A Loosely-Coupled Parallel Graphics Architecture Based on a Conflict-Free Multiport Frame Buffer

Satoshi Nishimura, Ryo Mukai, and Tosiyasu L. Kunii

Department of Information Science
Faculty of Science
The University of Tokyo
Hongo 113 Japan

Abstract

This paper describes a parallel computer architecture for real-time image synthesis. Our architecture is based on a loosely-coupled array of general-purpose processors equipped with a novel frame buffer subsystem called a conflict-free multiport frame buffer (CFMFB) which enables every processor to write any region of the screen without access conflicts. An efficient polygon rendering method using the CFMFB is also described. The method assigns a subset of the polygons to each processor, which independently calculates the images of the assigned polygons with the Z-buffer algorithm. The performance of our system is estimated through simulation experiments with sample scenes.

1 Introduction

Graphics systems are used in a variety of applications including computer aided design, scientific visualization, medical engineering and virtual reality systems.

There are two major requirements for such a graphics system. One is the rapid generation of 3D shaded images. Ideally, approximately 30 frames should be generated per second in order to provide a good man-machine interface or to produce computer animation videos quickly. This means that 1.5 million polygons need to be processed per second to render moderately complex scenes composed of 50,000 polygons. Today's graphics machines however can hardly achieve this performance.

The other requirement is flexibility, that is, the applicability to wide range of image generation algorithms. The optimal image generation algorithm varies from application to application. Moreover, some application uses several algorithms simultaneously to resolve the trade-offs between computation time and image quality.

The primary design goal of our graphics system is to meet both the above two requirements. Again, there are trade-offs. A single general-purpose processor with a typical frame buffer might be the most flexible system but does not satisfy the speed requirement. On the other hand, systems with algorithm-specific rendering hardware are superior in speed but the flexibility is sacrificed. Therefore, we have to find the best balance of the performance and the flexibility.

Our solution to the above is an array of general-purpose processors with a frame buffer accessible from all the processors. The only algorithm-specific component in our machine is the Z-buffer: all the other components are independent of image generation algorithms. The performance degradation due to the lack of special rendering hardware is overcome by the use of higher parallelism.

2 Background

2.1 Parallel Polygon Rendering

In this paper, we treat with only the polygon rendering through the Z-buffer algorithm as the image generation method, although our architecture is applicable to other algorithms as well.

The polygon rendering task can be divided into two stages: *geometric calculation* and *rasterization*. The geometry calculation determines the 2D screen coordinates and the color for each polygon vertex[1] from the 3D model coordinates and the normal vectors. This includes the multiplication of transformation matrices, clipping, and lighting calculations. The rasterization

[1] This is in case of the Gouraud shading. If the Phong shading is used, normal vectors are passed instead of the colors.

0-8186-2755-7/92 $03.00 © 1992 IEEE

computes the color for each pixel by filling the polygons with hidden surface removal.

There are three different approaches to parallelize the polygon rendering tasks: pixel parallel, polygon parallel, and function parallel. The pixel parallel approach assigns a non-overlapping subregion of the image space to each processor. On the other hand, in the object parallel approach, a subset of the polygons is assigned to each processor. The function parallel approach divides the task into different kind of subtasks and allocates them to distinct processors. The geometric calculation can be parallelized either by object parallel or by function parallel. As for the rasterization, all the approaches are possible, but generally either the pixel or polygon parallel approach is applied.

It is depending on various conditions such as the number of polygons which parallelization approach is superior in the rasterization. The pixel parallel approach is advantageous in that the frame buffer access conflicts can be avoided by using the *distributed frame buffer* [1, 2] in which each processor has only a fixed portion of the whole frame buffer. However, all the results of the geometric calculation (i.e. the screen coordinates and colors of polygons) have to be broadcasted to every processor for each frame generation, and therefore, communication overheads are serious if the number of the polygons is large. On the other hand, the object parallel approach allows both the geometric calculation and the rasterization to be performed in the same processor, and thus avoids the broadcasting overheads. Nevertheless, since the frame buffer access from each processor may overlap on the screen space, a single common frame buffer tends to be used. In this case, the access conflicts of the frame buffer are serious. However, in our opinion, the access conflicts can be avoided and the polygon parallel approach yields better results in most cases.

