The Metabuffer: A Scalable Multiresolution Multidisplay 3-D Graphics System Using Commodity Rendering Engines *

William Blanke
Department of Electrical and Computer Engineering
Chandrajit Bajaj, Donald Fussell, and Xiaoyu Zhang
Department of Computer Sciences
and Texas Institute for Computational and Applied Mathematics

The University of Texas at Austin
Austin, TX 78712

Abstract

Many computer graphics applications require multiple visual displays that are rendered in real time. However, dividing and balancing the rendering workload among a number of CPUs and then sending it to the individual displays efficiently is a difficult task. This report details the design and software prototype for a novel new type of graphics framebuffer that takes in RGB color, Z order, and alpha information from multiple COTS rendering engines, collages them, and then outputs composited images to multiple displays in a tiled configuration. The individual COTS rendering engines define their image data as viewports that can be located anywhere in the display space and be of any size/resolution. For example, one graphics board could generate a low-resolution background image spanning multiple displays while other rendering engines could concentrate on more detailed images in the foreground. This flexibility lends itself to multiple display virtual reality arrangements and foveated vision systems. This report details the architecture of the Metabuffer and the results from a multithreaded software prototype written in C++. It shows that the Metabuffer provides a simple framework to its host PCs for effective utilization of multiple visual displays using COTS rendering engines and produces constant time image collaging with a minimum of latency regardless of geometry or number of image viewports.

1 Introduction

A major technology trend over the past several decades has been the increasing integration and complexity of commodity digital system components. Personal computer hardware and software has made the construction of systems with entire computers as components very cost-effective. Given the need for large-format immersive displays such as videowalls or the CAVE [4] for scientific visualization and other VR applications, a natural approach is to take advantage of this trend by building cost-effective multiprojector displays driven by commodity PCs with commodity 3-D rendering hardware. There are two principal advantages to such systems. First is the provision of large-format displays, and second is the opportunity to do rendering in parallel on a number of commodity machines [15].

Parallel realtime rendering in a multidisplay environment must address the same concerns that have faced graphics architects since the early flight simulator systems of the 60's, and the architectural strategies are the same. Since a multidisplay image consists of a number of *tiles*, each of which is the image produced by one display device, a natural strategy is simply to connect each display to a single machine with rendering hardware to drive it. Thus, the global display space is partitioned into tiles, each of which is driven by a dedicated framebuffer and rendering pipeline. These machines can be connected via a high-speed network, perhaps along with other machines not directly connected to displays, to produce a parallel rendering platform. The key problems in rendering in such an environment involve synchronizing the renderers and load-balancing the computation in the face of the constraints imposed by the screen-space partitioning. However, such an approach requires little or no custom hardware and can produce a composite image of the same quality as that produced by each component tile by a single rendering engine.

An alternative approach based on image composition, which dates from flight simulator systems of the 60's, is to allow each machine in the network to render to any part of the image. The resulting individual images must then be composited to form a correct overall image through the use of a pipelined compositing network that takes the outputs of the individual frame buffers along with depth information in scan-line order and does hidden-surface calculations on a pixel-by-pixel basis to produce a correct composite image. This approach has the advantage of removing the constraints on load balancing imposed by screen-space partitioning, but requires custom compositing hardware. To provide high-quality images, the compositing hardware must do more than just hidden-surface removal. Some scheme for antialiasing the results and color correction at the borders of tiles must also be provided.

We are building a scalable multidisplay system, the Metabuffer, that takes the image composition approach and allows off-the-shelf PCs with 3-D rendering boards to be used with no modifications to the stock hardware. Our Metabuffer hardware supports a scalable number of PCs and an independently scalable number of displays– there is no *a priori* correspondence between the number of renderers and the number of displays to be used. It also allows any renderer to be responsible for any axis-aligned rectangular viewport within the global display space at each frame. Such viewports can be modified on a frame-by-frame basis, can overlap the boundaries of display tiles and each other arbitrarily, and can vary in size up to the size of the global display space. Thus each machine in the network is given equal access to all parts of the display space, and the overall display space is treated as a uniform display space, that is, as though it were driven via a single, large framebuffer, hence the name Metabuffer. Because the viewports can vary in size, our system supports multiresolution rendering, for instance allowing a single machine to render a background at low resolution while other machines render foreground objects at much higher resolution. Thus our system could allow *foveated* display for multiple users simultaneously if gaze tracking were done for all of these

*Research supported in part by grants from the National Science Foundation ACI-9982297, DMS-9873326, and a grant from the Texas Higher Education Coordinating Board TARP-003058-0847-1999

users.

