Adaptive Parallel Rendering on Multiprocessors and Workstation Clusters

Wai-Sum Lin, Rynson W.H. Lau, *Member*, *IEEE Computer Society*, Kai Hwang, *Fellow*, *IEEE*, Xiaola Lin, and Paul Y.S. Cheung, *Senior Member*, *IEEE*

Abstract—This paper presents the design and performance of a new parallel graphics renderer for 3D images. This renderer is based on an adaptive supersampling approach that works for time/space-efficient execution on two classes of parallel computers. Our rendering scheme takes subpixel supersamples only along polygon edges. This leads to a significant reduction in rendering time and in buffer memory requirements. Furthermore, we offer a balanced rasterization of all transformed polygons. Experimental results prove these advantages on both a shared-memory SGI multiprocessor server and a Unix cluster of Sun workstations. We reveal performance effects of the new rendering scheme on subpixel resolution, polygon number, scene complexity, and memory requirements. The balanced parallel renderer demonstrates scalable performance with respect to increase in graphic complexity and in machine size. Our parallel renderer outperforms Crow's scheme in benchmark experiments performed. The improvements are made in three fronts: 1) reduction in rendering time, 2) higher efficiency with balanced workload, and 3) adaptive to available buffer memory size. The balanced renderer can be more cost-effectively embedded within many 3D graphics algorithms, such as those for edge smoothing and 3D visualization. Our parallel renderer is MPI-coded, offering high portability and cross-platform performance. These advantages can greatly improve the QoS in 3D imaging and in real-time interactive graphics.

Index Terms—Computer graphics, parallel rendering, supersampling, polygon rasterization, symmetric multiprocessors, cluster of workstations, MPI programming, load balancing, speedup and efficiency, and scalable performance.

1 Introduction

To deliver the desired *quality of service* (QoS) in graphics or video applications, the parallelization of image or graphics rendering system is very much in demand. Applications such as scientific visualization are CPU-intensive and demand high-performance graphics rendering of complex 3D scenes. Although hardware graphics accelerator chips rapidly appear in modern computer systems [2], [14], [38], [42], software means to render 3D images are still very attractive on parallel computers, from both application flexibility and system efficiency viewpoints [29], [7].

In recent years, a surge of interest in building parallel renderers on MIMD (*multiple instruction stream and multiple data streams*) shared-memory multiprocessors [37], [39], message-passing multicomputers [5], [11], [24], and workstation clusters [15], [13], [10] has occurred. Others even attempted to use SIMD (*single instruction stream and multiple data streams*) graphics accelerators [43], [35]. In this paper, we concentrate on parallel polygon rendering on multiprocessors and workstation clusters [18].

This paper presents an *adaptive supersampling* approach to building parallel renderers on a shared-memory multiprocessor or on a cluster of workstations. Traditional rendering is mostly done by sampling at the pixel level. A good example is Catmull's *z-buffer algorithm* [4], which requires storing color and depth values of the closest object at pixel level. However, images produced with a pixel sampling method has the aliasing problem discussed in [39]. To remove aliasing, we need to solve the visibility problem with a refined resolution at the subpixel level.

This requires the calculation of the visible area of each polygon at the subpixel level. Existing hidden surface removal algorithms based on subpixel supersampling can be classified into *fixed-sized buffering* (FSB) methods, which use a fixed sized buffer for image generation, and *variable-sized buffering* (VSB) methods, which use a variable sized buffer for image generation. Examples of the FSB approach started with Crow's *supersampling z-buffer* method [9]. In his method, the scene is supersampled multiple times per pixel and then filtered down to the output resolution.

Crow's method requires a lot of memory and processing time to produce the supersampled image. The advantage, however, is that hardware implementation of the FSB method is relatively simple [2], [28]. Other FSB methods include [1], which clips all polygons against each other to produce a list of visible polygons for scan-conversion, and [16], which samples each pixel multiple times and accumulates the subpixel values to produce the final image. Both methods require less memory but a considerable amount of processing time.

The VSB methods can be implemented with either a span buffer [4], [32], [41], with which polygons are divided into horizontal spans, or a pixel buffer [3], [21], with which

- *S. Lin, X. Lin, and P. Cheung are with the University of Hong Kong, Hong Kong. E-mail: samlin@hkpc.org, {xlin, cheung}@eee.hku.hk.*
- *Rynson Lau is with the City University of Hong Kong. E-mail: rynson@cs.cityu.edu.hk.*
- *K. Hwang is with the Department of Electrical Engineering, University of Southern California, Los Angeles, CA. 90089. E-mail: kaihwang@usc.edu.*

Manuscript received 1 June 2000; revised 9 Sept. 2000; accepted 9 Oct. 2000. For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 111196.

TABLE 1
Notations and Conventions

Notation	Meaning
n	Machine Size (the number of processors)
α	Subpixel resolution
β	Number of polygons in the Scene
γ	Scene complexity (% of edge pixels in scene)
M	Memory Usage (in M Bytes)
R	Rendering Time (in second)
S	Speedup of parallel processors over single processor
E	System Efficiency of a parallel computer
G	Performance Gain of our method over others

polygons are divided into pixel fragments. These horizontal spans or pixel fragments will then be accumulated in the buffer for hidden surface removal and antialiasing. An example of the VSB method is Carpenter's A-buffer method [3]. It accumulates all visible polygon fragments as linked lists in the buffer. After all polygons are rendered, it performs a front-to-back traversal of each list to obtain the pixel color.

As such, memory usage of VSB methods depends on the complexity of the scene. Hence, they require an efficient scheme for runtime memory allocation. Hardware implementation of VSB methods for real-time image generation is usually complex [38], [42]. In [42], a modified A-buffer method was implemented as a chipset. In order to reduce memory cost and memory bandwidth, the image was partitioned into 16×32 pixel blocks. Each block was rendered independently one after another.

In [34], an adaptive sampling method was proposed. Whenever a pixel is covered by one or more polygons, a linked list of these polygons is created. These polygons are clipped based on oversampling. This VSB algorithm creates many linked lists and involves many clipping operations, which severely degrade its speed performance on parallel computers. Many parallel rendering systems suffer from performance degradation when applying antialiasing due to limited memory or too large an image file to send through the network. Reeth et al. [34] has presented a parallel antialiasing method, but no performance results were reported.

This paper presents a new, parallel graphics rendering algorithm, improved from the *adaptive supersampling* method proposed by [22]. This new method has some similarity to the adaptive grid work in computational fluid dynamics. However, their problem environments and mathematical formulations are very different. We prove the advantages of this parallel approach through experimentation on a SGI *symmetric multiprocessor* (SMP) server and on a Unix cluster of 12 Sun workstations at the High Performance Computing Research Laboratory of the University of Hong Kong. The scheme is efficient in time and space on both classes of parallel computers. To benefit the readers, we summarize in Table 1 the notations and conventions used throughout the paper. The Greek letters refer to image properties. Capital letters are reserved for performance measures on parallel computers.

The rest of the paper is organized as follows: In Section 2, we describe the adaptive supersampling method, which is based on the VSB approach. This new method attempts to make a compromise between Crow's supersampling method and Carpenter's A-buffer method. Section 3 presents the algorithmic steps of our parallel image renderer. Section 4 presents a load balancing procedure for successive frames. For the first time, this paper presents two parallel rendering schemes: Crow's renderer based on full supersampling and our new adaptive renderer based on adaptive supersampling. Relative performance of the two parallel renderers is revealed by benchmark experiments. The rendering performance results are presented in Section 5 along with an analysis of the load balancing effects. Section 6 presents memory-reduction techniques and discusses the effects on different supersampling resolutions. Section 7 analyzes several performance effects of our adaptive parallel render, compared with other known parallel renderers. In particular, we reveal the effects of the number of polygons rendered, of the scene complexities, and of the memory demands. Finally, we summarize the contributions of this paper and discuss future directions of research.

2 Basic Concept of Adaptive Supersampling

A *supersampling* method stores the color and depth values at the subpixel level, as illustrated in Fig. 1a. Hence, the size of the buffer is proportional to the subpixel resolution. Since the aliasing problem occurs largely around polygon edges and lines where surfaces intersect, supersampling of nonedge pixels is really unnecessary. With the low percentage of edge pixels in most images, we prefer to take supersamples only at polygon edges, as shown in Fig. 1b.

This idea leads to the *adaptive supersampling* method with a considerable amount of savings in buffer memory and in processing time, compared to Crow's supersampling across all pixels inside and outside of the polygons.

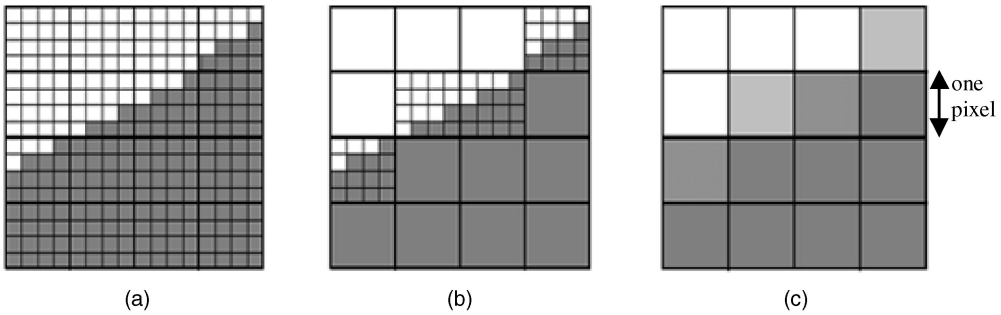


Fig. 1. Image rendered by two different supersampling methods. (a) Crow's method. (b) Lau's adaptive method. (c) Resulting image from both.

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

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

Fig. 2. Data structures for pixel and subpixel buffers.

Fig. 1 shows the difference between Crow's method (Fig. 1a) and Lau's adaptive approach (Fig. 1b). Even though they both produce the same final result (Fig. 1c), our method significantly demonstrates the advantages of shorter rendering time and lower memory requirement.

In our method, when scan converting a polygon, if the polygon covers the whole pixel, we sample the polygon once only. Otherwise, we supersample the polygon within the pixel region. We determine where the polygon edge crosses the pixel boundaries. The entry and exit positions of the polygon edge at the pixel are combined to form a bitmask index. This bitmask index is then used to access an appropriate bitmask from a precomputed bitmask table [21]. The bitmask is used to set the subpixel coverage.

In order for the buffer to handle information generated from different subpixel resolutions, two data structures are used to form the buffer, as shown in Fig. 2. The `Pixel` is the basic element of the 2D pixel buffer. If a polygon completely covers a pixel, the `Pixel` is used in a similar way as in Catmull's z-buffer method. If a polygon partially covers a pixel, the memory location for storing the z value becomes a pointer to an array of `Subpixels`.

If the pixel has only been sampled once, the flag `Supersampled` is set to FALSE; otherwise, it is set to TRUE. Unlike the A-buffer method [3], our method allocates at most one array of `Subpixels` 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 number of edge pixels in an image and the number of fragments in each edge pixel. Our method reduces it to one dimension: the number of edge pixels in an image. There is no need to traverse a possibly long list of fragments. Our method greatly simplifies the algorithm and makes it easier to implement in hardware.