2.2 Previous Architectures

A number of parallel architectures for polygon rendering have been proposed.

Pixel-Planes 5 [3] consists of up to 32 programmable processors and up to 16 rendering units interconnected by a high-bandwidth ring network. The programmable processors perform the geometric calculation in object parallel. The rendering units are special hardware for the rasterization which evaluates a quadratic expression in parallel for every pixel.

IRIS [4] is one of the most popular graphics workstations developed by Silicon Graphics. It has special hardware for polygon rendering which includes geometry engines performing the geometric calculation in function parallel and 20 image engines executing Z-buffer operations or region clearing in pixel parallel.

Several researchers have developed polygon rendering systems called *object processor pipelines* which make use of the object parallelism both in the geometric calculation and the rasterization [5, 6, 7, 8]. These systems consist of a linearly connected array of customized processors each of which (1) receives a pixel value (color and Z-value) from one neighbor, (2) modifies it by the allocated polygon's color if the pixel is inside the polygon and visible from the eye, and (3) sends the resultant value to the other neighbor.

In contrast to the above-mentioned systems, there exist graphics machines without customized hardware. The Cellular Array Processor (CAP) [1] developed by Fujitsu is a multi-computer system composed of 64 (more recently 256) general-purpose processors interconnected by both a common bus and point-to-point communication links. The AT&T Pixel Machine [2] uses up to 82 programmable digital signal processors, and like the CAP, it emploies both a broadcast bus and point-to-point links for interprocessor communication. Both of these machines have the distributed frame buffer and rasterize polygons in pixel parallel. The geometry calculation is performed in object parallel (CAP) or in function parallel (Pixel Machine).

The systems listed above have one or more of the following problems.

- **Lack of flexibility.** Systems with customized rendering hardware can be applied only to a limited class of rendering algorithms.
- **Serial traversal of object database.** As pointed out in [9], systems storing the entire object database in a single memory have performance limitation due to the bandwidth from the memory. This becomes serious when the performance exceeds 1 million polygons per second.
- **Broadcasting overhead.** As explained before, rasterizing polygons in pixel parallel requires the entire object information broadcasted for each time of image generation. The broadcasting overhead is serious if the number of polygons is large.
- **Limitation in parallelism.** In systems using the pixel parallel approach in the rasterization, part of the processing power is wasted because the pixels assigned to each processor are partially outside polygons and do not contribute to the resulting image. This is especially true when rendering small polygons or vectors. Supposing that we render a scene with more than 10000 polygons

onto a 1024 by 1024 screen, the average area of the polygons on the screen will normally be less than 100 pixels, and therefore, we believe a 64 processors system will no longer be practical.

- **Difficulty in load balancing.** In such a system that the geometry calculation and the rasterization are performed in separate hardware units, it is difficult to balance the loads between the units, because the computation time ratio between the two tasks varies depending on conditions such as polygon size or the number of vertices.

In the following chapters, we will describe a graphics system free from the above problems.

3 The VC–1 Architecture

This section introduces a graphics machine called VC–1. VC–1 is still under construction; currently only a prototype system with a single processor is completed.

3.1 Overview

VC–1 is a loosely-coupled multiprocessor with a novel frame buffer subsystem called a conflict-free multiport frame buffer (CFMFB) which enables every processor to write any region of the screen without access conflicts. Figure 1 illustrates the overall organization of VC–1.

Each processing element contains the Intel i860 CPU [10], a 64-bit RISC-based microprocessor with both high-speed floating point units and a graphics unit executing several graphics-oriented instructions. In our current plan, the system consists of 64 processors at 40MHz clock each with 8-Mbyte local memory. The peak performance will be 5.1 GFLOPS for single-precision data and 3.8 GFLOPS for double-precision data.

The above processors are interconnected in the hypercube topology by point-to-point communication links which are compatible with those of Inmos Transputers [11]. The bandwidth of each link is 2 Mbytes/sec. Employing the hypercube topology is not essential: simpler topology like the 2D mesh is enough for polygon rendering. Nevertheless, in order to experiment with various kinds of topologies against many algorithms, we chose the hypercube which includes many types of topologies as sub-networks.