In the rest of the paper, we will elaborate on the advantages and disadvantages of image composition approaches for parallel rendering, describe our system in more detail, show results from our current software simulator, and describe our project's status and future plans.

2 Background and Related Work

The fundamental problem in parallel real-time rendering [2, 1] has always been the need for parallel access to frame-buffer memory. This is just as true of multiprojector systems as it is for high-performance single-display systems. As indicated in [10], parallel rendering schemes can be classified according to where in the graphics pipeline they perform the critical hidden-surface computations. Three approaches were identified: sort-first, sort-middle and sort-last. While high-end graphics computers such as those developed by SGI use a sort-middle approach, sort-first and sort-last approaches are most appropriate for use in a system comprising parallel commodity computers.

In the sort-first approach, the display space is broken into a number of non-overlapping display regions, or *tiles*, which can vary in size and shape. Each processor renders a set of tiles and ships resulting pixels over the network to the processor containing the frame buffer for display. The key problem is to determine a set of tiles that balance the rendering load for the processors in each frame. Since the load generally varies from frame to frame, the tiling algorithm must be part of the computation performed at each frame time, in addition to the normal rendering work done by each machine. For efficiency, surface elements that overlap tiles are rendered by the machines responsible for each of the overlapping tiles, an overhead that increases as tile size decreases. Since load balancing can be done most effectively using many small tiles, there is a tradeoff between effectiveness of load balancing and the overhead of redundant rendering of surface elements that determines an optimal tile size. In addition, the tiling algorithm and the network must be fast to limit the parallel processing overhead. Such a scheme can be used for parallel rendering to a single display or to a video wall composed of multiple projection displays. The number of processors that are recipients of rendered tiles is equal to the number of projection displays used. This is the approach taken to scalable three dimensional graphics in the SHRIMP project [13].

In the sort-last approach, any processor can render any subset of the surfaces in the scene to any portion of the display space in parallel with similar computations by other processors. This makes the load balancing problem easier since screen space constraints are removed. Thus, no surface need be rendered more than once, and any surface element can be assigned to any processor. However, since the pictures computed by the processors overlap in this scheme, compositing hardware is needed to combine the output of the various processors into a single correct picture. The hardware must also assure that the quality of the composite image is similar to the image quality produced by the individual renderers. Such approaches have been used since the 60's in single-display systems [3, 6, 11, 12, 5, 14], and more recent work includes [8].

3 Our Approach

We have chosen to implement a sort-last scheme with several unique features. We will provide a uniform display space across a scalable number of projection displays, thus allowing any processor in our parallel system to have equal access to all parts of the display space, regardless of how many individual display devices are used. We will provide a simple abstraction to the rendering software in which each rendering system can declare its own frame buffer to be a viewport within the large display space realized by the union of all the display devices. Not only does this provide a very simple and uniform way to handle display resources in software, but it also provides for *multi-resolution* display. This is because the resolution of the final display depends on the size of the viewport. For example, one machine could render the entire background of the scene a low resolution by simply setting its display space viewport to the entire screen, while other processors could produce very high quality supersampled images of important foreground objects by setting their viewports to cover small regions of the display space. As another example, our system could be used for *foveated display*, by tracking the viewers' gaze and allocating processors with small viewports to render regions of the screen around the direction(s) of view at high resolution, while allocating a few processors with large viewports to render regions far from the viewers' gaze direction(s).

We have also devised a way to extract depth information from commodity PC rendering boards without changing the stock hardware, along with a simple control interface for our system, all based on the new generation of commodity digital video flat-panel output formats being released in PC display cards.

Due to the abstraction of a uniform display space that our hardware provides to the parallel renderers, we call this portion of our system the Metabuffer since it performs a function analogous to the frame buffer in a single-renderer system.

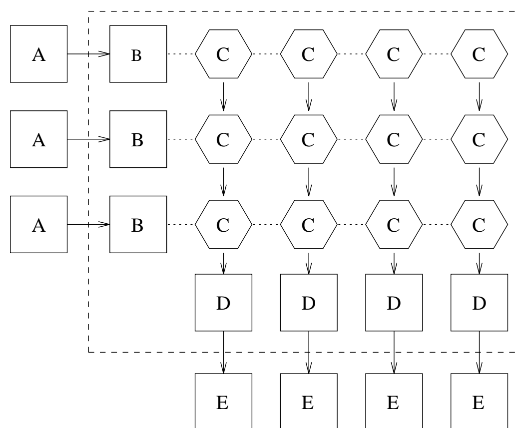


Figure 1: Metabuffer Architecture

4 System Organization

The architecture of the Metabuffer presents a number of challenges. The most difficult problem is the large amount of data that must be processed. Each pixel needs RGB color, Z order, and alpha information. A single frame will have millions of these pixels. A real-time rendered animation will need to display approximately 30 frames per second in order to be fluid and smooth. Multiply all of this by several rendering engines and several output displays and the large quantities of data involved are clearly evident.