Two buffers, denoted as `dxz` and `dyz`, in the `Pixel` store the depth increments of the visible polygon in x and y directions, respectively. These two values are calculated once for each polygon. They are used to determine the depth value of each subpixel so that hidden surfaces may be removed at subpixel level. Each `Subpixel` structure stores the `rgb` and `z` values of a subpixel. The pixel buffer is initialized so that all `rgb` fields store the background color. All flags, `Supersampled`, are set to FALSE indicating that `zORpblock` contains a depth value initialized with the maximum value. All `dxz` and `dyz` are initially set to zero. At runtime, polygons are processed in random order. Before scan converting a polygon, we calculate its depth increments in both x and y directions.

When scan converting the polygon, if a pixel is fully covered, the contents of `rgb` and `z` are updated, provided that the depth of the polygon at the pixel position is smaller than `z`. The precalculated depth increments in the x and y directions are then stored to `dxz` and `dyz`, respectively. If the pixel is partially covered by the polygon, an array of `Subpixels` is linked to the corresponding `Pixel`. The flag, `Supersampled`, is then set to TRUE indicating that `zORpblock` is a pointer. All `rgb` fields in the array are initialized and the depth values are initialized by interpolating the `z` value with `dxz` and `dyz`. Fig. 3 shows an example of how to buffer the adaptive supersampling values.

The polygon fragment is then supersampled to update the contents of the `Subpixel` array, based on the z-buffer method. The `dxz` and `dyz`, are the depth increments of `z` in x and y directions, respectively, within the pixel region.

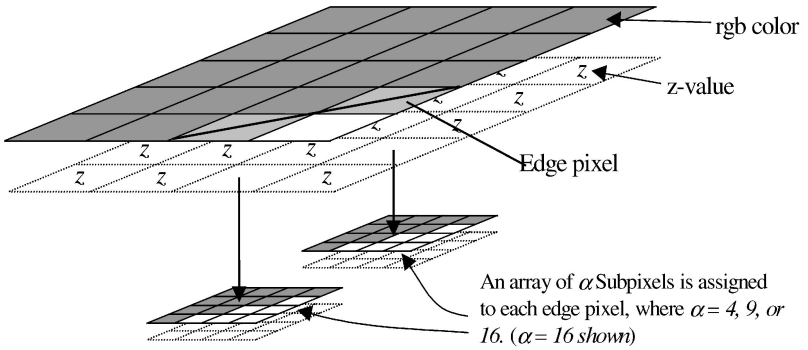


Fig. 3. Adaptive supersampling along the polygon edge.

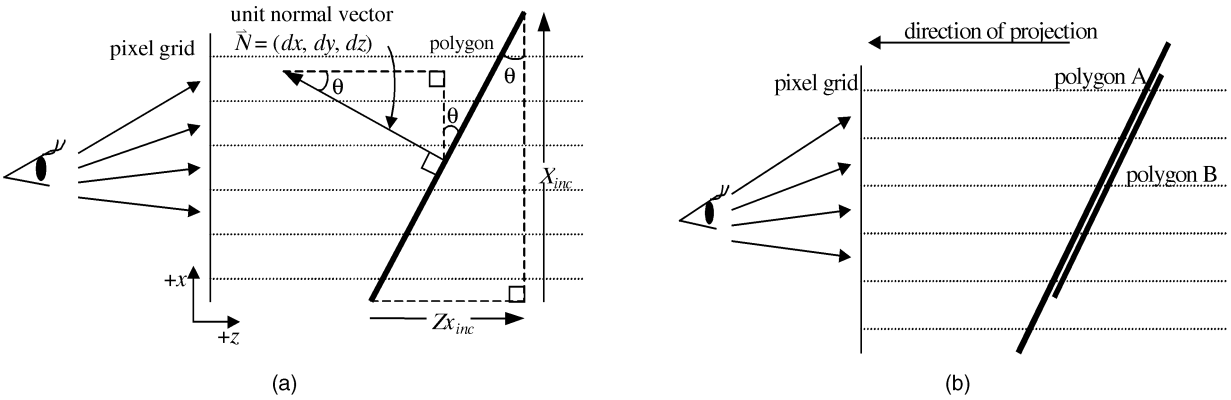


Fig. 4. Solving the surface intersection problem. (a) Calculation of the depth increments. (b) Two polygon surfaces close to each other.

From Fig. 4a, dxz is calculated by:

$$\texttt{dxz} = \frac{Zx_{inc}}{X_{inc}}. \quad (1)$$

In a 3D scene, polygons can assume any orientation. So, calculating X_{inc}, Zx_{inc} and, likewise, Y_{inc}, Zy_{inc} are not straightforward. Instead, we calculate the depth increments from the polygon normal vector. After a polygon has been transformed, we calculate the unit normal vector of the transformed polygon as $N = (dx, dy, dz)$. This unit normal vector may already be available in some implementations of the rendering process. They are used for detecting back-surfaces.

With this normal vector, the depth increments are calculated by:

$$\texttt{dxz} = \frac{dx}{-dz} \quad \text{and} \quad \texttt{dyz} = \frac{dy}{-dz}. \quad (2)$$

In addition, when polygons intersect each other, we need to determine where they intersect at a subpixel level to prevent aliasing from appearing along the line of intersection. In the traditional supersampling method, the depth calculation and comparison are performed at each subpixel position. Hence, the surface intersection problem is solved at subpixel level. In Carpenter's A-buffer method, the depth value is not available in subpixel level. An approximation method was suggested to detect the occurrence of surface intersection and to determine where it happened.

Basically, if two fragments overlap in depth, surface intersection is assumed to occur. The color of the two fragments will be blended proportionally according to their depth values. This method is problematic, as pointed out by Carpenter [3]. One of the problems is that when two parallel surfaces overlap each other in depth, as shown in Fig. 4b, the A-buffer method will mistreat them as intersecting surfaces and blend them together. Although Polygon B is not supposed to be visible, it will appear as behind a somewhat semitransparent Polygon A. Similar to traditional supersampling, our method solves the surface intersection problem at the subpixel level.

3 Parallelization of the Adaptive Renderer

To parallelize the rendering process effectively, we need to understand which stages are computation intensive and can be subdivided for parallelism. As described by Foley et al. [12], most computations are spent in three major steps in a high-quality graphics rendering process:

1. Floating point operations in transforming 3D objects,
2. Rasterizing the transformed polygons, and
3. Antialiasing the synthesized image.

As illustrated in Fig. 5, parallelizing Step 1 exploits *object parallelism* and parallelizing Step 2 exploits *image parallelism*. Examples of an object-parallel system can be found in [37]. Systems exploiting image parallelism are described by Fuchs and Poulton [14]. To achieve higher parallelism, some systems adopt a hybrid approach, incorporating both object-parallel and image-parallel

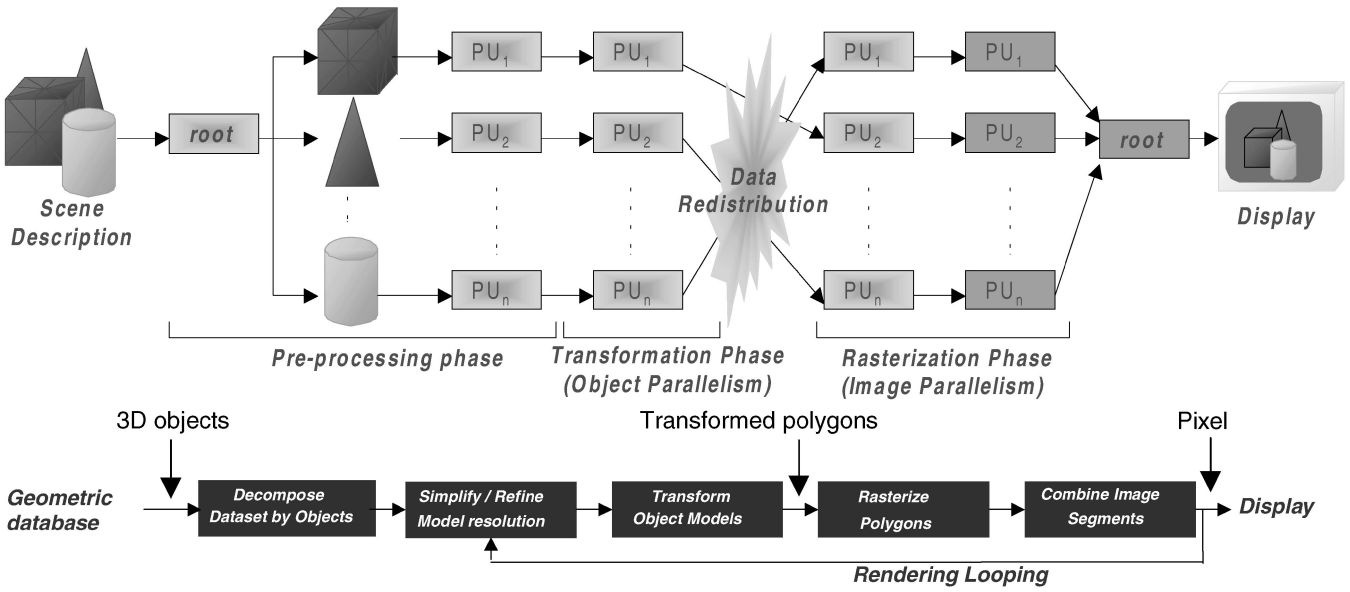


Fig. 5. Parallel image rendering pipeline based on the sort-middle architecture and adaptive supersampling (**PU: processing unit**).

methods [11], [6]. However, most parallel rendering systems did not exploit parallelism in Step 3.

3.1 Adaptive Parallel Renderers

Fig. 5 shows our parallel renderer built with multiple processing units (processors or computers) in a pipelining fashion. The system is divided into two main stages: *object-parallel* and *image-parallel* stages. The object-parallel stage includes two processes: the simplification and the transformation of object models. The image-parallel stage also includes two processes: the rasterization of polygons and combination of image segments. Initially, a process is responsible for reading 3D object models from the database. The models are then evenly distributed in a round-robin manner so that each processor handles nearly the same number of objects. Parameters such as colors and transformation matrices of each object are transferred along with the object data.

In the object-parallel stage, the distributed object models are simplified or refined according to the object distances from the viewer using the multiresolution modeling technique developed by Lau et al. [23], i.e., the closer an object is to the viewer, the higher the resolution. Simplified models are then transformed into the screen coordinates. By checking which region each polygon overlaps, the polygons are added to a sending-buffer to be later transfered to the corresponding processors.

In the image-parallel stage, the screen is partitioned to give a similar amount of workload to each processor. Screen partitioning is a pixel-level load-balancing method. The strategy used to subdivide the screen would therefore affect the effectiveness of load balancing among the processors. Mueller has addressed the problem of screen partitioning in detail [31]. Our rendering algorithm combines the concepts of both adaptive supersampling and multiresolution modeling proposed in [22], [23].