The most remakable feature of our architecture is scalability. The average access time of the CFMFB is nearly constant against the number of processors. This fact along with the absence of global busses guarantees the possibility of the almost linear speed-up of the system even with a thousand of processors.

3.2 Conflict-Free Multiport Frame Buffer

The *conflict-free multiport frame buffer* (CFMFB), the key component proposed by us, is such a frame buffer that every processor can write any region of the screen without access conflicts. This scheme enables us to perform the polygon-parallel rasterization as well as the pixel-parallel rasterization.

In generating images, each processor can compute its own subimage without reading the subimages created by other processors. Taking advantage of this nature, we solved the access conflicts.

3.2.1 Structure of the CFMFB

The CFMFB consists of *local frame buffers* (LFB's), a *pipelined image merger* (PIM), and a *global frame buffer* (GFB). The connection of these components are depicted in Figure 1. The LFB exists for each processor and holds the subimage (including Z-values) created by the corresponding processor. By using a method similar to the well-known virtual memory technique, the LFB virtually holds the pixel information of the entire screen. The PIM periodically superimposes the subimages stored in the LFB's and transfers the merged picture to the GFB. The Z-values are taken into account when the images are merged. The GFB really holds the pixel information (including Z-values) of the entire screen.

Local Frame Buffer

As the performance of processors increases, the frame buffer bandwidth becomes more and more important in fast image generation. Therefore it is desirable to use high-speed memory chips for frame buffers. It is, however, not cost effective to prepare high-speed memory of the full screen capacity for all of the LFB's. To reduce the memory size, we divide the screen into equal-sized rectangle regions called *patches*, and allocate memory only to the patches accessed by processors (Figure 2). Because of frame buffer access locality, the required memory size is considerablely reduced with this technique.

Figure 3 shows the block diagram of an LFB unit. The color and Z-value for each pixel are stored in the *image memory* which consists of high-speed static RAM chips. In our current plan, the capacity of the

