PART II

A PROTOTYPE SYSTEM DESIGN

CHAPTER 6

PIXELFLOW: A PROTOTYPE IMAGE-COMPOSITION SYSTEM

Part II (Chapters 6 - 9) describe the design of *PixelFlow*, a prototype image-composition system, to demonstrate the claims made in the thesis statement. Although the system has not been implemented yet, the design presented in these chapters is intended to illuminate its main implementation and performance issues.

The PixelFlow design has the following characteristics:

- **Flexible.** It can display many primitive types using advanced rendering algorithms.
- **Scalable.** Its performance scales linearly with the number of renderers.
- **Achieves higher performance than existing systems.** The PixelFlow design described here is expected to render 2.5 million Gouraud-shaded triangles in a two-card–cage system and be expandable to much higher performance—well above that of any existing or published system known to the author.

The PixelFlow architecture is a hybrid, having characteristics of both the supersampling-style and A-buffer–style architectures described in Part I. PixelFlow renderers use i860[1] microprocessors for geometry calculations and Pixel-Planes-style logic-enhanced memory chips for rasterization. A rectangular region of the screen is rasterized as a unit, as in a parallel *z*-buffer system; multiple regions are rasterized sequentially, as in a scan-line system. After each region is rasterized, pixels from the multiple renderers are scanned out and composited over an extremely high-bandwidth image-composition network. Antialiasing is performed by supersampling.

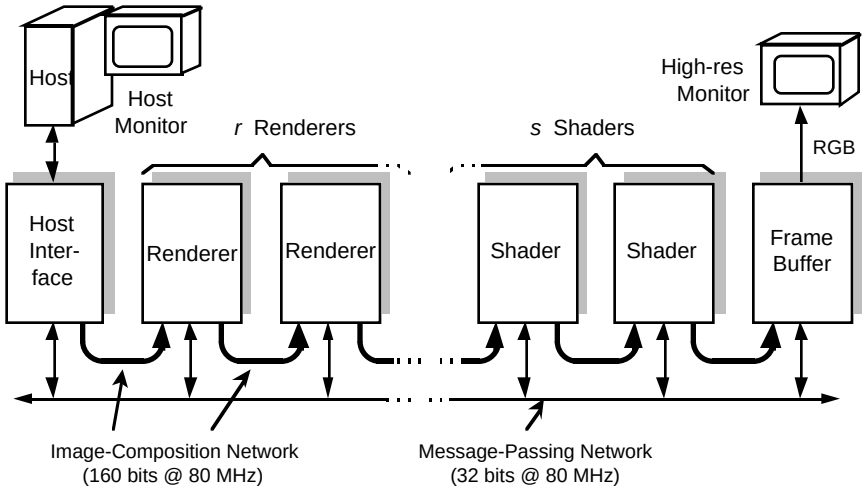


Figure 6.1: Block diagram of the prototype PixelFlow system.

PixelFlow systems can be implemented with varying levels of sophistication. We will describe a relatively unsophisticated system here, which uses only proven technology and components, to establish the claims of the thesis statement. The system we plan to build is more sophisticated, having a higher-speed image-

---

[1] Intel and i860 are trademarks of Intel Corporation.

composition network and more capable enhanced-memory chips. It requires an aggressive memory-chip design that has not yet been completed, and, therefore, is not suitable for our proof-of-concept design. We will briefly describe our plans for this enhanced system in Chapter 10.

Figure 6.1 shows a block diagram of the prototype PixelFlow system. The system is modular and can be configured with an arbitrary number of renderers. Each can display approximately 70,000 *z*-buffered, Gouraud-shaded, 100-pixel triangles per second. A two-card–cage system containing 36 renderers, therefore, can display approximately 2.5 million Gouraud-shaded triangles per second. A subset of the renderers can be configured as shaders for Phong shading, volume rendering, antialiasing, or more sophisticated shading algorithms.

This chapter describes the design objectives and major architectural decisions that led to the prototype-system design, and gives an overview of its components. Chapters 7 and 8 describe the image-composition network and renderer/shader board in detail. Chapter 9 describes the system's synchronization and control algorithms and presents simulation results.

6.1 DESIGN OBJECTIVES

The aim was to design a proof-of-concept image composition system that establishes the claims in the thesis statement and is interesting in its own right. We believed that a system containing 100,000 polygon-per-second renderers could reduce the cost of image composition to an acceptably small fraction of the overall system cost, and that this could lead to an interesting, realizable system. The following are the five major design objectives for the prototype system:

- **Scalability to extremely high performance.** The system must demonstrate the scalability property of image-composition architectures.
- **Support for advanced primitives and shading models.** The system should be flexible enough so that the same system resources can be used to render simple scenes rapidly, or to render sophisticated scenes with curved surfaces, volume data, Phong shading, multiple light sources, high-quality antialiasing, or texturing as rapidly as possible. Applications such as computer-aided modeling and scientific visualization must be able to trade image quality for rendering time.
- **High frame rate and low latency.** This is crucial for real-time applications, such as flight simulators and head-mounted displays. The system should sustain frame rates of at least 24-30 Hz. for antialiased images and have latency as close to one frame time as possible, with an upper bound of two frame times.
- **High resolution, antialiased display.** The system should support real-time frame rates for antialiased images at 1280x1024-pixel resolution (the current high-resolution–display standard).
- **Economically and technically feasible.** The prototype system must be realizable, both economically and technically. It must achieve its performance with a reasonable amount of hardware. (A rough target for a two-card–cage system is three million triangles per second). The system must use proven components and technology, making use of off-the-shelf or existing parts when available, and to avoid exotic or expensive parts and technologies.