As polygons may be projected onto different regions of the screen, all hybrid systems that parallelize both transformation and rasterization processes have to map data from an *object-space* to an *image-space* [7]. Sutherland et al. [36] described this mapping as a problem of sorting primitives to the screen. As noted in [29], different sorting methods lead to a taxonomy of three rendering architectures. They are called *sort-first*, *sort-middle*, and *sort-last* architectures. Our system adopted the hybrid approach using the sort-middle architecture. We did not choose the sort-first model because it introduces a data replication problem. Nor did we implement the sort-last architecture, in order to avoid massive data movement in the antialiasing process.

The sort-middle architecture does not have the above problems and at the same time provides the flexibility to incorporate other graphics algorithms. Other advantages of the sort-middle architecture over the others were elaborated in [11], [26]. Many parallel rendering systems [5], [11], [31] distribute polygons evenly among processors during the entire rendering process. This fine grained data distribution, however, cannot preserve the information on inter-polygon dependencies conceived inside the objects. This limitation impedes the flexibility of incorporating special graphics algorithms. In the sequel, we will address these issues and present our solutions along with benchmark results.

3.2 Adaptive Supersampling Algorithm

Crow pioneered the supersampling concept in [9]. His supersampling algorithm was originally written for sequential execution on an uniprocessor computer. On a SGI Indigo workstation with a single R4400 CPU, we rendered a scene with 2,000 polygons. Crow's supersampling method took 7.8 seconds, while Lau's adaptive supersampling method took only 0.96 seconds. In other words, the adaptive approach could be eight times faster in this simple image rendering on a single workstation. We have developed a parallel version of Crow's algorithm as

```
Parallel Version of Crow's Algorithm
Distribute polygons (triangles) evenly to all processors
RenderingLoop {
    For Each polygon {
        Transform the polygon to screen coordinate
        Store the polygon into buffer according to screen position}
    Total Exchange of the buffer contents among all processors
    For Each polygon in the buffer {
        Rasterize and Supersample the polygon}
    Filter the frame buffer into output resolution
    Display the image}
```

(a)

```
Adaptive Parallel Rendering Algorithm
Read and Distribute object models in round-robin
Initialize screen-partition tree
Determine subpixel resolution
RenderLoop {
    For Each object {
        Simplify/Refine object model
        Transform object models to screen coordinate
        Put transformed polygons into the outgoing buffers
        If a buffer full, send buffer to specific processor }
    Total Exchange of polygons remaining in all non-empty buffers
    If there are messages {
        For Each polygon in the buffer
            Rasterize the polygon with anti-aliasing}
    Combine image segments and Display the image
    Collect processor loading information
    Adjust screen-partition tree and subpixel resolution}
```

(b)

Fig. 6. Two parallel rendering algorithms with supersampling at the subpixel level. (a) Crow's renderer with full supersampling. (b) Our renderer with adaptive supersampling.

specified in Fig. 6a. Our parallel renderer with adaptive supersampling is specified in Fig. 6b.

Multiresolution modeling algorithms are used to minimize the number of polygons (triangles) to be processed and, therefore, the time to render an object. When an object is projected from the 3D space to the 2D screen, a distant object undeniably occupies a smaller screen area than a nearby one. If an application renders a distant model in full resolution, all polygons are transformed to the same few pixels. Millions of rasterization loops may be wasted because most details are not visible. The multiresolution modeling [23] is applied here.

It requires the information of neighboring polygon configurations. Hence, instead of distributing individual polygons, our parallel renderer decomposes the scene into coarser granularity. Individual objects are distributed evenly among processors, in which model simplification/refinement and transformation are performed. However, similar to the common coarse grained approaches, this strategy inevitably imposes load imbalance between processors when the size of the objects varies greatly. Unlike other approaches, our system focuses on the parallelization of the antialiasing process [25] of the adaptive supersampling method [23].

3.3 Data Redistribution Operations

We have tested our system with two data-exchange approaches between the transformation and rasterization phases, i.e., *asynchronous exchange* and *total-exchange*. In asynchronous exchange, we do not introduce any synchronization point in the rendering loop, except at the end of it where the output image is displayed. For any coarse grained parallel applications, workload on each processor may differ greatly. Any synchronization point within the rendering loop will introduce significant idle time, as the rendering process would be held back by the slowest processor.

To allow the processes to run as asynchronously as possible, special message passing coordination is necessary. Hence, asynchronous exchange is advantageous, when the workloads are highly unbalanced. Crockett presented a similar data-exchange algorithm in [5]. Fig. 7a shows a

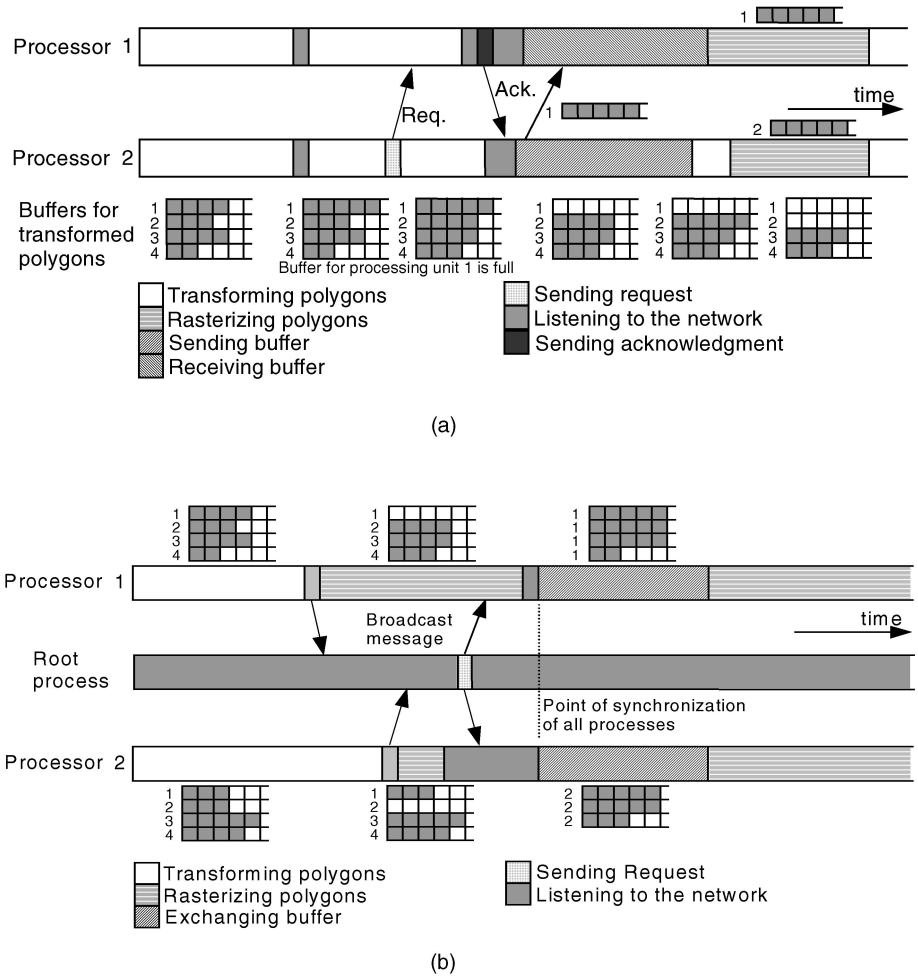


Fig. 7. Redistribution of transformed polygons among processors. (a) Asynchronous exchange between two processors. (b) Total exchange between two processors.

sample case of Crockett's approach in data transfer. Six communication steps are outlined below:

1. A number of buffers are assigned to each processor. Transformed polygons are stored in the appropriate buffers according to their screen positions.
2. When a buffer is full, a *data-transfer* message is sent to the target processor.
3. Each processor periodically probes for incoming messages from the network. When a data-transfer message is received, a *receive-ready* message is returned.
4. Buffer contents in the source processor are sent to the target processor, where the transformed polygons will be rasterized.
5. When all polygons are processed, a *transformation-complete* message is sent to the root process.
6. When the root process receives completion signals from all processors, a *global-complete* message is sent to start the next rendering loop.

In the total exchange, synchronization is needed after all polygons have been transformed. This method minimizes the time to probe for incoming messages from the network and reduces the communication overhead between processors. However, it is not suitable when the size of messages is very large and exceedes the network capacity. In addition, it cannot alleviate the load-balancing problem because the starving processors will remain idle until all the processors have finished their tasks. Fig. 7b shows an example of this buffer exchange method. Three major steps are outlined below:

1. When a processor has finished transforming all the polygons assigned to it and storing them in buffers according to their screen position, a *transformation-complete* message is sent to the root processor to signify the completion.
2. When the root has received the completion signals from all processors, it sends a *total-exchange* signal to all processors.
3. All processes will initiate an *all-to-all exchange* operation, through which they will exchange the contents of their buffers.

The asynchronous exchange and the total exchange methods have their own merit. Which one to use depends on how the data set is partitioned, how many polygons need to be transferred after the transformation, and the load-balancing status of the processes. We will discuss these load-balancing issues further in the next section.

```
Adaptive Load Balancing Algorithm
If (root process) {
    If (first frame)
        Initialize screen-partition tree
    Else {
        Check if screen-partition tree balanced
                from the loading information of previous frame
        If (tree unbalance) {
            Adjust screen-partition tree
            Send new partition information to all processors
        }
    }
}
```

Fig. 8. Adaptive load balancing between successive frames.

4 Adaptive Load Balancing

This section deals with how to efficiently partition the objects in the image in order to balance the workload in the rendering process. Often, each processor is assigned to handle a group of subdivided screen regions. The transformed polygons are sent to the corresponding processors for concurrent rasterization. Since polygons are likely clustered around the center of the screen, processors responsible for rendering the side-regions would starve for work. Therefore, the screen partitioning strategy will affect load balancing among processors and, hence, the performance. Mueller has presented the following two image parallel, load balancing strategies: *static load balancing* and *dynamic load balancing* [31].

A *static load balancing* scheme relies on a fixed screen partitioning for distributing polygons. The granularity of image parallel rendering has been carefully studied in [11]. The optimal granularity ratio is sensitive to different architectural characteristics. Static load balancing may work very well on one machine, but badly on another. It is also not adaptive to the problem size. A *dynamic load balancing* scheme uses dynamic screen partitioning for distributing polygons. Whitman classified dynamic load balancing schemes as either *demand driven* or *task adaptive* [40].

The demand driven approach decomposes the problem domain into smaller independent tasks and then assigns them to different processors. 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 smaller number of coarse grained tasks and assigns them to a few 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 the remaining job.

While static strategies cannot efficiently balance the coarse grain workload among processors, demand driven strategies increase the communications costs. Thus, it is necessary to develop a dynamic load balancing method with low communication overhead. In addition, balancing the whole process is more important than just the image parallel stage. Our adaptive scheme is based on the balancing of a binary tree to maintain the loading among processors balanced. In the following paragraphs, we outline the scheme.