Figure 1 shows how a Metabuffer architecture using three rendering engines and four output displays utilizes multiple pipelined data paths and busses to surmount this problem. External to the board, COTS rendering engines (A) deliver their data to onboard framebuffers (B) by means of the recently adopted industry standards for digital video transmission, the Digital Visual Interface (DVI). Since COTS rendering engines (A), at this time, transfer only 24 bits per pixel over these digital links, color is transferred on even screens, while alpha and Z information is transferred on

odd screens. At a refresh rate of 60 hertz, this is still fast enough to provide enough RGB, alpha and Z information for 30 frames per second. The onboard framebuffer (B) stores information from both transmissions in 64 bit wide memory. Control information, such as the location of the viewports and their final destination in the overall display, is stored on the first scanline of each rendering engine's image (A). This first scanline is never displayed. Instead, DSP code, viewport data, or anything else that is needed by the control logic of the frame buffer can be written here using standard OpenGL writepixel() calls.

When a full frame has been buffered, data is selectively sent over a wide bus to the composer units (C) based on viewport locations. The data will be streamed out in large blocks in order to take advantage of page mode memory access as much as possible. The composers (C) take only the data that is required to build their column's output image and ignore the rest. Each composer (C) then sends its data in pipeline fashion down the column to the next lower composer (C) so that the pixel Z order information can be compared with those Z values from the other COTS renderers (A). This way, only the front-most pixel is saved. The collaged data is then stored on another onboard framebuffer (D). These smart framebuffers (D) can perform post processing on the data for anti-aliasing and are also able to drive the offboard displays (E) again using the DVI specification.

5 System Operation

Encoded at the start of each rendering engine's image is control information that tells the input framebuffer which segments of the image should be sent to which composers and where they should be placed in the final display. This work is done by the computer hosting the rendering engine since it offloads the computational work to a full fledged CPU, which is more suited to this task than the streamlined Metabuffer. The control information is sent in tabular form, with one row corresponding to each image segment.

Dcomp	Sx	Sy	Sdx	Sdy	Dx	Dy	Dmultiple
1	0	0	75	75	25	25	1
2	75	0	25	75	0	25	1
3	0	75	75	25	25	0	1
4	75	75	25	25	0	0	1

The table above shows some typical data describing a viewport configuration (essentially the layout as described in section 5.1.2 later in this paper). Here, the image and display size are assumed to be 100 pixels by 100 pixels. Dcomp is the index number of the composer (or display) where the segment is to be sent. Sx and Sy refer to the source coordinates of the segment in the rendered image. Sdx and Sdy refer to the dimensions of the segment in the source image. Dx and Dy refer to the destination coordinates in the display image. Dmultiple is the replication factor of the source pixel. Since the ratio of source to destination pixels is 1:1, this multiple is 1. The input framebuffer broadcasts the entire viewport table over the bus to the composers at the start of each frame. Each composer then takes the entry that it is responsible for and stores it locally.

5.1 Analysis of Bus Data Flow

One of the most interesting problems of this project is how to efficiently transmit image data from the input framebuffers, through the bus, and then to each composer. Since the composers are arranged in a pipeline fashion, it is imperative that they have the data they need at the right time. If one composer is missing its data, a glitch in the image will occur.

Since the Metabuffer employs viewports of varying size and position, it is important to demonstrate that the bandwidth requirements of the composers will not exceed the limited data rate of the bus that connects them to the input framebuffers. If the bandwidth requirements are exceeded in certain viewport configurations, glitches in the output image are certain to occur. The analysis that follows proves that the Metabuffer has a constant bandwidth requirement regardless of the size or orientation of the viewports that are used.

In order to analyze the worst case data flow of the board, a scheme is used similar to the one presented in the paper by Kettler, Lehoczky, and Strosnider [9]. Since all data needs are periodic (because of the raster display), each task (display) can be described in terms of the amount of data needed (C), its period (T), and its deadline (D). By quantifying these values for some sample cases, it is easy to see that the bandwidth requirements do not change as the viewport geometry becomes more complex.