The next section describes major architectural features of the design in an order that roughly parallels their evolution.

6.2 ARCHITECTURAL DECISIONS

This section describes the main architectural decisions that led to the PixelFlow design and their motivation.

6.2.1 *Z*-buffer Rendering with Supersampling Antialiasing

The crucial system parameter for image-composition systems is bandwidth in the image-composition network. Initially, we believed that supersampling requires too much bandwidth to be practical in a

moderate-cost system. Two data words (color and *z*) are required for each subpixel sample, and conventional wisdom says that 8 to 16 samples are needed for adequate sampling. As a result, we focused our early efforts on scan-line–based A-buffer systems. Two realizations led us to reconsider the supersampling approach:

1) A-buffer compositing requires four data words per fragment and two to three fragments per pixel in complex images—a total of 8-10 words per pixel (see Section 4.3.2).

2) Supersampling can produce reasonable-quality images with as few as 5 samples per pixel, if sample locations and weights are chosen carefully [MOLN91].

Since supersampling requires only two words per sample, five samples per pixel require no more bandwidth than the two to three fragments per pixel in an A-buffer system. Supersampling also avoids artifacts that arise from sampling interpolated parameters only once per pixel. Further, the compositor in a *z*-buffer system is much simpler, so it can be smaller and run faster than an A-buffer compositor. This renewed our interest in *z*-buffer, supersampling systems. The missing link was a high-performance, compact *z*-buffer rendering engine.

6.2.2 Pixel-Planes 5-Style Renderers

Our group's previous graphics-system design was Pixel-Planes 5, a high-performance system that renders more than two million polygons per second using a "smart memory" approach. The central idea is to render an entire primitive in parallel on a 128x128-pixel rasterizer array. The rasterizer contains a processor for each pixel and computes all the pixels in parallel. The system computes an entire image by assigning a rasterizer to each region of the screen and processing all of the primitives that fall into that region [FUCH89].

One of the current design efforts in our laboratory is a development by IVEX Corporation of a one-board graphics engine for polygon rendering in a flight simulator. A feasibility study for that project was to measure the rendering performance of a system composed of one Pixel-Planes 5 Graphics Processor Board and one Renderer Board. Trey Greer, of IVEX Corp., implemented software to do this and determined that a Graphics Processor/Renderer pair can render approximately 50,000 antialiased triangles per second and 100,000 unantialiased triangles per second [GREE91].

A one-board renderer of this type satisfied our need for high rendering performance in a compact package. We considered other renderer implementations, such as an array of multiple i860s and a 4x4-pixel footprint processor, but found the Pixel-Planes 5 design to be the simplest and most economical. Also, since the new design would have much in common with Pixel-Planes 5, we could use existing designs and components. The serial ports of the enhanced-memory chips (EMCs) led to a natural, high-bandwidth interface to the image-composition network: a renderer containing 64–80 EMCs has 256–320 serial communication wires that can transmit pixel data at 20 MHz.

After some analysis, we realized that the 20 MHz EMC serial ports did not provide the composition-network bandwidth we needed. John Poulton suggested that the chips could be modified in a straightforward way to double the speed of the serial port to 40 MHz. This reimplementation of the EMC could use a newer CMOS technology with smaller feature size (1.2μ vs. 1.6μ), also allowing us to increase the amount of memory on the chip.

We decided that a renderer board with 80 improved EMCs (covering a 160x128-pixel region) provided a good balance between image-composition–network bandwidth, processor utilization, and bin-replication overhead. The EMCs' 320 serial-port wires would provide enough bandwidth to render high-resolution images with five samples per pixel at almost 30 Hz.

6.2.3 Wide, Slow Image Composition Network

We decided to implement the image-composition network as a pipeline, rather than a tree, to make the hardware easier to build (see Section 5.1). Each renderer board would have its own compositor to merge its pixel data onto the composition network. Presented with 320 40 MHz wires from each renderer, we had to choose an appropriate implementation for the image-composition network. The two main choices were:

- **Fast and narrow.** We could multiplex the signals to use fewer, high-speed wires. This would require level conversion to a higher-speed technology such as ECL, as well as multiplexing circuitry.
- **Wide and slow.** We could leave the network wide and use conventional TTL or CMOS technology for compositing and transmitting pixel data between boards, though many wires and connectors would be required.

Initially, we considered a fast, narrow image-composition network. ECL circuitry can run at speeds of up to several hundred MHz. This would minimize the number of wires connecting boards. Although we have had positive experiences with the 160 MHz 32-bit ring network in Pixel-Planes 5, it requires disciplined design, and the problems increase as the clock rate increases. We would also need to multiplex the parallel data paths on each board. This would be expensive in terms of parts and board area, particularly considering the cost of TTL/ECL level conversion—one part for every six signals.

The technical problems seemed more difficult with a very fast network, and the savings in parts were doubtful. The format of pixels transmitted from EMCs led to a simplification that tipped the balance in favor of a wide network. Each EMC has two 2-bit ports, over which pixels are transmitted in two-bit-serial fashion at 40 MHz. If the image-composition network runs at 80 MHz, the two-bit data from the EMCs can be multiplexed onto a single wire; pixel data could then be transmitted between compositor chips bit-serially.

Compositing requires comparing *z*-values of corresponding pixels. The serial nature of the EMC serial port led to the notion of serializing the composition calculations, allowing them to be performed in simple, fast programmable logic devices (PLDs). Gazelle Microcircuits manufactures PLDs with propagation delays as low as 5 nsec. (see Section 7.1.1). Compositor PLDs could be positioned near the board edge, minimizing the wire length between parts. All of these factors meant that we could design the image-composition network to run at 80 MHz with 160 wires passing from board to board, a number readily supportable using standard connector technology.