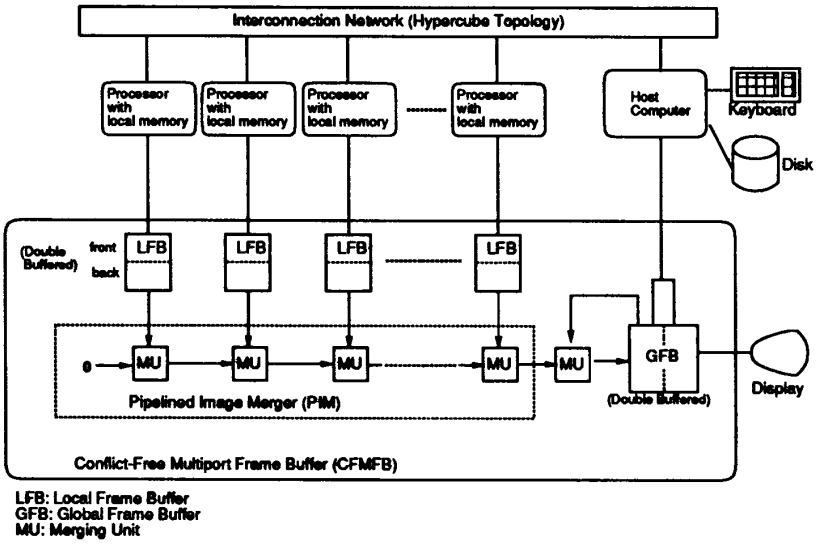

Figure 1: The structure of VC–1

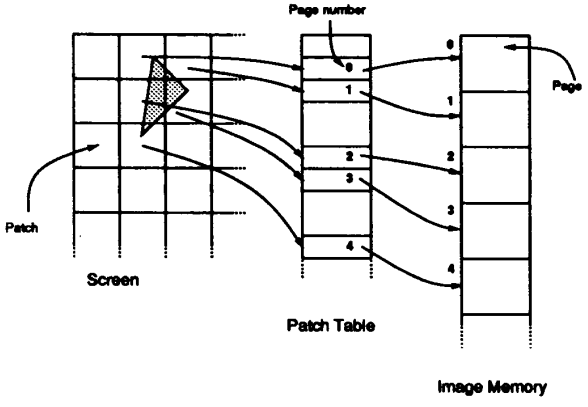

Figure 2: Patches and image memory allocation

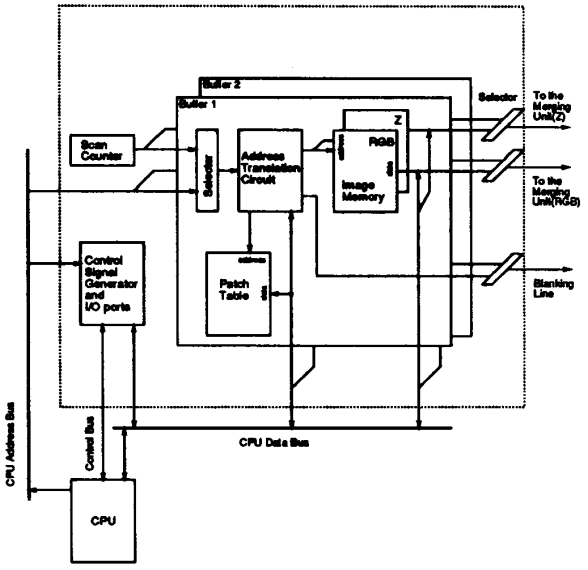

Figure 3: Local frame buffer (LFB)

image memory is one eighth of the full screen capacity. The address space of the image memory is divided into *pages* each of which has the capacity enough for a patch.

The *patch table* (PT) is a high-speed RAM maintaining the status for all patches. Each entry of the patch table consists of two fields. The first field is a 1-bit flag indicating whether an image memory page is allocated for the patch. If the flag is true, the second field contains the image memory address of the patch.

The *address translation circuit* (ATC) tied with the PT converts X-Y screen coordinates (virtual address) to the image memory address (physical address). The ATC is implemented by a set of selectors so that the size of patches is programmable (from 8×2 to 256×256, limited to a power of 2).

The *scan counter* controls the transmission from the LFB to the PIM. The counter value starts from the upper-left corner of the screen and ends at the lower-right corner. The scan counter never stops, i.e., when the counter reaches the end, it is reset, and then the next cycle is initiated at once. The values of the scan counters among LFB's are skewed in order to realize pipelined image merging.

LFB's are double-buffered, i.e., consist of two independent sets of frame buffers, namely, a front buffer and a back buffer. The front buffer is a part of the processor main memory, and freely accessible from the processor. The back buffer is read out according to the scan counter and its contents are transmitted to the GFB via the PIM pixel by pixel. The roles of the two buffers are swapped at each end of image generation.

In the front buffer, the action taken when the processor accesses the LFB is as follows. First, the patch index is determined, which is then used in reading the patch table entry. If the patch has an associated image memory page, the processor immediately accesses the image memory; otherwise, first a page is allocated, and then accessed. The latter situation is referred to as *patch misshit*.

There are two ways to handle the patch misshit. One is to use some special hardware for this. We have designed a simple page allocating circuit, which sequentially allocates image memory pages without increasing the frame buffer access time. The other way is to implement the page allocation task in software by interrupting the processor when the patch misshit occurs. Although this approach imposes context switching overheads, it enables us to employ more sophisticated page allocating strategies.

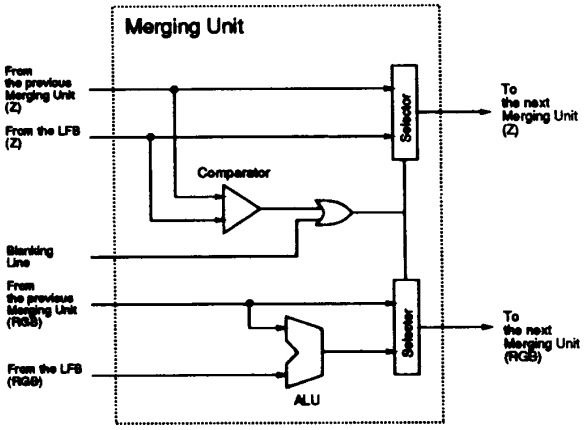

Figure 4: Merging unit

Pipelined Image Merger

The PIM is a linearly connected array of per-processor elements called *merging units* (Figure 4). Each merging unit takes two pixel values as its inputs and merges them into one. Let the input pixel value from the left merging unit in Figure 1 be represented as p, which is a vector consisting of RGB color intensities (p_r, p_g, p_b) and a Z-value (p_z). Also, we represent the pixel value from the upper LFB as q.