For example, if we assume that the smallest viewport is the size of an output screen (of w by w pixels), and that the viewports increase in size in even multiples, observations for the following cases hold true.

5.1.1 Case One

The input image is the same size as an output screen, but only one composer is used. The ratio of pixels from input to output is 1:1, so the composer requires a steady stream of data. As shown on the right, the total bandwidth required is one screen full.

Data	Period	Deadline
$C_1 = w$	$T_1 = w$	$D_1 = w$

This is the trivial case. The data needed (C) is equal to the period for the scheduling. A steady stream of data will satisfy this.

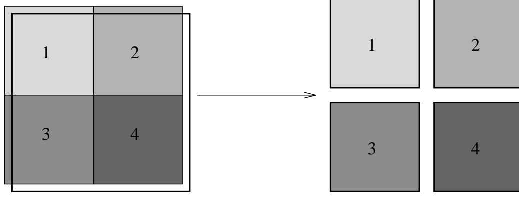
5.1.2 Case Two

Again, the input image is the same size as an output screen. However, in this case four different composers require data. But, according to the geometry of the display, only one composer will need data at any particular time. As shown on the right, none of the composer viewport areas overlap. They join together to form exactly one screen size. So, one screen size of data is needed. The ratio of pixels from input to output is 1:1, and there is no overlap, meaning only one pixel need be accessed on the bus at any one time.

Data	Period	Deadline
$C_{1a} = l$	$T_{1a} = w$	$D_{1a} = w$
$C_{1b} = w - l$	$T_{1b} = w$	$D_{1b} = w$

The variable l represents the vertical dividing line in the row between tasks 1a and 1b. For the purposes of scheduling, the horizontal divider is ignored, since this merely changes the display destination of the data, and not the data timing needs of the system. Adding all of the data values together (C) results in the same quantity as the period, which means the bandwidth is constant compared to the previous case.

5.1.3 Case Three

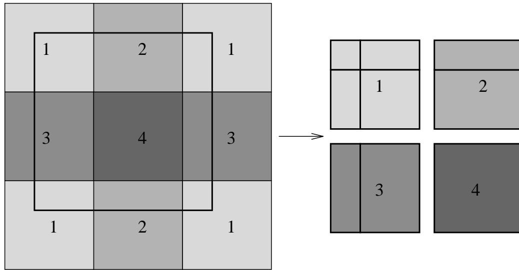


Here, the input image is four times as large in order to form a low resolution background display. In this case four composers will require data, but they will all require data at the same time! As shown on the right, four screenfulls of data are required. However, the saving grace here is that the ratio of input pixels to output pixels is 1:4. Thus, while four times the screens are being created, they are being furnished with one fourth of the data. This effectively means that the bandwidth requirements here are still constant. The fact that four composers require pixel data at the same time is a problem, but since the bandwidth requirements are scalable, a simple buffering scheme should satisfy each of the composers.

Data	Period	Deadline
$C_1 = w/2$	$T_1 = 2w$	$D_1 = 2w$
$C_2 = w/2$	$T_2 = 2w$	$D_2 = 2w$
$C_3 = w/2$	$T_3 = 2w$	$D_3 = 2w$
$C_4 = w/2$	$T_4 = 2w$	$D_4 = 2w$

Because pixels are being replicated to twice their size, the period (T) of the scheduling increases by a factor of two because there are half as many rows to process. Likewise, the data needed (C) decreases by a factor of two. If all of the C values are totaled, the result is $2w$, which is the same as the period.

5.1.4 Case Four



Finally, in case 4, the input image is again four times as large, but now it overlaps nine composers. From the right, it can be seen that from these nine composers, only four screens simultaneously need to be placed on the bus at the same time. And, from the analysis of case 3, because the ratio of pixels is 1:4, there is one-fourth the bandwidth requirement. Again, the bandwidth requirements remain constant. Since four composers must simultaneously have data, the bus must be buffered. Successive cases of larger viewports and more composers can be extrapolated in a similar manner.

Data	Period	Deadline
$C_{1a} = l/2$	$T_{1a} = 2w$	$D_{1a} = 2w$
$C_{1b} = (w - l)/2$	$T_{1b} = 2w$	$D_{1b} = 2w$
$C_2 = w/2$	$T_2 = 2w$	$D_2 = 2w$
$C_{3a} = l/2$	$T_{3a} = 2w$	$D_{3a} = 2w$
$C_{3b} = (w - l)/2$	$T_{3b} = 2w$	$D_{3b} = 2w$
$C_4 = w/2$	$T_4 = 2w$	$D_4 = 2w$

Because pixels are being replicated to twice their size, the period (T) of the scheduling increases by a factor of two because there are half as many rows to process. Likewise, the data needed (C) decreases by a factor of two. If all of the C values are totaled, the result is $2w$, which is the same as the period.

