Title	An anti-aliasing method for parallel rendering
Author(s)	Lin, WS; Lau, RWH; Lin, X; Cheung, PYS
Citation	The 1998 International Conference on Computer Graphics, Hannover, Germany, 22-26 June 1998. In Computer Graphics International Proceedings, 1998, p. 228-235
Issued Date	1998
URL	http://hdl.handle.net/10722/46067
Rights	©1998 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

An Anti-Aliasing Method for Parallel Rendering

Sam Lin Rynson W.H. Lau* Xiaola Lin P.Y.S. Cheung

Department of Electrical and Electronic Engineering,
The University of Hong Kong, Hong Kong
E-mail: wslin@eee.hku.hk

*Department of Computer Science,
City University of Hong Kong, Hong Kong
E-mail: rynson@cs.cityu.edu.hk

Abstract

This paper describes a parallel rendering method based on the adaptive supersampling technique to produce anti-aliased images with minimal memory consumption. Unlike traditional supersampling methods, this one does not supersample every pixel, but only those edge pixels. In this paper, we consider various strategies to reduce the memory consumption in order for the method to be applicable in situations where limited or fixed amount of pre-allocated memory is available. This is a very important issue, especially in parallel rendering. We have implemented our algorithm on a parallel machine based on the message-passing model. Towards the end of the paper, we present some experimental results on the memory usage and the performance of the method.

1. Introduction

Most graphics applications nowadays demand both high level of interactivity and high quality of output images. Applications such as computer games, computer-aided design and scientific visualization may expect the hardware system to be able to render anti-aliased images in a high frame rate, but most anti-aliasing methods developed consume a large amount of memory and are computationally very expensive. Recently, we presented a scan-conversion method called the *Adaptive Supersampling Method* [11], with some initial results. It is based on the z-buffer method, and requires minimal extra computational and memory costs to do anti-aliasing. The method is simple and suitable for hardware implementation or for running on a parallel machine. In this paper, we present an enhanced algorithm, which can further reduce the memory requirement when generating anti-aliased images. With this enhancement, the adaptive supersampling method could have better memory management so that it can be run on systems with limited memory. We have targeted to implement this algorithm under the hardware-based or software-based parallel system, which may provide a promising performance for rendering complex anti-aliased images.

The rest of the paper is organized as follows. In section 2, we review traditional image rendering methods and anti-aliasing techniques. In section 3, we summarize our original adaptive supersampling algorithm. In section 4, we present two techniques for memory saving and management. In section 5, we briefly discuss various parallel architectures and different parallel algorithms, and suggest one which may give the best performance. We also describe our parallelized algorithm implemented using the *Message Passing Interface*, MPI. Finally, in section 6, we discuss the overall memory usage and the performance of our algorithm on parallel architectures.

2. Image Rendering Methods

Image generation based on the original z-buffer algorithm [4] requires only the color value and the depth value of the closest object at each pixel. Because of the point sampling nature of the z-buffer method, the images generated usually have aliases. To solve this problem means that we need to solve the visibility problem in subpixel level. This requires the calculation of the visible area of each polygon at each pixel.

There are many rendering methods proposed to solve the aliasing problem. The supersampling z-buffer method [7] supersamples the scene and then filters the supersampled image down into the output resolution. The limitations of this method are that it requires a lot of memory to store the supersampled image and high computational cost to generate the image. The advantages are that it is a simple extension of the z-buffer method and hence it can be implemented in hardware without too much additional effort [1, 13]. The RealityEngine [2] is a hardware implementation of this kind, but the color and depth values are only sampled once per pixel to improve

the performance.

Another well-known anti-aliasing method is called the A-buffer method [3]. This method basically breaks polygons into pixel fragments. The visible fragments are accumulated in a temporary buffer for hidden surface removal and anti-aliasing. Memory usage of this method depends on the complexity of the scene and there is no theoretical upper limit. As such, this method normally requires run-time memory allocation, which makes the implementation of the A-buffer algorithm in hardware difficult. A rare example can be found in [20], which employs a multiple-pass algorithm to perform front-to-back hidden surface removal and shading. In order to reduce the memory cost and the required memory bandwidth, the image is partitioned into 16x32 pixel blocks. Each block is rendered independently one after another. Hence a double-buffered z-buffer and an image buffer can be stored on-chip.