Initially, the screen is partitioned according to the number of processors available for rendering and a screen partition tree is created. Each leaf node of the tree corresponds to one screen partition. The weight of each leaf node is determined from the processor loading in processing the corresponding screen partition. Once the weights of all the leaf nodes are determined, they can be propagated upwards until the root of the tree. The weight of the root is the total loading of all processors. In addition to the weight, each node of the tree also stores the screen size of the region. Before rendering an image frame, we traverse the tree top down and adjust the screen size to make sure that the loading of various nodes is balanced.

In real time or interactive graphics rendering, one often deals with a sequence of image frames. Our approach to load balancing was inspired by the coherence concept between frames [11]. There exists some coherence properties between successive frames. Partitioning successive frames into regions can be dynamically adjusted using coherence information, such as polygon count or loading time of different regions. Based on the loading time, Fig. 8 shows the steps in our adaptive load balancing scheme between two successive frames.

This load balancing concept is indeed a divide and conquer scheme. We try to maintain the loading of the two subtrees of the binary screen partition tree even at all levels. As shown in Fig. 9a, the algorithm starts by splitting the screen into regions. In case there are only two processors, it splits the screen into two. If one more processor is available, one of the regions is further split into two along the longer dimension of the region, either vertically or horizontally.

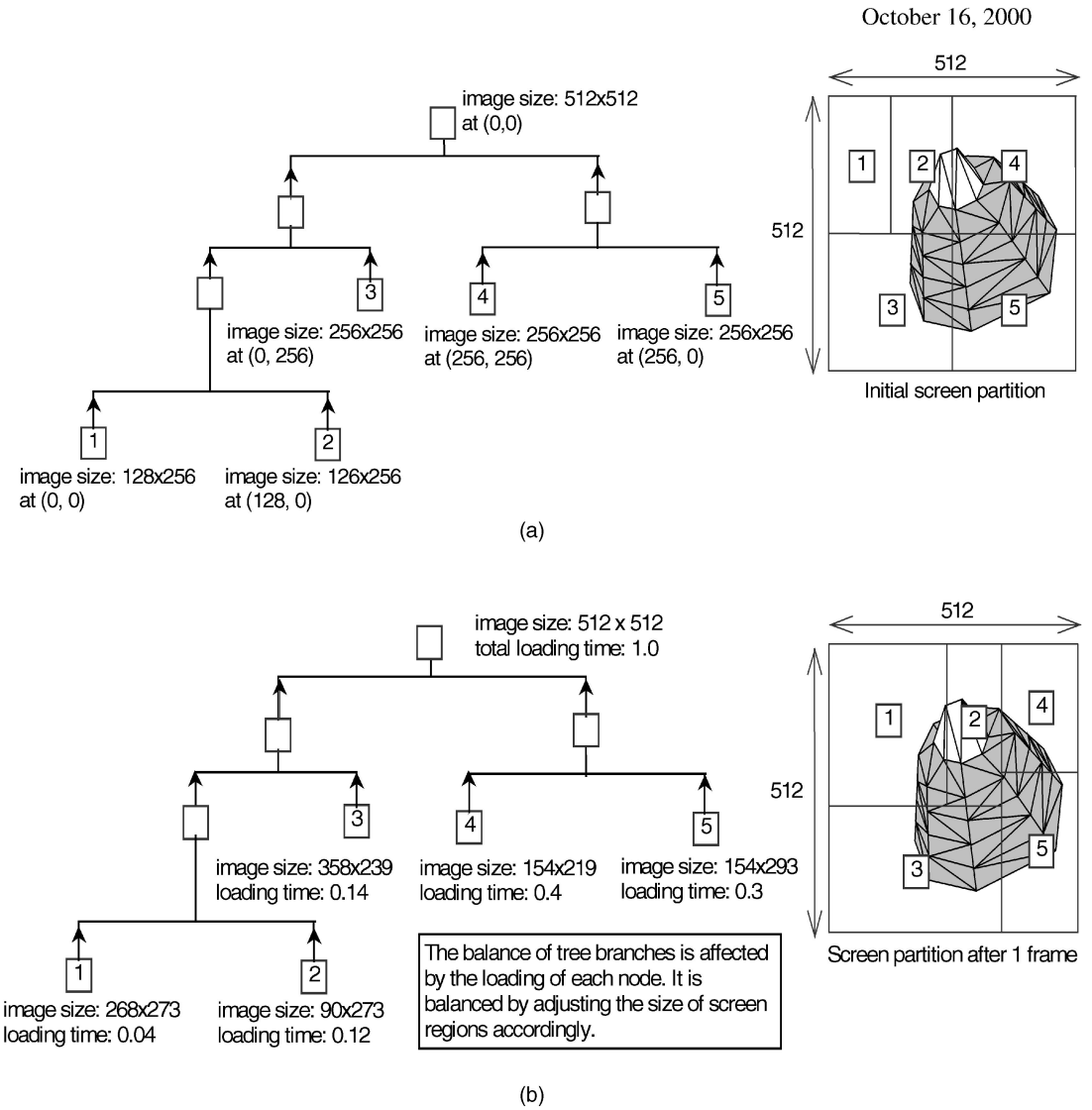


Fig. 9. Adaptive load balancing between successive image frames. (a) Initial screen partitioning for load balancing in the first frame. (b) Screen repartitioning in the second frame based on the loading information of the first frame.

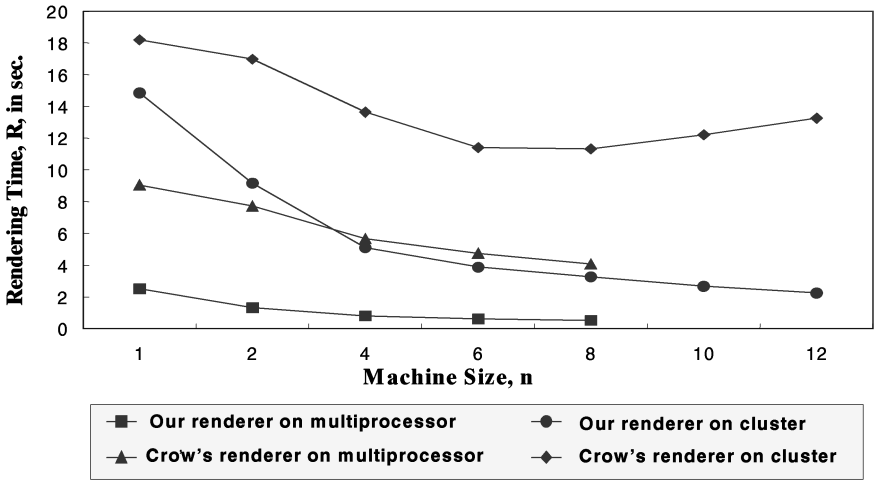
After the screen partition tree is created, the root will distribute the partition information to all the processors. The processors use the partition information to determine where each polygon should be sent after the transformation. The partition tree is adjusted at the end of rendering each frame, according to the loading of the processors in the frame. Fig. 9b shows the adjusted partition information at the second frame after the initial partition in Fig. 9a. Thus, the loading information collected in a frame affects the screen partitioning of the next frame. This dynamic load scheduling method minimizes the communication costs between processes.

Ellsworth [11] suggested the frame-to-frame coherence concept. His method is derived from polygon counts with a fixed number of eight regions per processor. Whitman [40] has also discussed the critical issues in dynamic load balancing for parallel polygonal rendering. We choose a divide-and-conquer approach with three distinctions:

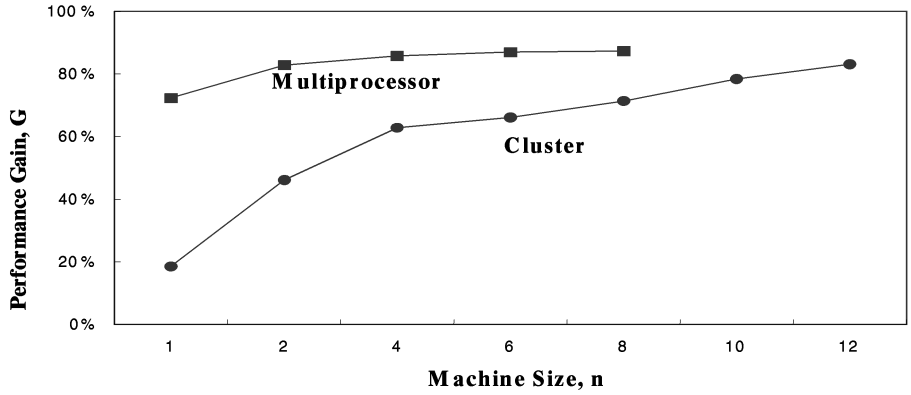
1. The screen regions are dynamically adjusted according to the processing time of different regions.
2. The minimization of communication overheads is included in the re-partitioning process.
3. The binary tree is balanced and frame-to-frame coherence is considered. We show next the effectiveness of using this adaptive load-balancing scheme.

5 Performance Results and Analysis

We have implemented Crow's renderer and our adaptive renderer on two parallel computers: a SGI *symmetric multiprocessor* (SMP) with eight CPUs sharing a common memory and a Unix cluster of 12 Sun workstations with distributed memory. To have a fair comparison, both renderers are MPI coded using the sort-middle architecture shown in Fig. 5. The data file contains 65,340 polygons and the output image resolution has 512 x 512 pixels.



(a)



(b)

Fig. 10. Performance of two parallel renderers on a SGI multiprocessor and on a Sun workstation cluster. (a) The performance of the two parallel renderers. (b) Performance gain of our renderer over Crow's renderer.

5.1 Parallel Rendering Performance

The *rendering time*, $\mathbf{R}$, is the major performance metric used. We plot $\mathbf{R}$ as a function of the number of processors, $\mathbf{n}$, in the parallel computer. Fig. 10a shows four curves corresponding to the rendering times of using two renderers on two parallel machines. Our renderer outperforms Crow's renderer on both types of machines. Interestingly, our renderer running on the cluster is even faster than Crow's renderer on the SMP when the machine size $\mathbf{n} > \mathbf{4}$. It is because our renderer has been optimized in each stage to reduce the amount of data transfer and to eliminate unnecessary computations. Note that the rendering time of Crow's scheme increases on the cluster as machine size increases to $\mathbf{n} > \mathbf{8}$.

This is due to the fact that Crow's scheme experienced higher communication cost as the cluster size increases beyond a certain limit. Our parallel renderer, showing a steady drop in rendering time, has no such performance degradation on the cluster platform. In these timing experiments, we have also measured the speed of our renderer as being able to process 147K polygons/s on the 8-processor SMP and 63K polygons/s on the workstation cluster with 12 nodes.