5.2 Buffering of Bus Data Flow

From a performance perspective, neglecting the needs of the composers, the bus should be buffered anyway. Using the onboard memory of COTS DSPs to buffer the bus allows the input framebuffer to send data efficiently using burst modes commonly available in current memory architectures. This way, an entire page of memory can be sent to each composer. While this will cause some latency at the beginning of each frame, it will not affect throughput. Instead, the buffer lets the composers operate at peak capacity and minimizes the risk of any glitches. More importantly, because the data is now buffered, the simultaneous data access of multiple composers is no longer a problem. They will be able to access their own local cache memory independent of what the other composers may be doing. Pixel replication data can also be stored locally, instead of needing to refer to the input framebuffer multiple times.

The buffer that each composer maintains closely resembles a queue, except for one important difference. While the buffer acts in a FIFO manner when Dmultiple is 1 (the source pixels and destination pixels are in a 1:1 ratio), if pixel replication needs to be done, it is necessary to remember data from the previous row. Therefore, the cache behaves like a queue, but also has a moving window of data that always stores the previous source row of size Sdx.

Obviously, the larger the buffer on the composers, the better the Metabuffer will run. There will be fewer glitches that could occur due to composer starvation. However, because of the uniformity in the geometry of the data, it should be possible to use very small buffers. The main concern is to be able to buffer an entire burst of memory from the input framebuffer, which will be the quantum for the system. This will likely be limited by the pixel row size, though, since burst mode reads must be continuous. Also, note that because of pixel replication, at least one row of data may need to be archived for future computations. If advanced smoothing is being performed then multiple rows may be needed. This should be taken into account when determining the size of the buffer on the composers. Again, because of the geometry of the data, this value will remain constant no matter what the replication factor.

5.3 IRSA Round Robin Bus Scheduling

In order to send data to the composers in a simple, yet good performing manner, an idle recovery slot allocation (IRSA) round robin approach [9] is employed which distributes data to the composers evenly based on the amount of data needed (C), the period (T), and the deadline (D). No effort is made to look ahead in the geometry of the viewports to find the most efficient way to send the data out. However, because of the previous discussion, the uniformity of the data transmitted to each buffer will result in few delays using this simple method.

In the event that a composer-side buffer becomes too full to cope with the data, the round robin scheduler performs an idle slot recovery operation. The composer receiving data drops a bit defined as BUSREADY on the bus for one clock cycle. Once the input framebuffer reads the low BUSREADY bit, it stops sending data to that composer and jumps to the next scheduled segment in the table. This way other composers can utilize the unused time on the bus. The scope of the BUSREADY bit will be limited by the fanout of the bus, but this is true of the bus in general, and the low number of displays typically used should not cause a problem here.

5.4 Sequence of Metabuffer Operations

For each frame, the Metabuffer follows a sequence of steps in order to compute the final collaged output display. In order to synchronize themselves, the pipeline composers and output framebuffer employ a PIPEREADY bit to communicate with each other. The members of the pipeline use this PIPEREADY bit to inform the previous composer of their status. When this PIPEREADY bit is high, the previous composer knows that everything is clear on the rest of the pipe and, if it is ready as well, it can relay this information to the next higher member of the pipeline. Once the top of the pipeline is reached and everyone is ready, transmission of the next frame can begin.

The fact that each composer's internal buffer must be filled with the new frame's data before it can signal PIPEREADY ensures that all the composers will be synchronized on the same frame at the same time. Also, the pipelined nature of the PIPEREADY bit does not hinder the scalability of the design, because it is immune to fanout. The PIPEREADY bit will introduce latency since it must bubble up the pipeline from the output framebuffer. However, the number of cycles needed to traverse the pipeline is relatively small (number of rendering engines) compared to the cycles needed to compute the entire frame (millions of pixels), and so it shouldn't be an issue.

5.4.1 Frame Transition

Input frambuffers finish the previous frame, switch to next frame, and start feeding data to the composers.

5.4.2 Waiting for PIPEREADY

At this stage, composers have not received a PIPEREADY bit bubbling up from the composers in the pipeline below, but accept data until their internal buffers are entirely full without transmitting any data for this frame (though the previous frame could still be in computation) down the pipe.