In [16], an adaptive sampling method is proposed. Whenever a pixel is covered by one or more polygons, a linked-list of these polygons is created. These polygons are clipped, and extra samples, or *oversamples*, will be taken in rasterization. They described this as a pixel-level virtual camera. However, this algorithm creates a lot of linked-lists and involves a lot of clipping operations, which greatly affect their performance.

3. Adaptive Supersampling Method

Because the traditional supersampling method allocates memory to store the depth and color values of each subpixel as shown in Figure 1(a), a lot of memory is needed and the actual memory usage is directly proportional to the subpixel resolution. Obviously, a lot of supersampled pixels are not use, because the aliasing problem occurs largely around polygon edges and lines where surfaces intersect each other. As these edge pixels contribute only to a small portion of the total number of pixels in the image (not exceeding 20% for the complex test images used in our initial experiment [11]), our idea here is to supersample a polygon only when we need. This may result in a considerable amount of saving in both memory and processing time compared to the supersampling method.

To achieve this, when scan-converting a polygon, if the polygon covers the whole pixel, we sample the polygon once only. When the polygon partially covers the pixel, we perform a supersampling of the polygon within the pixel region. We do this by calculating where the polygon edge crosses the boundaries of the pixel and using the intersection information to form a bitmask index. A bitmask index stores the enter and exit positions of the polygon edge. This bitmask index is then used to access an appropriate bitmask from a pre-computed bitmask table [10]. The bitmask can be used to set the subpixel coverage. In order for the buffer to handle the information generated from either of the two sampling resolutions, we employ a technique similar to the one used in the A-buffer method [3]. A standard 2D buffer is used for normal z-buffer scan-conversion (one sample per pixel). When a polygon edge is encountered, a larger memory block is allocated to the pixel for storing the subpixel samples. Figure 1(b) shows the distribution of memory blocks in an image.

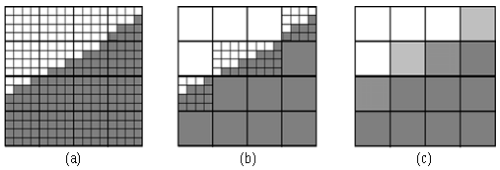


Figure 1. Images rendered with: (a) the traditional supersampling method, and (b) the adaptive supersampling method, (c) the resultant image after filtered the supersampled pixels down from (a) or (b).

Unlike the A-buffer method, there is at most one memory block allocated to a pixel no matter how many polygon edges are found in the pixel. In the A-buffer method, there are two dimensions of uncertainty. The first one is the number of edge pixels in an image and the second one is the number of fragments in each edge pixels. The new method reduces it to one dimension - the number of edge pixels in an image. There is no need to traverse through a possible long list of fragments as in the A-buffer method. The new method greatly simplifies the algorithm and makes it easier to be implemented into hardware. In addition, our method, like the supersampling method, can resolve surface intersection in subpixel level while the A-buffer method deals with the problem using approximation.

The new buffer uses two data structures as in our orignal paper [11], but there are some changes in the definitions of the two data structures. The major one is **Subpixel**. In the original paper, **Subpixel** was defined to contain enough memory to store the color and depth values of all subpixels. Here, it is redefined to store the color and depth value of a single subpixel, so that the number of subpixels in a pixel can be dynamically adjusted according to the amount of memory available in the memory pool. This will be described in details in the next section. The definitions of the two data structures are as follows:

```
typedef struct {
   Color rgb;
   Boolean SuperPixel;
   union {
      int z;
      Subpixel * pblock;
   } zOrpblock;
   short dxz, dyz;
} Pixel;

typedef struct {
   Color rgb;
   int z;
} Subpixel;
```

Pixel is the basic element of a 2D pixel buffer. If a polygon completely covers a pixel, the **Pixel** element is used in a similar way as in the traditional z-buffer method. If a polygon partially covers a pixel, the memory location for storing the z value is used as a pointer instead. **SuperPixel** indicates if the pixel has been supersampled. If it is set to 0, the pixel has been sampled once only; if it is set to 1, the pixel is a supersampled pixel. The attributes **dxz** and **dyz** are used to store the depth increments of the polygon in the x and y directions respectively. These two values are calculated once for each polygon and are used to generate the depth values for each subpixel so that hidden surface removal may be done in subpixel level.