6.2.4 Support for Deferred Shading

As described in Section 5.5.1, many shading calculations depend only on generic quantities, such as surface color and surface-normal vectors. These calculations can be performed only once per pixel, rather than once per primitive, if we use deferred shading (see Section 5.5.1).

To implement deferred shading efficiently in an image-composition system, a processor is required that can accept rasterized pixels from the image-composition network at full speed, shade the pixels, and forward them on to the frame buffer. The Pixel-Planes EMCs on the renderer boards are ideal processors for deferred shading, but the data is in the wrong place at the wrong time. We can solve this by designating a subset of the renderer boards as *shaders*.

In a system with shaders, renderers send pixel attributes such as intrinsic colors, surface-normal vectors, and *z* values over the image-composition network to the shaders, instead of RGB values. The shaders perform shading calculations and forward the shaded pixels to the frame buffer. They also blend pixel samples for supersampling antialiasing.

The compositors require additional operating modes to allow renderers to be used as shaders. They must be able to load pixels into the EMCs, unload them without compositing, and forward pixels without modification. These modes can easily be incorporated into the compositor PLDs (see Section 7.1.1).

6.2.5 Buffering for Multiple Regions

As described in Section 5.4, load-balancing is an important concern for images with an uneven distribution of primitives over the screen, or if the database distribution algorithm is imperfect (this is especially important for immediate-mode applications). An entire frame of buffering between the renderers and image-composition network is needed to solve this problem completely. If regions are processed in scrambled order, less buffering is required for non-pathological scenes, since successive regions come from different areas of the screen, and are likely to contain uncorrelated numbers of primitives.

Buffer chips can be added between the EMCs and compositor chips, or buffering can be incorporated into the EMCs themselves. External buffer memories must sustain the image-composition network's full band-

width at their input and output ports. This would require 80 buffer chips with double, bi-directional, 4-bit 40 MHz ports. This is possible using triple-ported VRAMs, but is expensive in dollars, board area, and in added control complexity.

The alternative, buffering several regions of pixel data on the EMCs, is not feasible with the 6-transistor static memory cells used in the Pixel-Planes 5 EMCs, but could conceivably be done if 4-transistor or 1-transistor dynamic-memory cells were used. Sample layouts of each type of memory cell indicate that 256 bits per pixel are possible with the 6-T design, 512 with the 4-T design, and 1024 with the 1-T design (all of these assume 256 pixels per EMC) [POUL91]. A 1-T memory system would require an extensive redesign of the EMC. A 4-T design provides sufficient buffering for four screen regions and can be incorporated into the framework of the Pixel-Planes 5 EMC.

Simulation shows that four-region buffering reduces the load-balancing problem by up to a factor of two for our test images (see Section 9.2). This is still far from the factor of ten or so that is possible, given the dynamic-load–balance ratios of these datasets. Eight or more regions of buffering would be preferable, but this would significantly raise the cost of the EMC. Four regions of buffering seems to be a reasonable compromise.

6.3 PIXELFLOW SYSTEM OVERVIEW

We now give a brief overview of the prototype PixelFlow system. We will describe each component at high level in this section. Crucial components, such as the image-composition network, the renderer/shader board, and the software architecture are described in more detail in Chapters 7 – 9.

A PixelFlow system is composed of one or more 19-inch card cages containing up to 21 circuit boards each and connected to a DECStation 5000, or other high-speed workstation. The boards in each card cage are attached to a common backplane, which contains wiring for a message-passing network and an image-composition network that extends to each system board.

The prototype PixelFlow system contains three main board types:

- *Renderer/shaders*, which are one-board graphics computers capable of rendering 70,000 *z*-buffered triangles per second or computing shading models for 160x128 pixels in parallel.
- A *host interface*, which implements the connection to the host computer and contains synchronization and diagnostic hardware for the image-composition network
- A *frame buffer*, which buffers and displays composited pixels.

Multiple card cages can be connected by placing them side-by-side and connecting their backplanes together with special *bridge boards* and connectors. *End boards* are needed to turn signals around at the left and right ends of the outermost backplanes. Figure 6.2 shows a logical diagram of a two-card–cage PixelFlow system.

The *message-passing network* is a general-purpose communication network that allows any board to send messages to any other board. It is a scaled-down version of the multi-channel token ring used in Pixel-Planes 5. It contains four point-to-point communication channels, each of which can transmit data between any pair of boards at 80 Mbyte/sec. The network is used for loading code and data during system initialization and for updating viewing parameters and editing the graphics database during interactive operation.

The *image-composition network* is a very wide (160-bit), high-speed (80 MHz) special-purpose communication network for rapidly moving pixel data between adjacent boards. It performs two functions in different parts of the system: it transfers pixel data between compositors in the renderers and transfers rendered pixels from the shaders to the frame buffer. The image-composition network is implemented as 160 wires that connect adjacent backplane slots. Compositor chips on the renderer and shader boards synchronously transmit data to compositors on adjacent boards.