Let $\mathbf{R}_{\mathbf{Crow}}$ be the rendering time of Crow's scheme and $\mathbf{R}_{\mathbf{Lin}}$ be that of our renderer. The *performance gain*, $\mathbf{G}$, is defined by:

$$\mathbf{G} = (\mathbf{R}_{\mathbf{Crow}} - \mathbf{R}_{\mathbf{Lin}})/\mathbf{R}_{\mathbf{Crow}}. \quad (3)$$

Fig. 10b shows the performance gains of our parallel renderer over Crow's scheme. On the SMP, our renderer outperforms the Crow's renderer by 87 percent. However, this performance gain occurs with four processors. With more processors, the performance gain does not increase much. This is due to the fact that our renderer has reached its' optimal speed. On the cluster of workstations, our renderer shows even a greater performance gain with increasing cluster size. With 12 workstations, the gain is 83 percent. This gain can be even greater with more than 12 workstations in the cluster, because Crow's method experienced more communication overheads as the cluster size increases beyond eight.

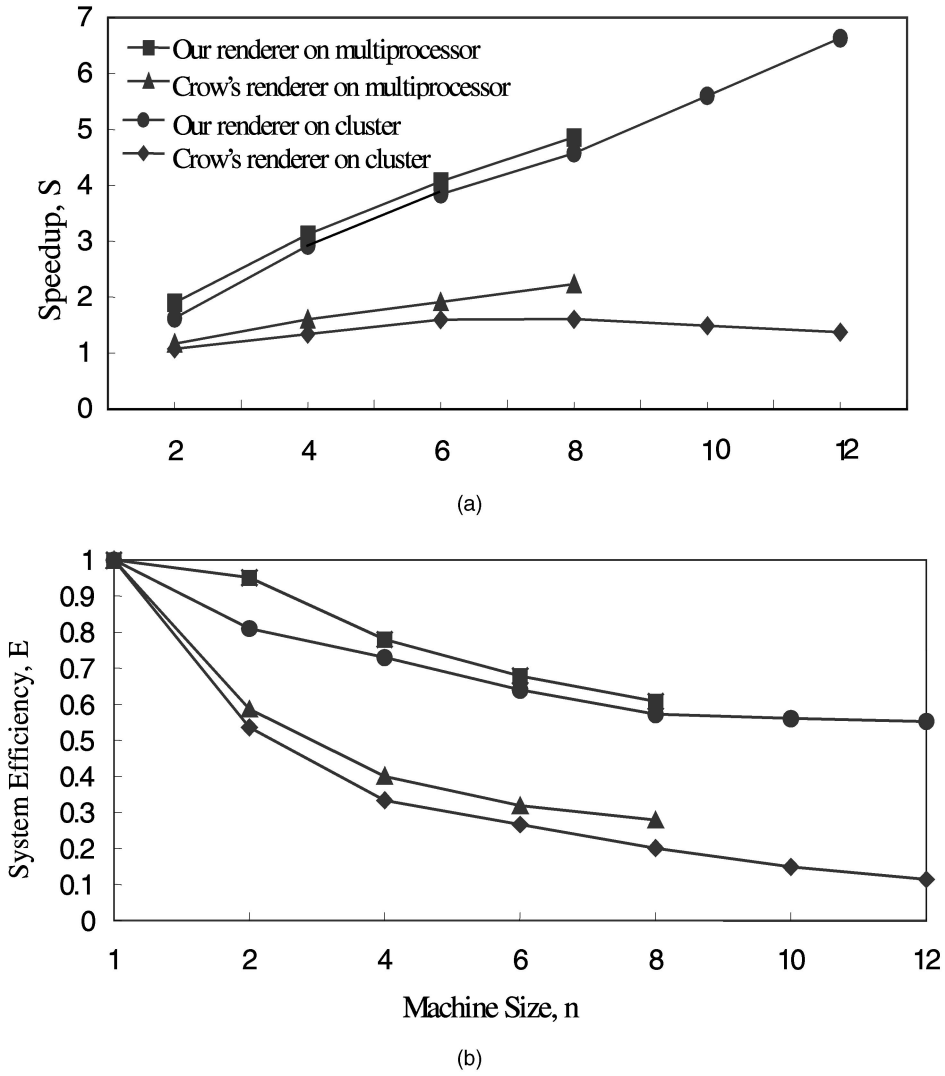


Fig. 11. Speedup and efficiency of two parallel renderers on a SGI multiprocessor and on a Sun workstation cluster. (a) Speedup factors. (b) System efficiency.

5.2 Speedup and Efficiency

Speedup and *efficiency* are used to compare the relative performance of the parallel renderers. Let $\mathbf{T_1}$ be the sequential execution time on a single processor and $\mathbf{T_n}$ the parallel execution time on $\mathbf{n}$ processors. With the same input data size, we define the speedup, $\mathbf{S}$, and efficiency, $\mathbf{E}$, as follows:

$$\mathbf{S} = \mathbf{T_1}/\mathbf{T_n}, \tag{4}$$

$$\mathbf{E} = \mathbf{S}/\mathbf{n} = \mathbf{T_1}/\mathbf{n}. \tag{5}$$

Fig. 11 shows the speedup and efficiency results obtained. Fig. 11a plots the *speedup factors*, $\mathbf{S}$, of the two renderers on the SMP and the workstation cluster, respectively. In terms of speedup, our parallel renderer achieves a factor of 4.5 on an 8-processor SGI server and 6.5 on a 12-workstation cluster. In comparison, the Crow's renderer has a maximum speedup of two on the SGI multiprocessor. In terms of system efficiency (Fig. 11b), our renderer operates with 65 percent efficiency on the workstation cluster, compared with a 10 percent efficiency on Crow's renderer. This clearly demonstrates the strength of our adaptive supersampling algorithm on parallel processors.

5.3 Effect of Load Balancing

Based on our experimental results, load balancing affects the performance of the rendering process on the two parallel computers differently. The difference comes mainly from the communication overhead encountered. In Fig. 12, we illustrate the performance effect of load balancing on a 10-workstation cluster. The effects on a shared memory multiprocessor are shown in Fig. 13. Here, the same scene file is processed on both machines, as we have done in previous experiments.

Polygons are distributed and processed by the 10 workstations, simultaneously. We recorded the times to render four successive image frames in four bar groups. Bar sections with different shadings correspond to the breakdown of the rendering time in three stages: transformation, data redistribution, and rasterization, from the bottom upward. The transformation time is very small, compared

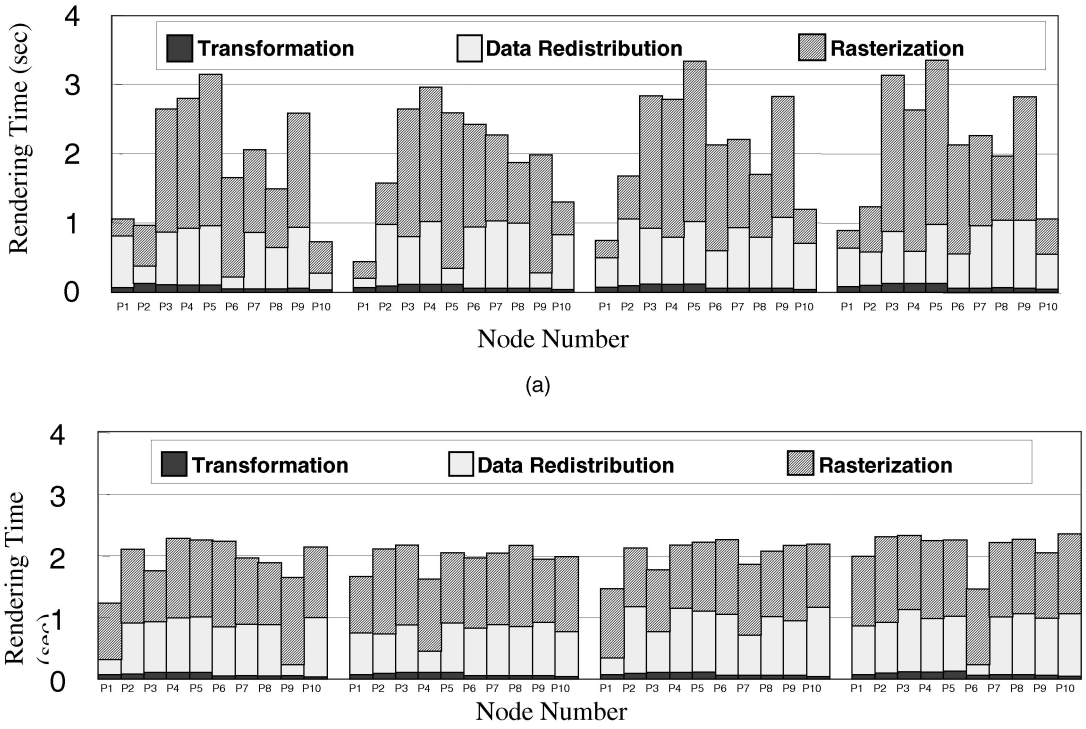


Fig. 12. Effect of load balancing on the rendering time over a cluster of 10 Sun workstations. (a) Without load balancing. (b) With load balancing.

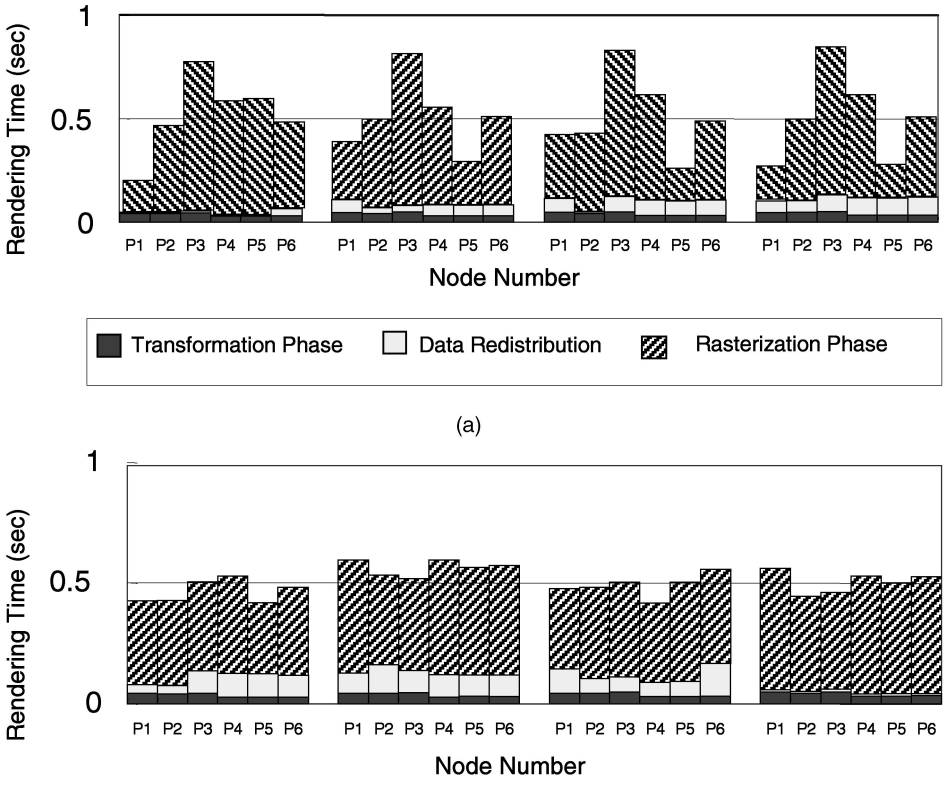


Fig. 13. Effect of load balancing on the rendering time over a 6-processor shared memory system. (a) Without load balancing. (b) With Load balancing.

with the other two stages. This time is independent of the workload distribution, as revealed in parts (a) and (b) of both Fig. 12 and Fig. 13. However, the load distribution does significantly affect the times for data redistribution and for rasterization.