4. Memory Management

Although our method could greatly reduce the memory consumption when compared with the traditional supersampling method, we target to further reduce the memory consumption in order to generate high resolution images with minimal memory. We propose two techniques to reduce the memory usage, the memory adaptation technique and the memory reclaiming technique. The memory adaptation technique concerns how to change the sampling frequency in the rendering process while the memory reclaiming technique concerns how to reclaim some of the **Subpixel** cells in an image.

4.1. Memory Adaptation Technique

In our algorithm, we employ a technique to allow the sampling frequency in a supersampled pixel to be changed according to the availability of memory resources and the statistical information of the previous frame. To ensure that all 100% of pixels can be supersampled without the need for dynamically allocated memory, we preallocate enough memory for supersampling the whole image using the lowest subpixel resolution, i.e. 2x2. We use three different subpixel resolutions for the edge pixels (4x4, 3x3, and 2x2), depending on the total number of edge pixels in the image. If it is small, we can supersample the edge pixels using a higher subpixel resolution, and vice versa.

In our method, the preallocated memory can be used to supersample 25% of the total number of pixels in the image at a subpixel resolution of 4x4. When rendering the first frame, we set the subpixel resolution to the highest one, i.e. 4x4. Whenever a new edge pixel is to be created, 16 **Subpixel** cells will be allocated to the pixel. If the memory usage in this frame exceeds the preallocated amount, no further supersampling will be done. Because of frame-to-frame coherence, the memory usage of the next frame is expected to be similar to that of the current frame. Based on the actual total number of edge pixels found in the current frame, the algorithm will decrease the subpixel resolution of the next frame from 4x4 to 3x3 as shown in Figure 2. This will allow up to 44% of the pixels in the image to be supersampled. We may further reduce the subpixel resolution to 2x2, which will then allow up to 100% of the pixels to be supersampled. The decision of using which subpixel resolution for rendering the next frame depends on the memory consumption of the current frame.

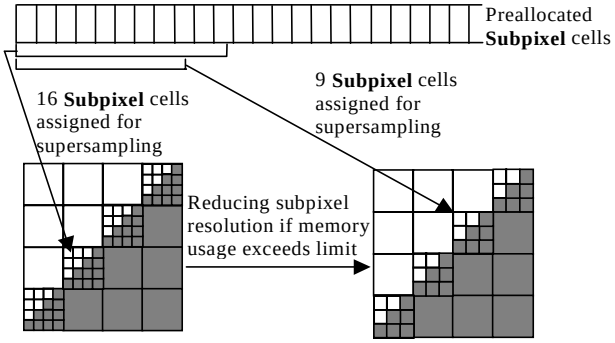


Figure 2. The subpixel resolution decreases when the memory usage of the previous frame exceeded the preallocated amount.

4.2. Memory Reclaiming Technique

The second technique to reduce memory consumption is to reclaim those unnecessary **Subpixel** cells. When scan-converting a polygon edge, a lot of edge pixels may be created. Some edge pixels may not represent the real edges of the object, as two adjacent polygon edge fragments may together cover the whole pixel as shown in Figure 3.

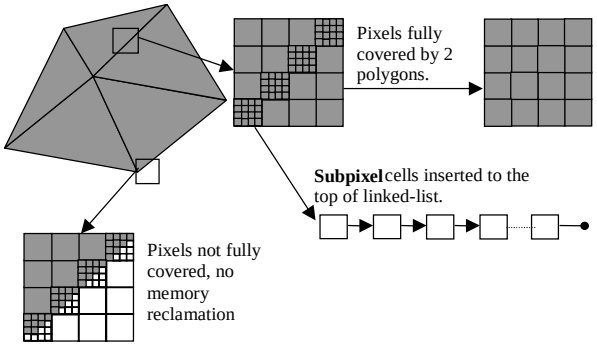


Figure 3. When all the subpixels are filled with the same color and similar depth values, Subpixel cells can be reclaimed.