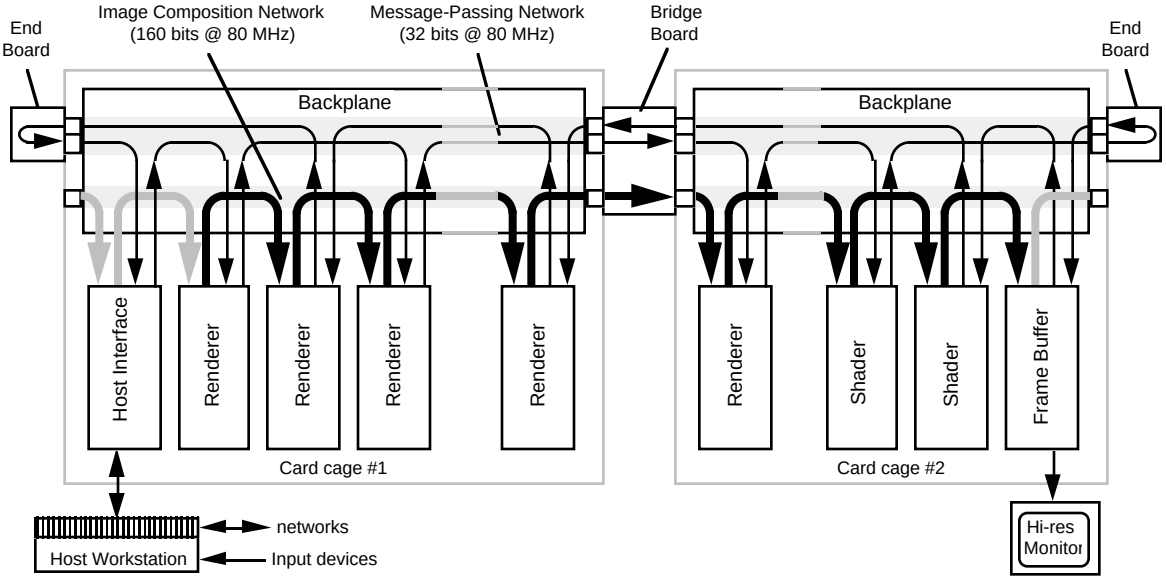


Figure 6.2: Two-card–cage PixelFlow system.

We now briefly describe the various components of the system. The image-composition network and the renderer/shader board will be discussed in more detail in Chapters 7 and 8.

6.3.1 Host and Host Interface

The host computer is a DECStation 5000[1], or some other general-purpose computer, that serves as the system master. It loads code and data onto the various system boards, distributes the display database over the rasterizers, runs the application, samples user inputs, and determines the changes to be made during each frame.

The host interface connects the host to the system backplane. In addition, it contains diagnostic and synchronization circuitry for the image-composition network. Figure 6.3 shows a block diagram of the host-interface board.

The *pixel buffer* is composed of 10 banks of VRAM memory that can load data onto the image-composition network at full speed. It provides a way for the host computer to send diagnostic data or background images to other boards in the system via the image-composition network. The corner turners and output multiplexer PLDs are similar to the corner turners and input demultiplexer PLDs used in the frame buffer (see Section 6.3.5).

The system is designed to support most applications using a retained-mode data structure. In a typical application, the host computer first initializes the system, configuring the different boards appropriately, and booting the processors on the renderer/shader boards. Next, it reads the database off disk (or wherever it resides) and distributes it and loads it onto the renderers. For load-balancing, the host distributes primitives over the renderers using scattering. It must keep track of where each primitive resides in the system, so it can update it or delete it when necessary (it can use the same general approach as is done in Pixel-Planes 5 [ELLS90a]).

---

[1] DECStation is a trademark of the Digital Equipment Corporation.

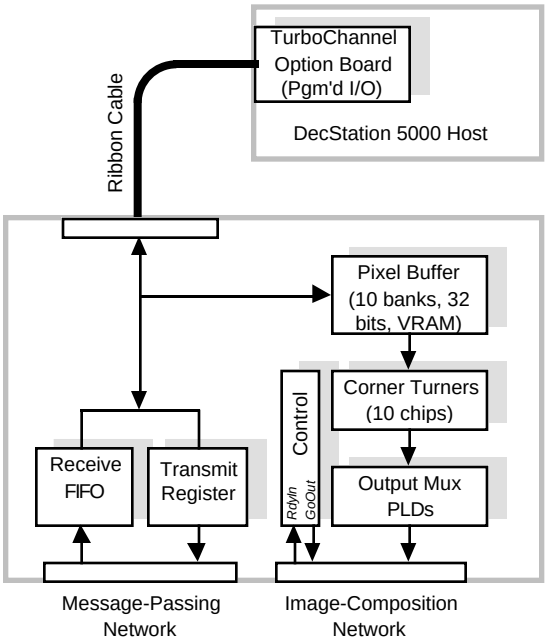


Figure 6.3: Block diagram of host interface.

After the database has been loaded and distributed, the host typically executes the loop shown in Figure 6.4 (various optimizations to this loop are possible). The host broadcasts editing commands over the message-passing network. It can keep up with the rest of the system as long as the editing commands are few and easy to calculate.

```
repeat {
     sample user inputs;
     send editing commands;
     send render command to the Renderers;
     wait for finished handshake from the Renderers;
}
```

Figure 6.4: Pseudocode for host algorithm.

Other application models for the system are possible. For example, the message-passing network and processors on the renderer boards represent a general, message-passing multicomputer. An application could be parallelized to run on the renderers and to generate primitives in immediate mode. The renderers, in this case, would process these primitives directly.

6.3.2 Message-Passing Network

The message-passing communication network is a four-channel token ring that supports point-to-point communication between any pair of boards. It is a scaled-down implementation of the multi-channel token ring used in Pixel-Planes 5 [GREE89].