The performance of the renderer when the loading of workstations is unbalanced is shown in Fig. 12a. It took about 3.4 seconds to finish the rendering in the worst case. From the shaded top section of the bars, the imbalance of the scheduling tree has the major effect on the rendering time during the rasterization stage. This stage involves scan converting the polygons, hidden surface removal, and shading. All of these operations take significant amount of computing time. Hence, the rasterization time may differ significantly when unbalanced numbers of polygons are assigned to different workstations, causing an unbalanced workload on them.

The rendering times of the remaining two stages, the transformation stage (lower portions of the bars), and the data redistribution stage (middle portions of the bars), did not change much with the load distribution. The transformation stage takes less than 0.2 seconds, while the data redistribution stage takes less than 1.2 seconds. However, the rasterization stage takes as long as 2.3 seconds to finish on some workstations. Recall that it takes 15 seconds to render the same image on a single workstation (Fig. 10a).

The balanced renderer (Fig. 12b) took 2.3 seconds to finish in the worst case, giving a 32 percent performance improvement over the unbalanced case. Balancing the workload shortens the rasterization time sharply. On the contrary, the data redistribution time is shorter in the unbalanced case than in the balanced case. This means that when a workstation does more in the rasterization stage, it may have spent less time to redistribute the polygons around the workstations. In Fig. 12b, seven or eight workstations spend almost equal amounts of rendering time on their assigned regions. Among four successive image frames, the balancing act becomes more apparent from the left to the right bar groups. This clearly demonstrates the adaptive and self-learning nature of our load balancing scheme.

Fig. 13 shows the results on a 6-processor system. The rendering time on the multiprocessor is significantly reduced from that on the workstation cluster. In the case of an unbalanced workload, Fig. 13a shows a maximum rendering time of 0.75 seconds, almost four times shorter that in Fig. 12a. With shared memory, the interprocessor communication, even with MPI code, becomes much shorter in time. In Fig. 13, the shorter middle sections (white) of the timing bars demonstrate this effect. On the average, the data redistribution time among the processors is limited to less than 0.15 second in Fig. 13, compared to 1.2 seconds on the workstation cluster in Fig. 12.

It is interesting to note that the data redistribution time may increase with the balanced workload (Fig. 13b), comparing with that shown in Fig. 13a. This is due to the fact that to balance the workload may introduce more communication demand. This is more true in implementing the renderer on a shared memory multiprocessor than that on a workstation cluster. Finally, we notice that it is the balancing of the rasterization time that has contributed most to the balance of the overall rendering time. This is true on both types of parallel computers that we have tested. This can be explained by the fact that the act of load balancing was designed mainly to balance the polygon counts and, thus, the rendering times in different regions of the image.

6 Memory Reduction Techniques

The memory management of a variable sized buffering method is an important issue. When rendering a highly complex image, the memory consumption of a VSB method can be very high. It is often very difficult for a process to allocate so much memory resources dynamically. To cope with this difficulty, we have developed two memory management techniques for the adaptive supersampling method to minimize its memory usage. One is the memory adaptation technique and the other is the memory reclaiming technique.

6.1 The Memory Adaptation Technique

The *memory adaptation technique* changes the subpixel resolution, α, according to the available memory and memory usage statistics in the previous frame. Thus, the algorithm can handle images of any complexity. Our method supports three subpixel resolutions: 4×4, 3×3, and 2×2. The choice depends on the total number of edge pixels in the image. If it is small, we supersample the edge pixels using higher subpixel resolution; otherwise, we use a lower subpixel resolution.

When rendering the first frame, we set the resolution to the highest, i.e., $\alpha = 4 \times 4$. Whenever an edge pixel is created, 16 `Subpixel` cells are allocated to the pixel. The system provides enough memory for 25 percent of all the pixels to be supersampled at this subpixel resolution. If the memory usage in this frame exceeds the preallocated limit, no further supersampling will be needed.

Due to frame coherence, the number of edge pixels in the next frame is expected to be similar to that in the current frame. If the number of edge pixels found in the current frame is higher than 25 percent of the pixels, the algorithm will decrease the subpixel resolution of the next frame from 4×4 to 3×3, as shown in Fig. 14. This allows up to 44 percent of the pixels in the image to be supersampled. We may further reduce the subpixel resolution to 2×2, which will allow up to 100 percent of the pixels to be supersampled. The decision of using which subpixel resolution to render the next frame depends on the memory consumption in the current frame.

6.2 Memory Reclaiming Technique

The second technique to reduce memory consumption is to reclaim the 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 together may cover the whole pixel, as shown in Fig. 15. An object with a lot of "internal" edges may produce many supersampled pixels. Because polygons sharing a common edge usually produce the same color values along the

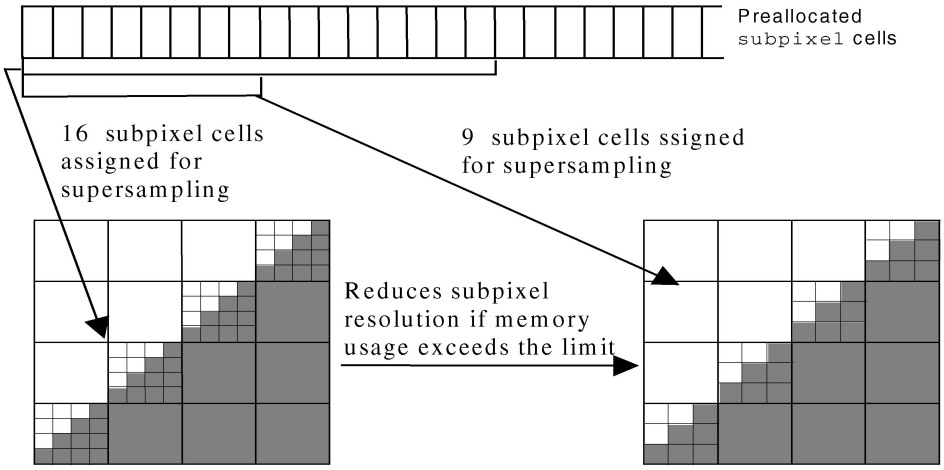


Fig. 14. Example on the change of subpixel resolution.

common edge pixels, we keep track of supersampled pixels with the same color values and similar depth values so that we may reclaim the `Subpixel` cells later.

Hence, if an edge pixel becomes fully covered and the difference between the maximum and minimum depth values within the pixel is smaller than a given threshold, we assume the pixel fully covered by two adjacent polygons and reclaim all the `Subpixel` cells allocated to it. The reclaimed `Subpixel` cells will be returned to the memory pool of `Subpixel` cells. Although this technique imposes a computation overhead, a large amount of `Subpixel` cells can be reclaimed in practice.

Fig. 16 shows the effect of changing the subpixel resolution, α, on the rendering time, $\mathbf{R}$. The rendering time decreases as α decreases because the system does less work in the sampling process. Our adaptive supersampling method can change the subpixel resolution during runtime according to the available memory. The adaptive supersampling minimizes the memory usage when applying antialiasing. The image pixels synthesized in a processor, which contain supersampling blocks, are already optimized to retain only the needed visual information for hidden surface determination.

Our strategy scales with problem size, as we can render larger images before the data volume saturates the memory bus. Our memory management policy enhances the efficiency of parallel computers in using a limited amount of local memory as pixel or subpixel buffers. Next, we discuss the performance effects of the scene complexity, which is also related to memory requirements.

7 Effects of Scene Complexity

The complexity of the scene can be roughly indicated by the total number of polygons in the scene. When rendering an image, the performance of the renderer is highly affected by the scene complexity. The higher the scene complexity is, the lower the performance of the renderer will be. In this section, we study experimentally how the performance and the memory usage of the renderer are affected by the change in scene complexity.

7.1 Performance Effect of Scene Complexity

Fig. 17 shows the effect on the rendering time, $\boldsymbol{R}$, when the number of polygons, β, in the scene increases. Our renderer demonstrated better scalability than Crow's renderer did, when the data size increases. As the scene complexity

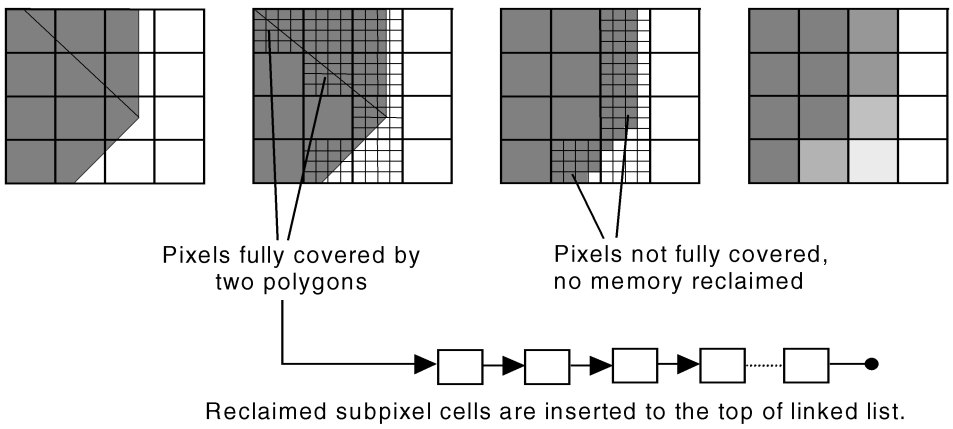


Fig. 15. Reclaiming of Subpixel cells from internal edge pixels.

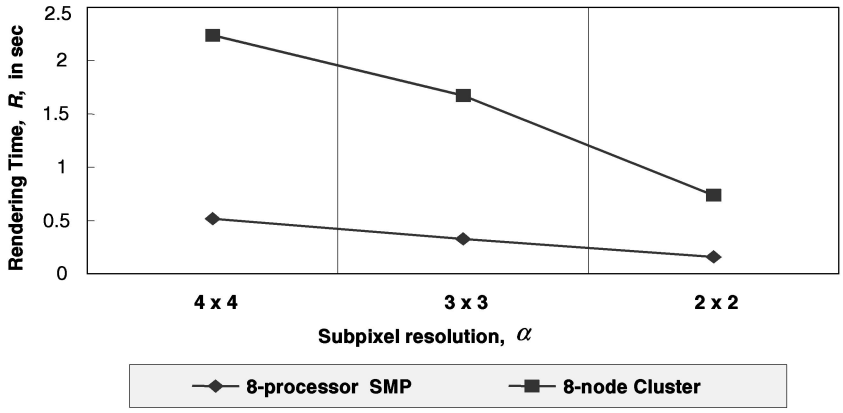


Fig. 16. Effects of changing subpixel resolution on the rendering time.

increases, the rendering time of Crow's method increases much more rapidly than ours. It is difficult to give a quantitative formula or a precise definition on the rendering time, as there are many factors affecting the rendering time apart from the number of polygons.