If an object is composed of a large number of polygons, it will have a lot of “internal” edges, and a lot of supersampled pixels may be resulted. In order to release these **Subpixel** cells, we have to keep track of the supersampled pixels to see if all the subpixels have the same color and similar depth values. Polygons sharing a common edge will result in the same color values along

the edge pixels. The depth value, however, may have some variations due to the interpolation of **dxz** and **dyz** of the two polygons. Hence, when computing the subpixel depth values, if the difference between the maximum and minimum depth values is smaller than 1 unit, we may consider the pixel fully covered by two adjacent polygons and reclaim all the **Subpixel** cells allocated to the pixel. The reclaimed **Subpixel** cells will be inserted to a linked-list for later use. Thus, we can reuse preallocated memory and avoid memory fragmentation during the rendering of an image. Although this technique imposes an overhead on the algorithm, as a result of more comparisons, a large amount of supersampled pixels can be reclaimed. We will discuss some results of this technique in section 7.

5. Parallel Rendering

In image rendering, there are two major phases that account for most of the computational cost.

- *Transformation phase*: object transformation and clipping, and
- *Rasterization phase*: rasterization and anti-aliasing.

The parallelism in the transformation phase is generally called *object parallelism*, in which geometric primitives are sent to different processors for transformation and clipping. The parallelism in the rasterization phase is called *image* or *pixel parallelism* [5, 6], in which the screen is divided into regions and each processor is responsible for one or more of the regions.

5.1. Parallel Rendering Systems

Parallel rendering is not a new issue. The design and implementation of either hardware-based systems or software-based systems can be found in many literatures [5, 18]. One way to parallelize the rendering process is to implement the rendering pipeline directly into a parallel hardware system [9]. This approach has been very successful in producing very high performance rendering systems. However, these systems can only perform graphics operations and are not suitable for general-purpose computations.

Another way to parallelize the rendering process is by implementing the rendering algorithms on a massively parallel architecture [18]. Theoharis in [17] explored the algorithms on SIMD and hybrid SIMD/MIMD systems. Whitman also evaluated several algorithms for MIMD shared memory systems [18]. In these systems, the processors are not specifically designed for graphics operations, and the communication between processes often has bandwidth limitations. Most of the research cited above, therefore, tries to cope with these limitations in order to maximize the parallelism and the potential of the underlying hardware. Although a massively parallel machine may have a large number of processors, each processor usually contains a very limited amount of memory. If anti-aliasing is used, the memory requirement may increase dramatically. Thus, memory management on this kind of machine is important. Whitman in [18] has derived different caching methods to minimize the cost of remote memory access.

Sometimes, a network of workstations or personal computers can also be configured as a parallel rendering system. Although most of these computers are inexpensive, the aggregate processing power and memory capacity can be quite high. However, due to the high communication latency and low network bandwidth, the performance of the resulting rendering system is usually low. To overcome these limitations, the rendering algorithm should break the jobs with a larger granularity, and sometimes, replicate the object database in order to minimize the communication between processors.

From the above discussions, we may have noticed that many systems suffer from the degradation of performance when applying anti-aliasing due to either the limited amount of memory available to each processor or the amount of information needed to be sent through the network. In [16], a parallel anti-aliasing method was described, although no results were presented. In the current stage, we are focusing on measuring the memory usage and its distribution among the processors. We would like to determine the minimum amount of memory that we should preallocate to each processor, and the strategies for handling memory usage when it exceeds the preallocated amount. Apart from the issue of different implementation platforms, the design of the algorithm is also important in developing an efficient parallel rendering system.

5.2. Sorting Algorithms

As a polygon may be projected anywhere on the screen, polygons rendering may be viewed as a sorting problem on the screen space. Hence, the selection of different sorting methods will affect the technique used for data decomposition and the frequency of data communication, which in turn affect the performance of the parallel algorithm. Molnar et al. in [14] have a detailed description of different kinds of sorting methods. The choice of sorting methods leads to taxonomy of different architectures. They are *sort-first*, *sort-middle* and *sort-last*.

In *sort-first*, each processor is assigned a portion of the screen to render. Before the polygons are distributed, they are first transformed to determine where they should be transferred. The corresponding processors will then perform the remaining operations in the transformation phase and the complete rasterization phase. Sort-first will minimize the communication between processors if the objects in the image are static or moving very slowly. This is because there is no need to redistribute the transformed polygons to other processors. However, sort-first induces to load imbalance as polygons may all fall into a single region. In addition, more data would be replicated among

the processors if most objects are overlapped in many regions. Mueller implemented a sort-first rendering system in [15].