The merging operation in the merging unit is characterized by a merging operator $\oplus$ defined by one of the following expressions:

1. Z-comparison mode

$$p \oplus q \equiv \begin{cases} p & \text{if } p_z < q_z \text{ or BL} \\ q & \text{otherwise} \end{cases}$$

2. addition mode

if not BL

$$\begin{aligned} (p \oplus q)_r &\equiv \min(p_r + q_r, r_{max}) \\ (p \oplus q)_g &\equiv \min(p_g + q_g, g_{max}) \\ (p \oplus q)_b &\equiv \min(p_b + q_b, b_{max}) \\ (p \oplus q)_z &\equiv \text{undefined} \end{aligned}$$

otherwise

$$p \oplus q \equiv p$$

where BL is the state of the *blanking* line from the LFB indicating that there are no corresponding image memory pages, and $r_{max}, g_{max}, b_{max}$ are the maximum possible color intensities. The Z-comparison

mode is suitable for the Z-buffer algorithm, and the addition mode is appropriate for ray tracing or other image generation algorithms.

The merging process is performed in pipelined fashion. For the sake of convenience, let us define the following symbols.

n : number of processors in the system
M : horizontal screen size
N : vertical screen size
LFB_i : i-th LFB from the left in Figure 1 ($1 \leq i \leq n$)
MU_i : merging unit connected to LFB_i
y_i : output of MU_i
$p^i_{x,y}$: pixel value in LFB_i at screen position (x, y) $(1 \leq x \leq M, 1 \leq y \leq N)$

At the beginning, MU_1 merges $p^1_{1,1}$ with null pixel value o ($o_r = o_g = o_b = 0, o_z = \infty$), which makes y_1 be $p^1_{1,1}$. In the next step, MU_1 calculates $p^1_{2,1}$, and at the same time MU_2 computes $p^2_{1,1} \oplus y_1$. Thus, MU_n outputs the continuous stream of a merged picture as follows:

$$\bigoplus_{i=1}^{n} p^i_{1,1}, \bigoplus_{i=1}^{n} p^i_{2,1}, \cdots, \bigoplus_{i=1}^{n} p^i_{M,1}, \bigoplus_{i=1}^{n} p^i_{1,2}, \cdots, \bigoplus_{i=1}^{n} p^i_{M,N}$$

The time needed for merging the contents of all the LFB's into the GFB (t_d) is $M \cdot N \cdot p + (n-1) \cdot p$ where p is the pipeline pitch. The first term is the time for scanning the entire screen and the next term is the pipeline delay. In the current implementation, $M = 640, N = 400, n = 64$, and $p = 60$nsec, then $t_d = 15.4\text{msec}(\approx 1/65\text{sec})$.

Global Frame Buffer

There are three reasons for inserting the GFB between the PIM and the CRT display:

- separating the scan rate of the PIM from that of the CRT,
- continuing the image composition even during the blanking periods of the CRT scan, and
- handling the LFB overflow explained in Section 3.2.2.

The GFB is double-buffered, i.e., it consists of two independent full-screen frame buffers with Z-buffering, one for CRT refresh and the other for storing the images from the PIM. Whenever an image generation is completed, the roles of these buffers are alternated.

The host computer can directly access the GFB for system maintenance or other purposes.

The GFB operates in one of the two modes, namely, overwriting mode or accumulative mode. In the overwriting mode, the pixel value from the PIM is stored without modifications. In the accumulative mode, the pixel value is merged with the old value having stored in the GFB, and then, the resultant pixel value is written. For this purpose, the GFB has its own merging unit same as the one in the PIM. The GFB mode is switched during image generation to realize the fast screen clearing described in Section 3.2.3.

3.2.2 LFB overflow handling

When a patch misshit occurs in an LFB, it is possible that no more free pages are left in the image memory. We call this situation LFB overflow.

