided by the number of sample points.

To provide an indication of the performance in terms of cost-effective full scene antialiasing, the results of memory requirement, bandwidth requirement,

Table 4. Summary of performance

	Memory size	Memory bandwidth	per-pixel color diff.
$8 \times S$	56 Bytes	1181030 Bytes	1338250
$8 \times RuF$	35 Bytes	395890 Bytes	1336154
Reduction ratio	35.7%	67.1%	1.3% (difference)

and per-pixel color difference are shown in Table 4 when 8 sparse sample points are used. It shows that the proposed architecture can reduce the memory size and the memory bandwidth size by 35.7% and by 67.1% with a slight color difference of 1.3%, compared with the conventional supersampling. As shown in the results, the proposed architecture can efficiently render the high quality scene with an economic hardware cost. Moreover, the simplicity of rendering process for our scheme allows us to have fast rendering through well-defined pipeline with a single pass.

Acknowledgment. The authors are grateful to the anonymous reviewers of the earlier version of this paper, whose incisive comments helped improve the presentation. This work is supported by the NRL-Fund from the Ministry of Science & Technology of Korea.

References

1. Aila, F., Miettine, V., Nord, P.: Delay Streams for Graphics Hardware. ACM Transactions on Graphics **22** (2003) 792–800
2. Akeley, K.: RealityEngine graphics. Computer Graphics (SIGGRAPH 93) **27** (1993) 109–116
3. Carpenter, L.: The A-buffer: an Antialiased Hidden Surface Method. Computer Graphics (SIGGRAPH 84) **18** (1984) 103–108
4. Haeberli, P.E., Akeley, K.: The Accumulation Buffer: Hardware Support for High-Quality Rendering. Computer Graphics (SIGGRAPH 90) **24** (1990) 309–318
5. Jouppi, N.P., Chang, C.F.: Z^3: an Economical Hardware Technique for High-quality Antialiasing and Transparency. In Proceeding of Graphics hardware (1993) 85–93
6. Lee, J.A., Kim, L.S.: Single-Pass Full-Screen Hardware Accelerated Antialiasing. In Proceeding of Graphics hardware (2000) 67–75
7. The Mesa 3D Graphics Library. `http://www.mesa3d.org`
8. Watt, A.: 3D Computer Graphics. Third Edition (2000) Addison-Wesley
9. Wittenbrink, C.M.: R-Buffer: A Pointless A-Buffer Hardware Architecture. In Proceeding of Graphics hardware (2001) 73–80