5.4.3 Buffers Are Filled

When the internal buffers of the composers become full, each drops the BUSREADY bit on each transmission request from the input framebuffers, effectively stalling the Metabuffer.

5.4.4 Output Framebuffers Signal Completion

When the output framebuffers realize that they have finished building the old frame, they switch to a new frame and send a high PIPEREADY bit to the previous composer.

5.4.5 Composer Relays Finish Signal

When a composer gets a PIPEREADY bit from the following composer (or output framebuffer), it checks to see if its internal buffer is fully prefetched solely with the data from the new frame (all data from the old frame has been cleared out). If so, it relays the PIPEREADY bit to the previous composer in the pipeline. If not, it stalls until it is entirely prefetched.

5.4.6 Master Composer Signals Start of Frame

Once the PIPEREADY bit gets to the master composer (the composer at the top of the pipeline), and the master composer is ready, everything is set for that pipeline to begin computation of the next frame. The master composer starts the frame by sending a STARTFRAME bit down the pipeline and then streaming out data.

5.4.7 Composers in Pipe Begin Frame

The other composers in the pipeline, once they read the STARTFRAME bit, relay that bit down the pipeline and begin their computation. The STARTFRAME bit is important because it automatically establishes each composer's position on the pipeline (since each successive composer must be offset one cycle to be synchronized). Only the head composer at the top of the pipeline needs to be initialized via a PIPEMASTER bit set via a jumper when the circuitboard is installed.

5.4.8 Input Framebuffer Streams Out Data

Now that the pipeline is started and data is flowing, the input framebuffer will no longer get BUSREADY low bits, and can resume streaming data out to the computing composers in a round robin fashion.

Now that data is flowing through the busses and the pipeline, each composer, using an internal index of the output display, determines if the segment it is responsible for intersects the current coordinates. If so, it attempts to fetch the proper pixel information from the cache and compares it to the Z value of the previous pixel in the pipeline. Once an entire display has been sent to the output framebuffer, the process repeats itself.

6 Software Simulator

Because of the complexity of the Metabuffer and the fact that designing hardware is expensive and time consuming, a prototype of the Metabuffer has been built in software. This prototype is modeled as closely as possible to the operation of the Metabuffer architecture discussed previously in this paper. Since this software prototype will be the basis for the first hardware implementation of the Metabuffer, all coding was done strictly with the Metabuffer architecture in mind.

By building the prototype in software first, it is possible to do much more extensive testing and to try many more design alternatives in the same amount of time than with hardware. Changing a signal or reworking an algorithm means only recompiling the source code, instead of rewiring a circuit board or burning another FPGA. Also, with a software prototype, a Metabuffer consisting of hundreds or thousands of rendering engines can be simulated. Building a prototype Metabuffer of that size in hardware would require an enormous amount of resources.

Although the software prototype cannot operate in real time, it can be used to thoroughly simulate the operations of the Metabuffer. Except for speed, just about any other aspect of the design can be programmed and evaluated. New algorithms can be tested on the prototype just as if they were encoded into a DSP. Likewise, applications that use the Metabuffer can be tested at an early stage with the software prototype to solve design issues, taking into account that the final hardware version of the Metabuffer will offer more performance, while operating the same.

The Metabuffer software prototype was completed in C++, since the highly modular design concept lends itself to the use of object oriented programming. Each module (input framebuffer, composer, and output framebuffer) is defined as a separate C++ class. The data hiding capabilities of object oriented programming means that it is possible to create a large Metabuffer with possibly thousands of composers simply by replicating one class over and over again. Also, once the class is defined, changing the layout of the Metabuffer simply means adjusting the number of framebuffers and composers being used via the creation or deletion of class instances.

In addition, every instance of every C++ class in the prototype runs in a separate thread of execution using a standard pthread library. The threads are synchronized through the use of a barrier

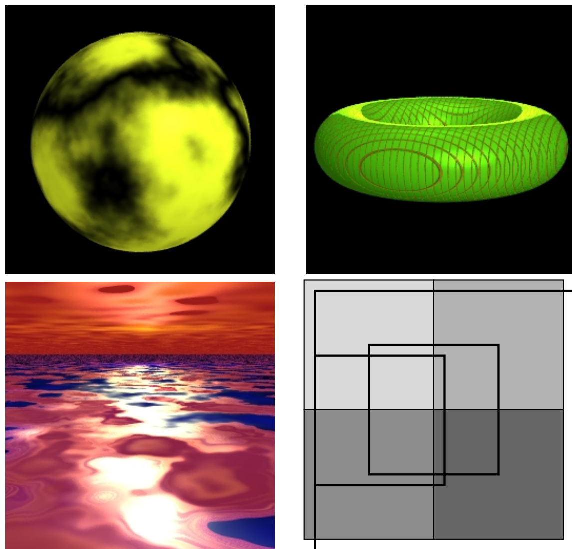
built with pthread synchronization primitives. Software threads closely simulate the parallelism advantages and synchronization problems that the Metabuffer has through the use of multiple hardware composers. Also, because of distributed computing packages built on top of the standard pthread API, the Metabuffer can be easily adapted to run on networked workstation clusters. Parallelizing the Metabuffer in this way would greatly increase simulation speed and also ease the I/O and computation requirements of using only a single host computer.