When the LFB overflow occurs, the processor waits until the whole contents of the back buffer are transferred to the GFB and then swaps the LFB. Since the GFB can operate in the accumulative mode, once the contents of the back buffer are transferred to the GFB, we can reuse the buffer for storing a different region of the screen.

In the worst case, the processor have to wait for $M \cdot N \cdot p$ duration, which is the time necessary for transmitting the whole buffer. However, such a case seldom occurs because the image generation overlaps with the transmission. In most cases, the transmission is already completed when a patch misshit occurs, and thus no access delay is imposed.

3.2.3 Fast Screen Clearing

Prior to generating an image, the screen must be cleared. The CFMFB provides a way for clearing the screen very quickly.

The explicit clearing of the GFB can be avoided in the following way. After the GFB is swapped, the mode of the GFB is put in the overwriting mode during the first round of the image merging scan so that the pixel values from the PIM overwrite the old contents of the GFB. When the first round is completed, the GFB mode is switched to the accumulative mode.

LFB's are cleared as follows. As for the PT, the processor must clear all the patch table entries after each LFB swapping. On the other hand, the image memory is automatically cleared while it functions as the back buffer: when a pixel value in the back buffer is read out according to the scan counter, the pixel value is cleared at once using the read-modify-write cycle. Thus, after an LFB swap, the image memory is guaranteed to be cleared.

4 Simulation Results

To estimate the polygon rendering performance of VC–1, we developed a simulator program which imitates the system behavior of VC–1. This section first gives an overview of our parallel polygon rendering method and then shows the simulation results.

In our polygon rendering method, the polygon parallel approach was taken in both the geometric calculation and rasterization processes.

The outline of the routine for rendering a frame is as follows. First, scene database is distributed from the host computer to each processor's local memory. Polygon data is partitioned into non-overlapping subsets, each of which is assigned to a distinct processor. Other database components, such as the eye position, are duplicated to all the processors' local memory. Next, the host computer broadcasts a DRAWSTART packet to all the processors. After receiving the DRAWSTART packet, each processor starts geometric calculations for each of the assigned polygons, and finally stores rasterized images to the processor's own LFB. Since there are no conflicts in the frame buffer access, the processor can create the images independently of others. When the processor completes the image generation, it swaps the LFB and then returns a DRAWEND packet to the host computer. After the host computer receives the DRAWEND packets from all the processors, the GFB is swapped, and thus the rendering process is completed.

In generating consecutive frames, the scene database is incrementally modified, i.e., only interframe difference of the database is transmitted between frames rather than reloading the whole database.

The simulation program was run on Intel i860 Station (33MHz clock version) under the UNIX operating system. The outline of our simulation method is as follows. We define an *event* as either a trap (system call) or an interruption in each processor. The simulation program runs each task in each processor from an event to the next event in turn. The time between the events is measured by a real-time clock, and therefore, results must be coincide with those on multiprocessors with 33MHz i860's.

In the simulation, we assumed the following conditions:

Communication speed of links:	1.25Mbytes/sec
Context switching overhead:	2μsec
Screen size:	640×400 pixels
Patch size:	8×8 pixels
Capacity of LFB image memory:	32000 pixels

Table 1: Estimated rendering time (single teapot)

No. of processors	Rendering time (msec)	Speed up	Average LFB-overflows
1	440.7	1.0	0
3	152.5	2.9	0
7	73.5	6.0	0
15	37.7	11.7	0
31	21.7	20.3	0
63	12.9	34.2	0

Table 2: Estimated rendering time (16 teapots)

No. of processors	Rendering time (msec)	Speed up	Average LFB-overflows
1	6496.8	1.0	7
3	2202.1	3.0	4.7
7	971.8	6.7	2.3
15	461.3	14.1	1.0
31	241.8	26.9	0.2
63	131.5	49.4	0

LFB read/write cycle time:	500nsec
PIM transmission period:	17.93msec

Tables 1 and 2 show the results obtained by our simulator. For Table 1, we used a sample scene with one Gouraud-shaded teapot composed of 4096 triangles. For Table 2, a relatively large scene including 16 identical teapots (containing totally 65536 triangles) was used. The rendering time is defined as the time since the host computer has broadcasted the DRAWSTART packet until the host computer has received all the DRAWEND packets. The 'Average LFB overflows' field means the average number of LFB overflows in each processor.