Other factors include the arrangement and the size of polygons in the scene. For example, we may consider an extreme case when there is a large polygon covering the entire screen, the number of polygons and, therefore, the scene complexity may be considered as very low. However, Crow's renderer will take a rather long time to render the image, as it requires to sample all Subpixels. For our method, it takes considerable less time (about sixteenth of the time) to finish the rendering.

7.2 Scene Complexity Effects on Memory Demand

An advantage of our adaptive supersampling method is that it is memory adaptive. Fig. 18 compares the memory usage, M, of four scan conversion methods against the scene complexity. The size of the image used in the tests is 512×512. Each pixel of the z-buffer requires seven bytes, and each pixel of the A-buffer requires 12 bytes but each fragment needs 24 bytes. This calculation includes four extra bytes to store the second depth value for detecting surface intersection. The first polygon edge encountered causes two fragments to be created one for the polygon edge and another for the background.

Among the four rendering schemes, the z-buffer method requires the least memory (1.8 MB) and Crow's method requires the most memory (28 MB); both are independent of the scene complexity. The memory requirement of the A-buffer method increases linearly with the scene complexity. Our adaptive supersampling method requires no more than 10 MB of memory; each pixel of the buffer requires 12 bytes and each edge pixel with 2×2, 3×3, or 4×4 subpixel resolutions requires an extra 28, 63, or 112 bytes of memory, respectively.

The reason our method has a zigzag appearance in memory usage is that, when the number of edge pixels increases (due to the increase in scene complexity) to a point that the available memory is not enough, the subpixel resolution of all edge pixels will be decreased. This results in a sudden drop in memory usage. As the number of edge pixels continues to increase, the available memory may yet again not be enough, so we must reduce the subpixel resolution further. This results in another sudden drop in memory usage, and so on.

From our experience, the maximum number of edge pixels created when rendering complex images does not exceed 20 percent of the total number of pixels in the image.

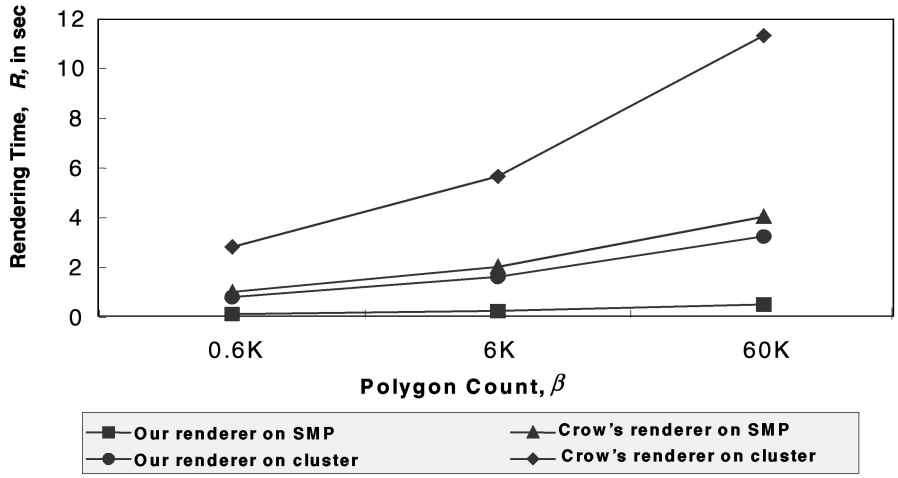


Fig. 17. Effect of polygon count on the rendering time.

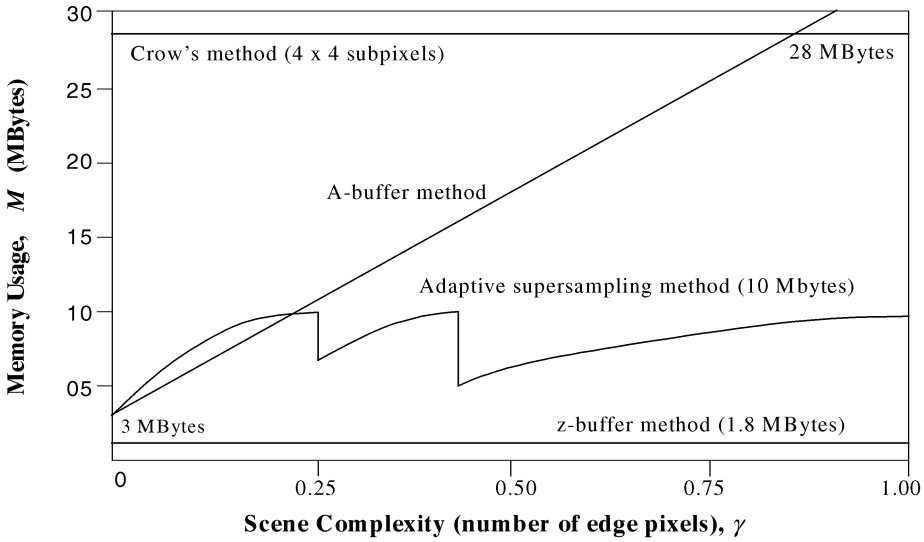


Fig. 18. Memory usage against scene complexity with different rendering methods.

This is because when the scene complexity increases, more and more objects tend to obscure each other. In our implementation, we allocate enough memory to render images with up to 25 percent of edge pixels at 4×4 subpixel resolution. This adds up to a total of 10 MB of memory. If there are more than 25 percent of edge pixels, we reduce the subpixel resolution and keep the maximum memory usage of the renderer to 10 MB at any time.

In some applications, the scene complexity may vary greatly. In order to handle images of any complexity, we need to provide enough memory to supersample all pixels at a 2×2 subpixel resolution; with this amount of memory, we may supersample 25 percent of all the pixels at 4×4 subpixel resolution. However, if we are certain that an application will never handle scenes 100 percent with edge pixels, we may reduce the size of the allocated memory, accordingly.

On the other hand, if an application frequently generates highly complex images, we may allocate more memory to the buffer to improve the quality of the output images. In some cases, consecutive images may not preserve frame-to-frame coherence, e.g., a rapid change of the viewing direction. The memory adaptation technique may give an incorrect prediction. Thus, some edge pixels may not be supersampled when all the available `Subpixel` cells are used up. This problem will be corrected in the next frame as the memory adaptation technique reduces the subpixel resolution.

8 Conclusions

In this research, we have developed an efficient rendering method for MIMD parallel processors. We cover both shared-memory multiprocessors and message-passing PC or workstation clusters. We have integrated Lau's adaptive supersampling and multiresolution modeling methods into the adaptive parallel renderer. The new renderer simplifies the rendering process and saves both memory and processing time. Instead of supersampling every pixel in the image, as done in Crow's scheme, our algorithm takes supersamples only at polygon edges.

Adaptive supersampling is the key that enables the reduction of the workload in resolving surface visibility. Polygon edging was also independently studied in [17] without using the concept of supersampling. For a complex scene, our adaptive renderer saves 60 percent of buffer memory, compared with the use of A-buffer method and Crow's method. In terms of rendering time, our adaptive renderer achieved more than 80 percent speed gain over Crow's parallel renderer. This translates into a speedup factor of 4.5 on an 8-processor SGI server and of 6.5 on a 12-workstation Unix cluster.

As a comparison, the Crow's render has a maximum speedup of two on both parallel platforms we have tested. In terms of system efficiency, our renderer achieved 65 percent efficiency on the 12-node cluster, compared with only a 10 percent efficiency achievable by Crow's renderer. Our adaptive parallel renderer is shown to be easily portable on either a shared-memory multiprocessor or a cluster of workstations. The coarse-grain rasterization process demands a load balancing among the processors. Our adaptive load-balancing method provides a very satisfactory solution to this imbalance problem in polygon processing.

Our experimental results produced concrete evidence of the effectiveness of the newly developed adaptive parallel render, regardless of the complexity of the 3D scene. The revealed effects of subpixel size and of scene complexity are useful to those who want to use the adaptive renderer with further optimization or fine-tuning for even higher performance. We have exploited parallelism primarily at the polygon level. Thus, we can expect finer granularity per polygon. This makes the adaptive parallel renderer attractive for use not only on fine-grain multiprocessors, but also on coarse-grain multicomputer clusters or even *massively parallel processors* (MPP).

The sort-middle architecture is shown especially effective in reducing all-to-all communication time. It is also more flexible to incorporate special 3D graphics algorithms

in this architecture. The MPI-coded rendering program can be efficiently executed not only in a cluster of workstations, but also in a shared-memory multiprocessor environment Our scheme was developed mainly for software implementation on general-purpose multiprocessors or workstation clusters.

Our work complements the earlier works reported in [2], [5], [6], [9], [11], [22], [40]. Further research could be extended to build special hardware mechanisms, similar to those reported in [14], [30], [42], [35] to yield even higher performance using the adaptive supersampling approach. For implementation details of the proposed renderer, readers are advised to refer to Lin's thesis [26].

Based on parallel computer technology in 2001, the multiprocessors or server platforms outperform the workstation clusters in this new rendering scheme. However, if the message-passing overhead could be further reduced in future clusters using faster message passing or better latency-hiding techniques, the performance gap between these two classes of parallel computers could be closer. The proposed polygon renderer, being adaptive and MPI-coded, has a high portability with a cross-platform performance. Our experimental results reinforce this point observed by Crockett [8].

ACKNOWLEDGMENTS

This research was supported in part by Hong Kong RGC grants 7048/98E, HKU 3/96C, HKU 7022/97E, in part by the City U grant for Project No. 9030796, and in part by a special research grant from the Engineering Dean's office, University of Southern California.

REFERENCES

[1] G. Abram, L. Westover, and T. Whitted, "Efficient Alias-Free Rendering Using Bit-Masks and Look-Up Tables," *ACM Computer Graphics,* vol. 19, no. 3, pp. 53-59, July 1985.

[2] K. Akeley and T. Jermoluk, "High-Performance Polygon Rendering," *ACM Computer Graphics,* vol. 22, no. 4, pp. 239-246, Aug. 1988.

[3] L. Carpenter, "The A-Buffer, an Antialiased Hidden Surface Method," *Computer Graphics, Proc. SIGGRAPH '84,* vol. 18, no. 3, pp. 103-108, July 1984.

[4] E. Catmull, "A Hidden-Surface Algorithm with Anti-Aliasing," *ACM Computer Graphics,* vol. 12, no. 3, pp. 6-11, Aug. 1978.