The ring is implemented as 32 data wires that transmit data at 80 MHz. The four channels are time-division multiplexed onto the data wires, such that data for a single channel is sent on every fourth word. A message from any sender to any receiver can take place over any of the four channels. The ring network has one node for each board in the system. Each node has a unique *id*, which is used to address messages destined for that node.

The ring transmit interface is fairly simple: the sender formulates a message and prepends a destination address containing the receiver id and other user-defined address information. It asserts this address on its

transmit port and asserts the *TxReady* signal. The ring then attempts to acquire the receiver. Receivers are treated as resources that can be assigned to senders; each resource has a corresponding bit in a circulating control field. Once the receiver is acquired, the ring waits for a free channel. Once the receiver and a channel have been acquired, the ring interface asserts the *TxGo* signal. This tells the sender that it can begin clocking message data out the transmit port at 20 MHz. The sender terminates the communication by deasserting the *TxReady* signal.

The receive interface works as follows: The receiver board asserts the *RxReady* signal when it is ready to accept data (from any sender). The ring interface asserts *RxPut* when data arrives at the receive port. The receive board may deassert *RxReady* at its discretion when it begins receiving data. The ring will not allow another message to arrive until *RxReady* is asserted again. The receiver board is responsible for buffering incoming data. This usually means that it must have a first-in-first-out buffer (FIFO) of some sort, which can limit the length of messages that can be sent.

Figure 6.5 shows a block diagram of the ring-node interface.

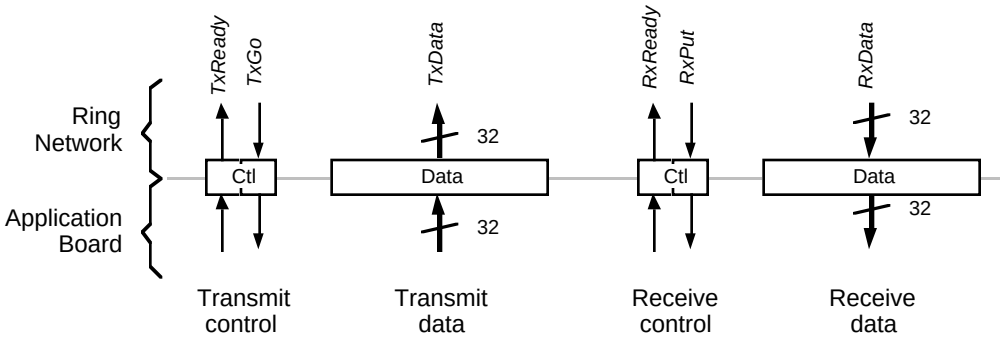


Figure 6.5: Ring node transmit and receive interface.

The communication ring in Pixel-Planes 5 has eight 20-Mword/sec channels, in contrast to the four channels in PixelFlow. However, the Pixel-Planes 5 ring must convey all of the primitives in the database between Graphics Processors and Renderers every frame, as well as all of the pixels to the frame buffer. PixelFlow has none of this high-bandwidth traffic, so its bandwidth requirements are less. A four-channel ring can be implemented with TTL logic and PLDs running at 80 MHz. (This clock rate is possible with TTL because the wiring channels between boards are short.) As in Pixel-Planes 5, the ring is collapsed onto the backplane to avoid having to "close" the ring with a long connection (see Figure 6.6).

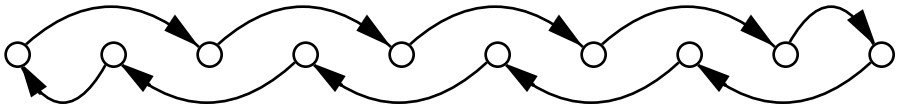


Figure 6.6: Ring flattened on backplane to reduce wire lengths.

6.3.3 Renderer/Shaders

The renderer/shader boards are the workhorses of the system. Renderers store and rasterize a partition of the database. Shaders shade and antialias composited pixels. Although renderers and shaders perform different functions, the same board design is used for each. The board is configured as a renderer or a shader by the software that is loaded onto it. We first describe the components on a renderer/shader board, and then describe the tasks performed in each configuration.

Renderer/Shader-Board Components. The renderer/shader board is composed of three main parts: a *graphics processor*, a *rasterizer*, and a *compositor*. The graphics processor (GP) is a fast floating-point

processor based on the Intel i860 microprocessor. It is patterned after the Pixel-Planes 5 Graphics Processor Board [MOLN89]. It has 8 Mbytes of local VRAM memory, a transmit and receive interface to the message-passing network, and a FIFO interface to the rasterizer.

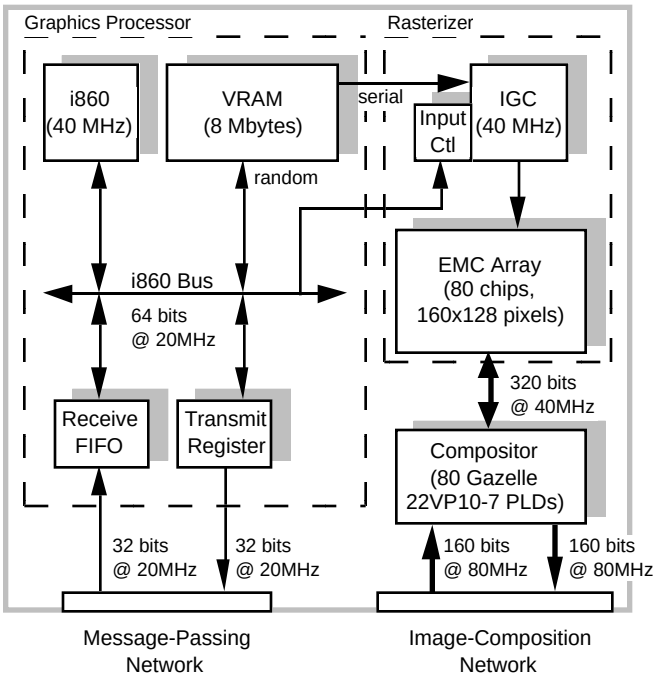


Figure 6.7: Block diagram of renderer/shader board.