As the results show, in case of 16 teapots, our approach is so efficient that a 63 processors' system can draw 500K triangles per second. However, parallelism is limited in case of a single teapot. This is because the polygons are clustered in hundreds and therefore there exist completely idle processors if the number of processors exceeds 41. The polygon clustering is necessary for increasing frame buffer access locality and thus decreasing LFB overflows. Although we could enhance the processor utilization by reducing the size of clusters, our current rendering program can not efficiently handle small clusters due to processing overheads increasing in proportion to the number of clus-

ters. In future, we will optimize the program so as to deal with smaller clusters.

The number of LFB overflows is zero in the single teapot's case because the original image occupies less than one eighth of the full screen. In case of 16 teapots, LFB overflows often occur, but the frame buffer access is not suspended at all since the interval time of the LFB overflows is far longer than the transmission period of the PIM.

5 Concluding Remarks

This paper proposed a novel frame buffer structure for multiprocessor systems called a conflict-free multiport frame buffer (CFMFB), and described a machine for real-time image synthesis which consists of an array of general-purpose processors and the CFMFB. Through simulation experiments, we investigated the polygon rendering performance of the machine. From the results obtained, it was found that the machine efficiently renders polygons even at its maximum configuration, that is, a 64 processors' system.

In future, we are planning to do the followings:

- performance comparison with previous architectures for image synthesis,
- investigating whether the CFMFB is effective in other image generation methods such as ray tracing or volume rendering,
- in order to allow each processor to read the contents of the GFB, inputing the GFB contents from the tail of the PIM and loading a part of the whole image to each LFB, which make the machine suitable for image processing applications, and
- implementing efficient visual simulation by performing both simulation tasks and visualization tasks in the same multiprocessor.

Acknowledgements

We would like to thank Mr. Hiroyuki Nitta of Kubota Computer Inc. for his valuable comments and assistance in gathering the equipments and materials necessary for building the prototype. This work is supported by Kubota Computer Inc.

References

[1] H. Sato, M. Ishii, K. Sato, M. Ikesaka, H. Ishihata, M. Kakimoto, K. Hirota, and K. Inoue, "Fast Image Generation of Constructive Solid Geometry Using a Cellular Array Processor," *ACM Computer Graphics*, vol. 19, pp. 95–102, July 1985.

[2] M. Potmesil and E. M. Hoffert, "The Pixel Machine: A Parallel Image Computer," *ACM Computer Graphics*, vol. 23, pp. 69–78, July 1989.

[3] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel, "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories," *ACM Computer Graphics*, vol. 23, pp. 79–88, July 1989.

[4] K. Akeley and T. Jermoluk, "High-Performance Polygon Rendering," *ACM Computer Graphics*, vol. 22, pp. 239–246, August 1988.

[5] D. Cohen and S. Demetrescu, "A VLSI approach to Computer Image Generation," tech. rep., Information Sciences Institute, University of Southern California, 1981.

[6] G. C. Roman and T. Kimura, "VLSI perspective of real-time hidden-surface elimination," *Computer-Aided Design*, vol. 13, pp. 99–107, March 1981.

[7] U. Claussen, "Parallel Scanconversion," in *Proc. of IFIP Working Group 10.3 Conference on Parallel Processing*, pp. 125–137, June 1988.

[8] M. Deering, S. Winner, B. Schediwy, C. Duffy, and N. Hunt, "The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics," *ACM Computer Graphics*, vol. 22, pp. 21–30, August 1988.

[9] D. Ellsworth, H. Good, and B. Tebbs, "Distributing Display Lists on a Multicomputer," *ACM Computer Graphics*, vol. 24, pp. 147–154, March 1990.

[10] L. Kohn and N. Margulis, "Introducing the Intel i860 64-Bit Microprocessor," *IEEE Micro*, vol. 9, pp. 15–30, August 1989.

[11] Inmos Ltd., *Transputer Reference Manual*. Prentice Hall, 1988.