In *sort-middle*, polygons are distributed evenly among the processors without concerning the screen location of the polygons. After the transformation phase, the polygons need to be redistributed among processors depending on which region of the screen they fall in. However, it is not necessary to transfer the whole polygon structure to the other processors. Only information such as screen coordinates of the polygon, and color of each vertex is needed. This can minimize the amount of data transferred through the network. Rasterization takes place after the information related to the polygons has reached the corresponding processors. Sort-middle is a natural architecture because it performs redistribution between the transformation and rasterization phases. We can design different algorithms for object parallelism and image parallelism independently. There are many examples on this type of architecture [2, 6, 8].

In *sort-last*, polygons are distributed among the processors. Transformation and rasterization are carried out on the same processor. Hence, polygons may fall into different positions of the screen. The images from all processors are then collected and combined together to form an output image [12]. Since the whole image generated by each processor needs to be transferred to one of the processors to do image composition, the amount of data transferred can be very high. The situation could be worse if traditional anti-aliasing method is applied. Additional samples of the image will increase the amount of data to be transferred by several times. Our adaptive anti-aliasing algorithm would obviously minimize the amount of data to be transferred during the image composition stage, and improve the performance significantly.

5.3. Load Balancing

Screen subdivision is one of the image parallelism methods by which primitives are divided according to their projected screen positions. In most cases, processors are assigned to handle a group of subdivided screen regions. The transformed geometric primitives are transferred to the corresponding processors for rasterization. Therefore, the strategy used to subdivide the screen would affect the method for task decomposition, and hence the load balancing between the processors. We can classify the strategies for handling load balancing as either *static* or *dynamic*.

5.3.1. Static load balancing

Static load balancing schemes rely on fixed screen partitioning to distribute polygon primitives to specific regions of the screen, and are therefore totally affected by the granularity ratio. Granularity refers to the degree of data decomposition. A fine-grained decomposition breaks the dataset into smaller units, while coarse-grained decomposition breaks the dataset into larger units. Experiments show that coarse-grained decomposition usually results in poor load balancing, while fine-grained decomposition generally results in better load balancing but at the time higher overheads for tasks scheduling, communication, redundant calculations and more data replication among processors. The overheads are more significant on distributed memory systems as mentioned earlier. The granularity ratio has been studied in [8] and [18]. Hence, using the coarsest granularity that allows reasonable load balancing may be the best way for achieving good parallel timings in distributed memory architectures. However, the optimal granularity ratio is different for different architectures. Figure 4 shows our method of subdivision in which the screen is subdivided into small square regions, each region consists of 64x64 pixels. Regions of the same pattern are assigned to the same processor for scan-conversion and display.

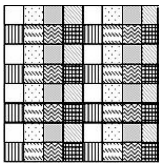


Figure 4. Regions of the screen with the same pattern are assigned to the same processor.

5.3.2. Dynamic Load Balancing

Two major approaches are currently used in *dynamic load balancing*, the demand-driven approach and task adaptive approach [19]. The demand-driven approach decomposes the problem domain into smaller independent tasks, and then assigns them to different processors to finish the tasks. Once a processor has finished its current task, another task is assigned to it until all the tasks are completed. The task adaptive approach decomposes the problem into a relatively small number of coarse-gained tasks, which are then assigned to different processors. If a processor has finished all its tasks, it communicates with another processor with the largest remaining workload, and then helps it to finish half of its remaining job. In this approach, additional costs including communications among processors and subdivision calculations would increase the overheads. Because the cost of remote memory references makes dynamic task assignment, data migration, and maintaining global status information more expensive, this approach is not suitable for use in message-passing systems. Therefore, most of the algorithms using dynamic load balancing strategy are implemented on shared-memory architectures.

6. The Message-Passing Parallel Renderer

With the sort-first model, it is possible that most polygons are assigned to one processor in the transformation and rasterization phases. Thus, this processor will be heavily loaded while others stay idle. The sort-middle model also suffers from load imbalance during the rasterization phase. But it does not have the problem in the transformation phase because the polygons are evenly distributed among the processors for transformation. For the sort-last model, the transformed polygons may have various sizes, and the size of job may vary significantly. We have developed a renderer to test the memory distribution when rendering an anti-aliased image using our adaptive method. Although we can foresee the improvement of the sort-last model used in our method, the sort-middle model is simple and easy to break the rendering pipeline into 2 stages. It also provides a better load balancing than the other two models. In view of this, we have adopted the sort-middle model to implement our renderer using the message-passing paradigm.