The rasterizer is a massively parallel rasterization processor, patterned after the Pixel-Planes 5 Renderer Board [EYLE90]. It contains an *Image Generation Controller* (IGC) and an array of 80 modified *Enhanced Memory Chips* (EMCs). The compositor contains 80 compositor chips with associated control circuitry. Figure 6.7 shows a block diagram of a renderer/shader board.

The GP handles communication over the message-passing network, stores data, and sends commands to the IGC, which controls the EMC array. The IGC is equipped with two input FIFOs, one for storing rendering commands (shading commands in the case of a shader), the other for storing compositing commands. The IGC converts word-parallel commands and coefficients into bit-serial commands required by the EMCs.

Each EMC contains 256 pixels. Each pixel contains 512 bits of local memory, a 1-bit ALU, a leaf node of the linear-expression tree, and a variable-length (64 or 128-bit) transfer buffer. The transfer buffer in each EMC is connected to a 4-bit, 40 MHz, bidirectional serial data port, which is wired to a compositor chip. The EMCs rasterize and shade using algorithms developed for Pixel-Planes 5 [FUCH89].

Compositor chips merge the pixel stream from the current renderer with the pixel stream from the upstream renderers. They also load and unload pixels onto the shaders, and transfer pixels to the frame buffer for display.

Renderers. On a renderer board, the GP stores its portion of the database in its VRAM memory. At the beginning of each new frame, the GP receives editing commands and a new viewing transformation from the host. At the start of the frame, the GP processes these commands and sorts primitives into screen regions. Then, region-by-region, the GP transforms primitives into screen coordinates and loads them into the IGC FIFOs.

The IGC and EMC array execute these commands, rasterizing the primitives that fall into each region. When a region is finished, it is copied into the transfer buffer in the EMCs, and composited with the corresponding region on each of the rasterizers. The composited pixel data from the last renderer can either be loaded directly into the frame buffer, or loaded into a shader, for antialiasing or advanced shading

calculations.

Shaders. In a shader, the EMC array does almost all of the work. The GP's only function is to load shading instructions into the IGC FIFOs. The shader receives rasterized regions, one at a time, over the composition network into the transfer buffer of its EMC array. It copies this pixel data into another portion of pixel memory and then performs shading calculations on it. When the region has been completely shaded, color data is copied back into the transfer buffer, and sent out to the frame buffer for display.

To keep the image-composition network busy, the shader's operation must be pipelined. It does this by shading one region while the next region is being loaded into its transfer buffer. When the new region is loaded, it is shaded while the previous region is transferred out.

6.3.4 Image-Composition Network

The image-composition network merges pixel streams from each renderer into a single pixel stream, loads and unloads the shaders, and transfers shaded pixels to the frame buffer for display. It is composed of the compositor chips on each renderer/shader board, their associated control circuitry, and the interconnect wiring on the system backplane.

Compositor chips are paired one-to-one with EMC chips on renderer/shader boards, a total of 80 per board. Each compositor chip has a 4-bit 40 MHz bidirectional port connecting it to its associated EMC, and two 2-bit, serial ports connecting it to compositor chips on adjacent boards. The input port connects to the renderer/shader upstream. The output port connects to the renderer/shader (or frame buffer) downstream. The input and output ports are each 2 bits wide and run at 80 MHz. Figure 6.8 shows the connections for a single compositor chip.

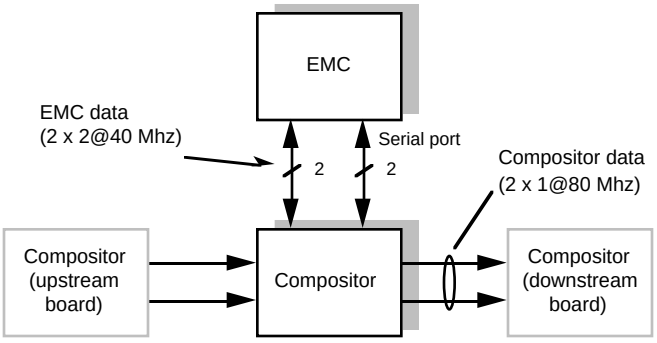


Figure 6.8: Connections to a single compositor chip.

Because the communication port from each EMC is much narrower than the data for one pixel, pixels are transferred serially. Due to layout considerations in the EMC, the EMC's serial port presents 2-bit slices of each of two pixels over its serial port on each 40-MHz clock tick. This 2-bit-wide data is converted by the compositor chips into bit-serial data at twice the clock frequency—80 MHz. The entire image-composition network operates on bit-serial data. This makes comparison of *z* values exceedingly simple: *z* values are sent down most-significant-bit first. The bits are compared one at a time. The first bit that differs determines which pixel is in front. This sets state bits in the compositor chip that forward the remainder of the nearer pixel's data to the output bitstream. Figure 6.9 diagrams the bit-serial compositing process for one pixel.

The compositors on the EMCs implement several operations in addition to the composite operation described above. They allow pixels to be loaded into the EMC array, unloaded without compositing, or simply forwarded on without modification. These extra modes are needed when the board functions as a shader (these modes are described in detail in Section 7.1.1).