Finally, at every chance, old code in the software prototype is optimized with the future hardware in mind. For example, multiplies and divides are eliminated. Where possible, adds and subtracts are replaced with increments and decrements. Because part of the hardware design will actually consist of DSPs running natively coded software, it is likely that portions of the prototype may only have to be ported to the DSP instruction set. Optimizing this code now and fully testing it in the prototype means an easier transition to DSPs.

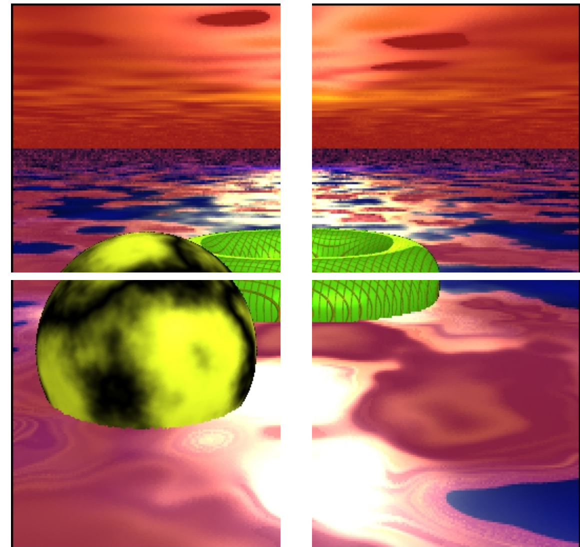
7 Experimental Results

In order to test the software prototype of the Metabuffer, it was necessary to obtain a source of rendered images and Z order values. Eventually this data will come from the digital output of COTS rendering engines. For now, though, images and Z order values were generated using the Rayshade ray tracer.

Although the standard distribution of Rayshade contains no way to output Z order information, a patch exists that allows for this information to be outputed in the form of a range map. Reading an image in TIF format and the Rayshade generated Z order information into the input frame buffer class simulates the transmission of a frame of RGB data and a frame of Z order data from the rendering engine.

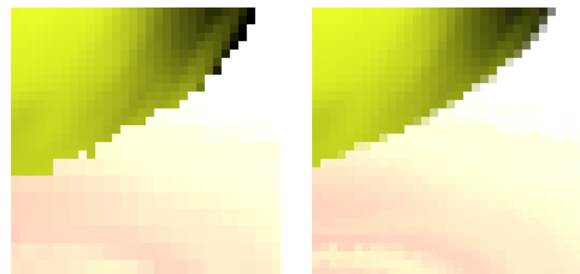


The images above show the TIF images that were rendered using Rayshade: a ball, a tube, and finally a seascape. The final diagram illustrates how these images were distributed to the four output displays by being broken up into viewports. Note that every image is sent to at least two output displays. As discussed in the bandwidth section earlier in this paper, the location and geometry of the viewports is arbitrary. The bandwidth requirements over the bus remains constant.



Running the three images into a three input framebuffer by four output frame buffer Metabuffer yields the four output screens shown above. Note that the tube resides in four separate displays, despite being rendered on a single machine. Also, see how the seascape here is being used as a low resolution background display with the higher resolution foreground images layered on top. Finally, the Z order of the input images is always taken into account, whether that means that the ball is in front of the tube, or that the ocean surface laps at the base of the foreground objects.

One problem with compositing separate images like the ones above is the aliasing that results on the edges. A solution that we have implemented involves supersampling. Simply increasing the detail of the input images and then having the output framebuffers average the pixel values down to the original size effectively smooths the image. Only the problem pixels at the edges are affected. The rest of the composited image pixels remain as sharp as on the original.



The two images above (magnified eight times to show the difference in detail) demonstrate the effect supersampling has on the resulting image quality. On the left, no supersampling has been performed. There is a jagged transition between the different input images at the Z buffer transition. On the right, the input images were rendered to be four times as detailed and the final output pixels were averaged by the output framebuffer from the four nearest pixels that traveled through the composer pipeline. The jagged transition is now much smoother while the rest of the image has lost no quality.