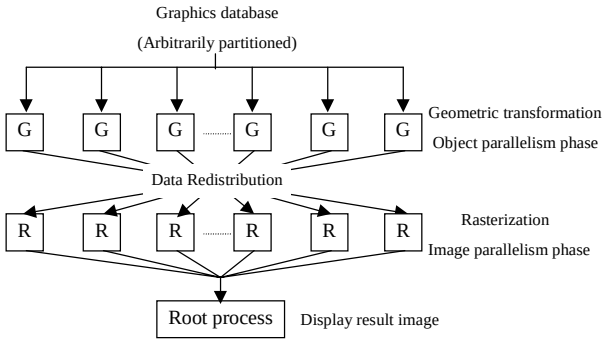


Figure 5. Overall algorithm of the message-passing renderer.

The renderer is implemented on an SGI Power Challenge with 8 processors using the library of *message passing interface*, MPI. The structure of the algorithm is shown in Figure 5. At first, objects in the database are divided into geometric primitives in the form of polygons. They are distributed evenly among the processors for transformation, back-face culling, and clipping. The transformed polygons may fall into different regions of the screen, and hence redistribution of data is necessary. In order to lower the volume of transferred data, only screen coordinates, and color values of the transformed polygons are sent to the corresponding processor, where rasterization takes place.

In the image parallelism stage, we adopt the static load balancing technique to minimize the communication between processors. The strategy for partitioning the screen has been described in section 5. A root process is responsible for collecting resultant regional images and displaying them.

7. Results and Discussions

In this section, we discuss the memory consumption, the distribution of memory, and the load balancing between processors of the new adaptive method.

7.1. Memory Usage

We have tested some images of various complexities. From our experiments, majority of the complex images contain no more than 20% of edge pixels. Figure 6 shows the percentages of the supersampled pixels when rendering Figure 8(a) and 8(b). Figure 8(a) contains the letter 'B' comprising of roughly 800 large polygons. The object was made to rotate randomly on the screen.

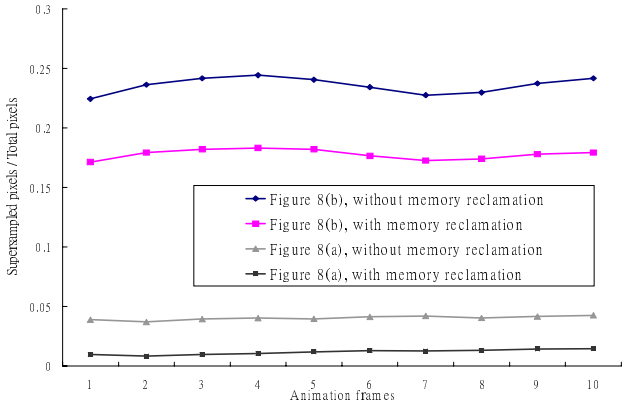


Figure 6. Memory usage in Figure 8(a) and 8(b), before and after memory reclamation.

The total number of supersampled pixels used without memory reclamation was less than 5% of the total number of pixels in the image. If we render the image with 512x512 pixels and 4x4 subpixels resolution, the 2D array of **Pixel** cells consumes about 3.14MB of memory and the **Subpixel** cells consume about 1.47MB. In other words, the whole image consumes about 4.61MB of memory. When rendering the same image using memory reclamation technique, the number of supersampled pixels drops to 1.3% of the total number of pixels in the image. Thus, only about 3.52MB of memory is consumed, and up to a maximum of 74% supersampled pixels were reclaimed. To render Figure 8(b), 10.14MB of memory was used and about 24% of the pixels were supersampled. By using the memory reclamation technique, about 28% of the supersampled pixels were reclaimed. Thus, the percentage of supersampled pixels decreased to about 17%. Comparing with the traditional supersampling method where 29.4MB of memory is needed for rendering an image with the same pixel and subpixel resolutions, tremendous amount of memory has been saved. Figure 8(c) shows a test image of the space shuttle. The number of supersampled pixels created was also below 5% of the

total number of pixels in the image.