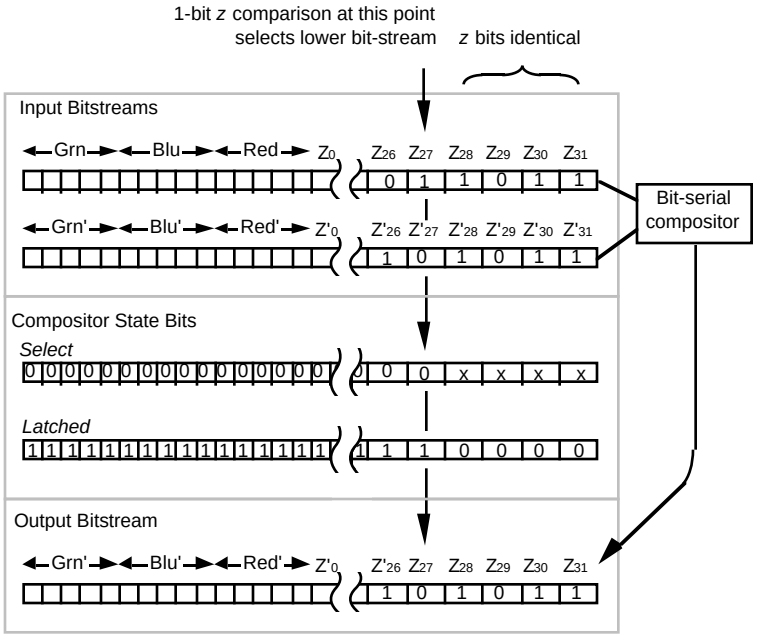


Figure 6.9: Bit-serial composition of two pixel streams.

The image-composition network operates by transferring a 160x128 region of pixels at a time. Synchronization circuitry on each board assures that each renderer is ready prior to beginning a transfer, and sequences the transfers on each of the boards. Each transfer runs uninterrupted until a full 160x128 region has been transferred.

Since each compositor chip has a communication bandwidth of 40 MHz • 4 bits = 160 Mbit/second to/from its associated EMC, the total bandwidth between the rasterizer and compositor on each renderer/shader board is 80 • 160 Mbit/second = 1600 Mbyte/second. This bandwidth determines the system's maximum update rate, screen resolution, and number of samples per pixel. Since these properties can be traded off against each other, the network bandwidth can be used in a variety of ways. Typical examples are:

- 1280x1024 screen, Gouraud-shaded polygons, 5 samples per pixel, 27 frames per second.
- 1280x1024 screen, Phong-shaded polygons, 1 sample per pixel, 55 frames per second.
- 640x512 screen, Phong-shaded polygons, 5 samples per pixel, 53 frames per second.

6.3.5 Frame Buffer

The frame-buffer board implements a conventional 1280x1024, double-buffered, 24-bit frame-buffer. Its input port accepts pixel data arriving over the image-composition network. It, therefore, must support the composition-network bandwidth and convert the bit-serial pixel streams arriving over the network into word-parallel pixel data needed for the frame buffer. Because the frame buffer interacts with the image-composition network in real time, it requires a fairly sophisticated controller. It is provided with an i860, which serves as controller and can also read and write data to the frame buffer, as necessary. Figure 6.10 shows a block diagram of the frame-buffer board.

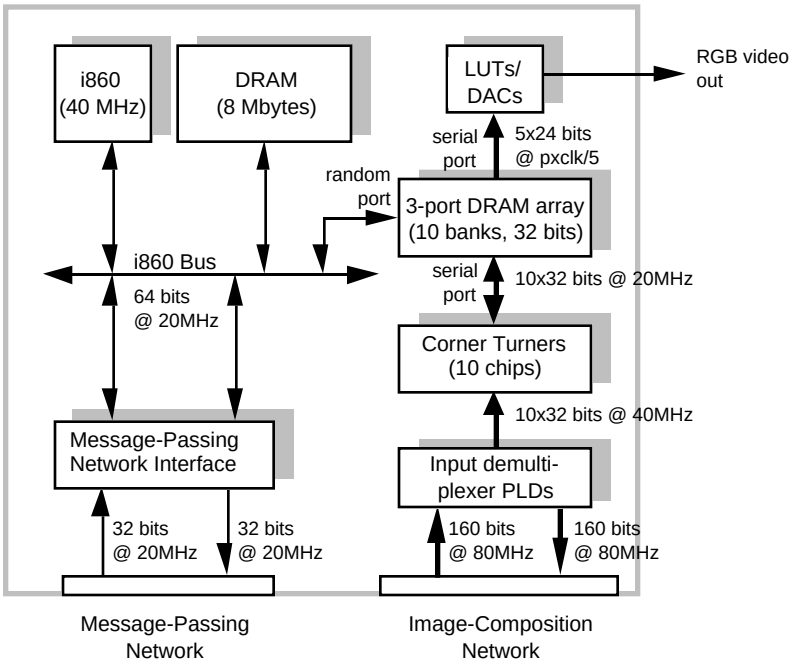


Figure 6.10: Block diagram of a frame-buffer board.

The frame buffer contains ten 32-bit banks of triple-port 1 Mbit DRAM chips (these DRAMs, such as the Micron MT43C4258, are like VRAMs, except that they have two high-speed serial ports [MICR91]). This is exactly enough memory to support a double-buffered 1280x1024 screen.

The crucial part of the frame-buffer design is the interface between the image-composition network and the DRAM. The frame buffer must be able to accept pixel data from the image-composition network at full speed. This is done in two stages: an array of high-speed demultiplexer PLDs, and an array of corner-turner chips.