8 Conclusions

The Metabuffer provides for leveraging today's commodity PC technology to construct cost-effective, parallel high-end graphics

rendering systems with multidisplay capability. It has the advantages of easing load balancing by providing a uniform display space abstraction to the software, supporting multiresolution and foveated display, and providing a scalable platform with no changes to stock hardware. It does require the development of non-trivial custom hardware to perform image compositing. However, a parallel effort at Stanford University has been able to design hardware that can support a version of this type of image compositing [7]. Fortunately, most of this work can be done without resorting to custom VLSI, at least for prototypes.

We can also hope to avoid the fate of so many parallel architecture projects in the past, in which the development of custom switching hardware took so long that the advantages of parallel computation were swamped by the rapid development of commodity semiconductor technology. This is not only through avoiding using custom silicon, but also because our hardware must be designed to handle video standards, which change more slowly than processor and system clock speeds. Our system will be usable with many future generations of processors, even with a slower development cycle.

Essentially our hardware implements a pipelined switch connecting a number of broadcast channels. Each of the latter could be implemented as a high-speed network, and thus we can consider the Metabuffer hardware as a pipelined network switch capable of performing a limited set of transformations on the data passing through it. In essence, then, this is a high-speed active network switch. It will be interesting to see what other kinds of applications besides rendering can be implemented with similar hardware, and to actually develop a version of the Metabuffer that works as a network switch. This should allow us to provide a parallel system with a set of display resources which can be dynamically allocated not just to different portions of a multidisplay image but also to distinct physical displays located in various places on a LAN, in the spirit of [8].

References

[1] C. Bajaj, I. Ihm, G. B. Koe, and S. Park. Parallel raycasting of visible human on distributed memory architectures. In *Proceedings of Joint Eurographics-IEEE TCVG Symposium on Visualization*, pages 269–276, May 1999.

[2] C. Bajaj, V. Pascucci, D. Thompson, and X. Y. Zhang. Parallel accelerated isocontouring for out-of-core visualization. In *Proceedings of IEEE Parallel Visualization and Graphics Symposium*, pages 97–104, October 1999.

[3] M. Bunker and R. Economy. Evolution of ge cig systems. *SCSD Document*, 1989.

[4] D. J. Sandin C. Cruz-Neira and T. A. DeFanti. Virtual reality: The design and implementation of the cave. *Computer Graphics*, 27(4):135–142, August 1993.

[5] M. Green C. D. Shaw and J. Schaeffer. A vlsi architecture for image composition. In *Proceedings of the 1988 Eurographics Workshop on Graphics Hardware*, pages 183–199. Eurographics Seminars, 1988.

[6] D. S. Fussell and B. D. Rathi. A vlsi-oriented architecture for real-time raster display of shaded polygons. In *Graphics Interface '82*, May 1982.

[7] Pat Hanrahan. Scalable graphics using commodity graphics systems. Views pi meeting, Stanford Computer Graphics Laboratory, Stanford University, May 17, 2000.

[8] A. Heirich and L. Moll. Scalable distributed visualization using off-the-shelf components. In James Ahrens, Alan Chalmers, and Han-Wei Shen, editors, *Parallel Visualization and Graphics Symposium – 1999*, San Francisco, California, October 1999.

[9] J. P. Lehoczky K. A. Kettler and J. K. Strosnider. Modeling bus scheduling policies for real-time systems. In *Proceedings of 16th IEEE Real-Time System Symposium*, pages 242–253. IEEE Computer Society Press, 1995.

[10] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, 1994.

[11] S. E. Molnar. Combining z-buffer engines for higher-speed rendering. In *Proceedings of the 1988 Eurographics Workshop on Graphics Hardware*, pages 171–182. Eurographics Seminars, 1988.

[12] S. E. Molnar. Image composition architectures for real-time image generation. Phd dissertation, technical report tr91-046, University of North Carolina, 1991.

[13] Rudro Samanta, Jiannan Zheng, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Load balancing for multi-projector rendering systems. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, August 1999.

[14] R. Weinberg. Parallel processing image synthesis and antialiasing. *Computer Graphics*, 15(3):55–61, July 1981.

[15] X. Y. Zhang, W. Blanke, C. Bajaj, and D. Fussell. Loading balancing on graphics pc clusters for multi-tiled displays. Technical report, University of Texas at Austin, 2000.