Figure 8(d) shows an extreme test case of a very complex scene. The test file contains more than 3,000 tetrahedral objects randomly distributed on the image. With our memory reclamation technique, the total number of **Subpixel** cells used to render the image was below 50%. In this situation, the subpixel resolution has been set to 2x2. However, such complex scene is not likely to appear in most applications because polygons are normally clustered to form objects, instead of randomly distributed around the screen resulting in a lot of edge pixels being created.

In some applications, the scene complexity may vary greatly. In order to handle images with any complexity, we need to preallocate enough memory for supersampling all pixels of the image at a 2x2 subpixels resolution. In such situation, we need about 10.44MB of memory for rendering an image of 512x512 pixels. Of course, if we are certain that the application will never have to handle scenes with 100% of supersampled pixels, we may reduce the amount of preallocated memory accordingly.

In some cases, consecutive images may not preserve the frame-to-frame coherence, e.g. a rapid change of the viewing direction, the memory adaptation technique may give a wrong prediction. Thus, some edge pixels may not be supersampled when all the preallocated memory is used up. However, this problem normally occurs only for a very short moment, and our adaptive algorithm will correct the situation in the next frame.

Figure 7 shows the maximum and minimum memory usage among the 8 processors when rendering Figure 8(b). The distributions of memory among the processors were very close, which means that every processor handled similar number of supersampled pixels during the rendering process. This actually reflects the situation of load balancing. The average memory usage of each processor is about 0.98MB.

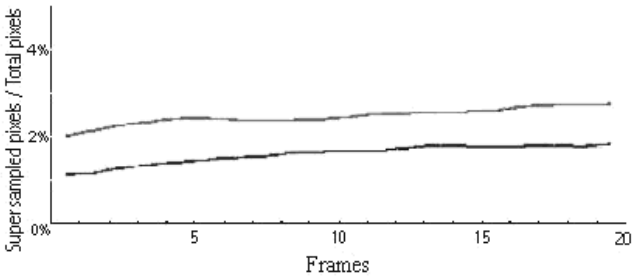


Figure 7. The maximum and minimum amount of memory usage of the 8 processors during 20 frames of animation on Figure 8(b).

7.2. Performance Analysis

Table 1 shows the performance results of rendering Figure 8(a). The image contains large polygons of over 50x50 pixels each. The timings on each phase of the rendering process for 20 frames were measured. It can be seen that the workload of the transformation phase was evenly distributed as all processors spent the same amount of time on the transformation. From our measurements, the renderer can approximately process up to 20-30K large anti-aliased polygons per second.

	Min. Time	Max. Time
Transformation Phase	0.0002s	0.0002s
Data redistribution	0.0006s	0.0120s
Rasterization Phase	0.0170s	0.0300s

Table 1. The minimum and maximum times spent on each stage of the message-passing renderer when rendering Figure 8(a).

8. Conclusions

From the results of our implementation, although the traditional supersampling method solves the aliasing problem, it increases the rendering time and memory usage dramatically. The new adaptive supersampling method requires less memory and has a much higher performance. Unlike the A-buffer method, the new method never has to traverse a possibly long list of fragments. Here, we have presented two memory saving techniques to further reduce the memory usage of our adaptive method. We have also investigated the parallelization of our method.

Toward the end of the paper, we have discussed the memory usage and the performance of our adaptive anti-aliasing method when running in parallel architectures. As a future work, we are currently considering the possibility of implementing the method in hardware.

References

[1] K. Akeley and T. Jermoluk. High-Performance Polygon Rendering. *ACM Computer Graphics*, 22(4):239-246, August 1988.

[2] K. Akeley. RealityEngine Graphics. *ACM Computer Graphics*:109-116, August 1993.

[3] L. Carpenter. The A-buffer, an Antialiased Hidden Surface Method. *ACM Computer Graphics*, 18(3):103-108, July 1984.

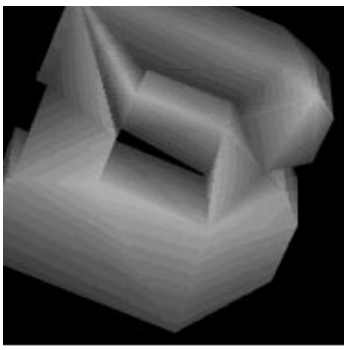
[4] E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces.* Ph.D. Dissertation, Computer Science Department, University of Utah, 1974.