[5] T.W. Crockett, "A MIMD Rendering Algorithm for Distributed Memory Architectures," *ACM Parallel Rendering Symp.,* pp. 35-42, 1993.

[6] T.W. Crockett and T. Orloff, "Parallel Polygon Rendering for Message-Passing Architectures," *IEEE Parallel and Distributed Technology,* vol. 2, no. 2, pp. 17-28, Feb. 1994.

[7] T.W. Crockett, "Beyond the Renderer: Software Architecture for Parallel Graphics and Visualization," *Proc. Eurographics Workshop Parallel Graphics and Visualization,* Chalmers and Jansen, eds., pp. 1-15, Sept. 1996.

[8] T.W. Crockett, "Portability and Cross-Platform Performance of an MPI-based Parallel Polygon Renderer," technical report, Inst. for Computer Applications in Science and Eng., NASA Langley Research Center, 1997.

[9] F.C. Crow, "The Problem in Computer-Generated Shaded Images," *Comm. ACM,* vol. 20, no. 11, pp. 799-805, Nov. 1977.

[10] T.A. Davis and E.W. Davis, "Rendering Computer Anomation on A Network of Workstations," *IEEE Trans. Computers,* pp. 726-730, 1998.

[11] D. Ellsworth, "A New Algorithm for Interactive Graphics on Multicomputers," *IEEE Computer Graphics and Applications,* vol. 14, no. 4, pp. 33-40, July 1994.

[12] J. Foley, A. van Dam, S. Feiner, and J. Hughes, *Computer Graphics: Principles and Practices.* Addison-Wesley, 1990.

[13] B. Freisleben, D. Hartmann, and T. Kielmann, "Parallel Ray Tracing: A Case Study on Scheduling on Workstation Clusters," *IEEE Trans. Computers,* pp. 596-605, 1997.

[14] H. Fuchs and J. Poulton, "Pixel-Plances: a VLSI-Oriented Design for a Raster Graphics Engine," *VLSI Design,* vol. 2, no. 3, pp. 20-28, Mar. 1981.

[15] C. Giertsen and J. Paterson, "Parallel Volume Rendering on A Network of Workstations," *IEEE Computer Graphics and Applications,* pp. 16-23, Nov. 1993.

[16] P. Haeberli and K. Akeley, "The Accumulation Buffer: Hardware Support for High-Quality Rendering," *ACM Computer Graphics,* vol. 24, no. 4, pp. 309-318, Aug. 1990.

[17] R. Herrell, J. Baldwin, and C. Wilcox, "High-Quality Polygon Edging," *IEEE Computer Graphics and Applications,* vol. 15, no. 4, pp. 68-74, July 1995.

[18] K. Hwang and Z. Xu, *Scalable Parallel Computing: Technology, Architecture, and Programming.* chapters 8-9, 1998.

[19] K. Hwang, H. Jin, E. Chow, C.-L. Wang, Z. Xu, "Designing SSI Clusters with Hierarchical Checkpointing and Single I/O Space," *IEEE Concurrency,* vol. 7, no. 1, pp. 60-69, 1999.

[20] K. Hwang, H. Jin, and R. Ho, "RAID-x: A New Distributed Disk Array for I/O-Centric Cluster Computing," *Proc. Nineth IEEE Int'l Symp. High-Performance Distributed Computing (HPDC-9),* pp. 279-286, Aug. 2000.

[21] R.W.H. Lau and N. Wiseman, "Accurate Image Generation and Interactive Image Editing with the A-Buffer," *Proc. Eurographics '92,* vol. 2, no. 3, pp. 279-288, Sept. 1992.

[22] R.W.H. Lau, "An Adaptive Supersampling Method," *Image Analysis Applications and Computer Graphics,* Chin et al. eds., pp. 205-214, Dec. 1995.

[23] R.W.H. Lau, M. Green, D. To, and J. Wong, "Real-Time Continuous Multi-Resolution Method for Models of Arbitrary Topology," *Presence: Teleoperators and Virtual Environments,* vol. 7, no.1, pp. 22-35, Feb. 1998.

[24] T.Y. Lee, C.S. Raghavendra, and J.N. Nicholas, "Image Composition Schemes for Sort-Last Polygon Rendering on 2D Mesh Multicomputer," *IEEE Trans. Visualization and Computer Graphics,* vol. 2, no. 3, pp. 202-217, July-Sept. 1996.

[25] S. Lin, R.W.H. Lau, X. Lin, and P.Y. Cheung, "An Anti-Aliasing Method for Parallel Rendering," *Computer Graphics Int'l,* pp. 228-235, June 1998.

[26] S. Lin, "Adaptive Parallel Rendering," M. Phil. thesis, Dept. of Electrical and Electronic Eng., the Univ. of Hong Kong, Hong Kong, Aug. 1999.

[27] K.L. Ma and T. Crockett, "A Scalable Cell-Projection Volume Rendering Algorithm for 3D Unstructured Data," *Proc. Parallel Rendering Symp.,* pp. 95-104, Oct. 1997.

[28] S. Molnar, J. Eyles, and J. Poulton, "PixelFlow: High-Speed Rendering Using Image Composition," *ACM Computer Graphics,* vol. 26, no. 2, pp. 231-240, July 1992.

[29] S. Molnar, M. Cox, D. Ellsworth, and H. Fushs, "A Sorting Classification of Parallel Rendering," *IEEE Computer Graphics and Applications,* vol. 14, no. 4, pp. 23-31, July 1994.

[30] J. Montrym, D. Baum, D. Dignam, and C. Migdal, "InfiniteReality: A Real-Time Graphics System," *ACM Computer Graphics,* pp. 293-301, 1997.

[31] C. Mueller, "The Sort-First Rendering Architecture for High-Performance Graphics," *Proc. ACM Symp. Interactive 3D Graphics,* pp. 75-84, 1995.

[32] E. Nakamae, T. Ishizaki, T. Nishita, S. Takita, "Composting 3D Images with Anti-aliasng and Various Shading Effects," *IEEE Computer Graphics and Applications,* pp. 21-29, Mar. 1989.

[33] S. Nishimura and T. Kunii, "VC-1: A Scalable Graphics Computer with Virtual Local Frame Buffers," *ACM Computer Graphics,* pp. 365-372, Aug. 1996.

[34] F. Reeth, R. Welter, and E. Flerackers, "Virtual Camera Oversampling: A New Parallel Anti-Aliasing Method for Z-Buffer Algorithms," *Computer Graphics Int'l,* pp. 241-254, 1990.

[35] B. Schneider and J. van Welzen, "Efficient Polygon Clipping for an SIMD Graphics Pipeline," *IEEE Trans. Visualization and Computer Graphics,* vol. 4, no. 3, pp. 272-285, July-Sept. 1998.

[36] I. Sutherland, R. Sproull, and R. Schumacker, "A Characterization of Ten Hidden-Surface Algorithms," *ACM Computing Survey,* vol. 6, no. 1, pp. 1-55, Mar. 1974.

[37] J. Torberg, "A Parallel Processor Architecture for Graphics Arithmetic Operations," *ACM Computer Graphics,* vol. 21, no. 4, pp. 197-204, July 1987.
[38] A. Schilling and W. Strasser, "EXACT: Algorithm and Hardware Architecture for an Improved A-Buffer," *ACM Computer Graphics,* pp. 85-91, 1993.
[39] A. Watt and M. Watt, *Advanced Animation and Rendering Techniques.* Reading, Mass.: ACM Press and Addison-Wesley, pp. 119-124, 1992.
[40] S. Whitman, "Dynamic Load Balancing for Parallel Polygon Rendering," *IEEE Computer Graphics and Applications,* vol. 14, no. 4, pp. 41-48, July 1994.
[41] T. Whitted and D. Weimer, "A Software Test-Bed for the Development of 3D Raster Graphics Systems," *ACM Computer Graphics,* vol. 15, no. 3 pp. 271-277, Aug. 1981.
[42] B. Rivard, S. Winner, M. Kelley, B. Pease, and A. Yen, "Hardware Accelerated Rendering of Antialiasing using a Modified A-buffer Algorithm," *ACM Computer Graphics,* pp. 307-316, 1997.
[43] C.M. Wittenbrink and A.K. Somani, "Time and Space Optimal Data-Parallel Volume Rendering Using Permutation Wrapping," *J. Parallel and Distributed Computing,* vol. 46, pp. 148-164, 1997.

Wai-Sum Lin received the BS and MPhil degrees from the computer engineering program, jointly offered by the Deptartment of Computer Science and Information Systems and Department of Electrical and Electronic Engineering at the University of Hong Kong (HKU) in 1997 and 1999, respectively. He is currently a research assistant in the High Performance Computing Research Laboratory of HKU. His research interests include graphics rendering, antialiasing, and parallel and distributed computing.

Rynson W.H. Lau received the BSc degree in computer systems engineering in 1988 from the University of Kent at Canterbury, UK, with a first class honor and the PhD degree in computer graphics in 1992 from the University of Cambridge, UK. During his study, He received a few awards and scholarships from both universities. Prior to joining the City University of Hong Kong in 1998, he taught at the Hong Kong Polytechnic University. From 1992 to 1993, he worked at the University of York, England, on a defense project on image processing. He is a member of the ACM and of the IEEE Computer Society.

Kai Hwang received the PhD degree from the University of California at Berkeley. He is a professor and director of the Internet and Cluster Computing Laboratory at the University of Southern California (USC). This work was carried out at the University of Hong Kong during his leave from USC. An IEEE fellow, he specializes in computer architecture, parallel processing, and Internet computing. His latest books: *Advanced Computer Architecture* (McGraw-Hill, 1993) and *Scalable Parallel Computing* (McGraw-Hill, 1998) are widely used as standard college texts. He has lectured worldwide and performed consulting work for many high-tech companies and research organizations. His current research interest lies in network-based multicomputer clusters, Internet security, and middleware for cluster availability and single-system-image services in e-commerce and collaborative engineering design.

Xiaola Lin received the BS and MS degrees in computer science from Peking University, Beijing, China, respectively, and the PhD degree in computer science from Michigan State University, East Lansing, Michigan in 1992. He is currently with the Department of Electric and Electronic Engineering, the University of Hong Kong. He has published several papers on wormhole routing for various multicomputer networks. His research interests include wormhole routing, parallel and distributed computing, design and analysis of algorithms, and high-speed computer networks.

Paul Y.S. Cheung received the BSc (Eng) degree with first class honor in 1973 and PhD degree in 1978, both in electrical engineering from the Imperial College of Science and Technology, University of London. After working for Queen's University of Belfast for two years as a laboratory engineer, he joined the University of Hong Kong in 1980 and is now a senior lecturer/associate professor. He served as the dean of Engineering at the University of Hong Kong from 1994-2000. He was the IEEE Asia Pacific Director in 1995-96 and served as the IEEE Secretary in 1997. His research interests include parallel computer architecture, Internet computing, VLSI design, signal processing, and pattern recognition. He is a senior member of the IEEE.