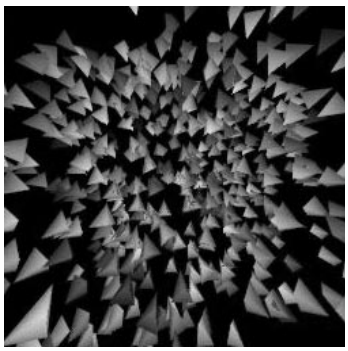
[5] T. Crockett. *Parallel Rendering*. Technical Report 95-31, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, 1995.

[6] T. Crockett and T. Orloff. A MIMD Rendering Algorithm for Distributed Memory Architectures. *ACM Parallel Rendering Symposium*:35-42, 1993.

[7] F. Crow. The Aliasing Problem in Computer-Generated Shaded Images. *Communications of the ACM*, 20(11):799-805, November 1977.

[8] D. Ellsworth. A New Algorithm for Interactive Graphics on Multicomputers. *IEEE Computer Graphics and Applications*, 14(4):33-40, July 1994.
[9] H. Fuchs. VLSI for Graphics. *Techniques for Computer Graphics, Springer-Verlag*:281-294, 1987.
[10] R.W.H. Lau and N Wiseman. Accurate Image Generation and Interactive Image Editing with the A-buffer. *Proceedings of EuroGraphics '92*, II(3):279-288, September 1992.
[11] R.W.H. Lau. An Adaptive Supersampling Method. *Image Analysis Applications and Computer Graphics, LNCS 1024, Springer-Verlag*:205-214, December 1995.
[12] T.Y. Lee, C.S. Raghavendra and J.N. Nicholas. Image Composition Methods for Sort-Last Polygon Rendering on 2D Mesh Architectures. *ACM Parallel Rendering Symposium*:55-62, 1995.
[13] S. Molnar, J. Eyles and J. Poulton. PixelFlow: High-Speed Rendering Using Image Compo-sition. *ACM Computer Graphics*, 26(2):231-340, July 1992.
[14] S. Molnar, M. Cox, D. Ellsworth and H. Fushs. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications*, 14(4):23-31, July 1994.
[15] C. Mueller. The Sort-First Rendering Architecture for High-Performance Graphics. *ACM Symposium on Interactive 3D Graphics*:75-84, 1995.
[16] F.V. Reeth, R. Welter and E. Flerackers. Virtual Camera Oversampling: A New Parallel Anti-Aliasing Method for Z-Buffer Algorithms. *CG International'90*:241-254, 1990.
[17] T. Theoharis. *Algorithms for Parallel Polygon Rendering.* LNCS 373, Springer-Verlag, Berlin, 1989.
[18] S. Whitman. *Multiprocessor Methods for Computer Graphics Rendering.* AK Peters, 1992.
[19] S. Whitman. A Task-Adaptive Parallel Graphics Renderer. *IEEE Computer Graphics and Applications*, 14(4):41-48, July 1994.
[20] S. Winner, M. Kelly, B. Pease, B. Rivard and A. Yen. Hardware Accelerated Rendering Of Antialiasing Using A Modified A-buffer Algorithm. *ACM Computer Graphics*:307-316, 1997.


(a)


(b)


(c)

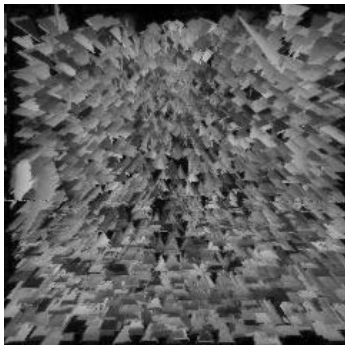

(d)

Figure 8. (a) shows the letter 'B', containing about 800 very large polygons. (b) shows 500 tetrahedral objects scattered on the screen and rotate randomly. (c) shows the output image of a shuttle. (d) shows 3000 tetrahedral objects scatter on screen; about 50% of the whole image are supersampled pixels, and the subpixel resolution is lowered to increase the availability of memory.