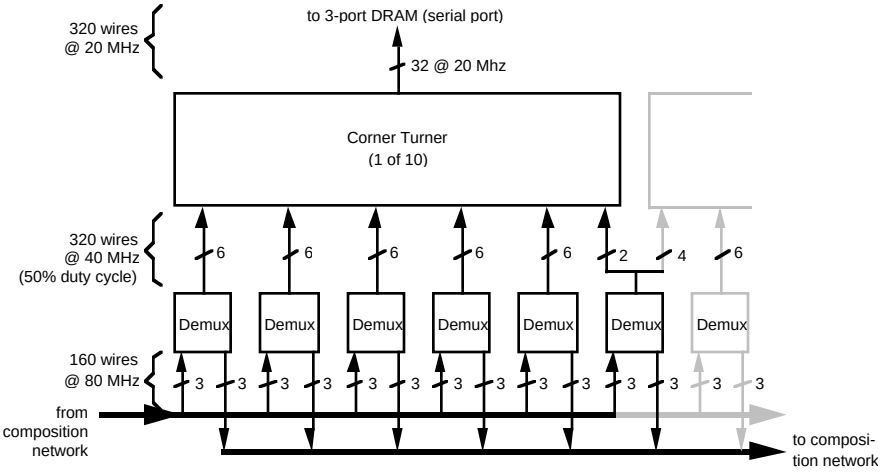


Figure 6.11: Demultiplexer PLDs and corner turners.

The incoming pixel stream is directed to an array of 54 high-speed 22V10 PLDs, similar to the compositors on a renderer/shader board. These PLDs demultiplex the 160 wires running at 80 MHz to 320 wires running at 40 MHz (see Figure 6.11). In addition, they forward the original pixel stream back onto the

image-composition network so that more than one frame-buffer board can be included in a system.

The second stage consists of ten corner turners. These are programmable gate arrays, which receive 2-bit serial pixel data from the input demultiplexers and convert it into word-parallel format compatible with the DRAMs.[1] They also divide the data rate by two, since the input pixel stream contains *z* information, as well as color information, and the *z* information is spurious at this point. The output of each corner turner mates with one of the serial ports of one bank of DRAM. The serial port accepts pixels from the corner turner at 20 MHz.

The DRAMs' second serial ports are connected to 5-way multiplexing video look-up-tables and digital-to-analog converters in the conventional way [FOLE90].

The i860 has access to the DRAMs' random port. It can read and write pixels, allowing it to read the contents of the frame buffer or write data to it (useful for text, windows, and debugging). The frame-buffer memory controller occasionally takes over the DRAMs' random port to perform row loads for either of the two serial ports.

6.3.6 Algorithms and Performance

The PixelFlow architecture supports a wide variety of rendering algorithms and primitive types, from Gouraud-shaded polygons, to Phong-shaded volume data, to directly rendered Constructive-Solid-Geometry objects. It supports immediate-mode and retained-mode traversal methods. It is scalable over a wide variety of performance levels. One of the main motivations for developing the architecture was to support experiments with new rendering algorithms.

The system was designed with high-quality polygon rendering as its primary application. Polygon rendering can be thought of as a least-common-denominator for other algorithms. Figure 6.12 shows the steps for rendering polygons from a retained-mode database and the system components on which they are executed.

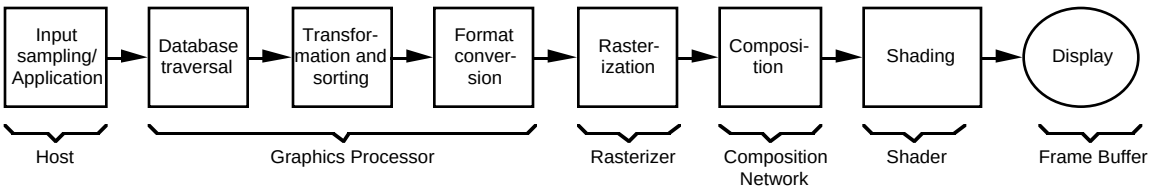


Figure 6.12: Assignment of rendering steps to processing units.

We now briefly describe the major steps of this rendering algorithm.

Host computer. In a typical application, the host computer is the overall system master. It loads code and data onto the various system boards, distributes the display database over the rasterizers, samples user inputs, and determines the changes to be made during each frame. Because the host is under-powered compared to the rest of the system, and because the host-interface is a low-bandwidth link, the host only participates minimally in image generation.

The host communicates with the rest of the system over the message-passing network. It prepares editing commands each frame that describe changes needed in the next frame. It transmits these to the renderers. It can keep up with the rest of the system as long as the editing commands are few and easy to calculate.

Renderers. The renderers maintain a portion of the distributed display database in their local memories. Each frame, they execute the editing commands transmitted by the host, then transform and rasterize their portion of the primitives.

---

[1] A custom corner-turner chip was built for Pixel-Planes 5. The corner turner for the PixelFlow frame buffer has slightly different requirements. It could be implemented as a custom chip as well, or possibly as a programmable gate array [ACTE91, XILI91].

Because the system uses Pixel-Planes 5 style rasterizers, it rasterizes one 160x128 region of the screen at a time. This means that primitives must be sorted into bins for each screen region before rasterization can take place. The straightforward method for doing this is to transform all of the primitives and copy the rendering commands into the appropriate bins, as in Pixel-Planes 5 [FUCH89]. The problem with this is that it costs a frame of latency: one frame is transformed and bin-classified while the previous frame is rasterized. We plan to save this frame of latency by classifying primitives into bins in a preprocessing step, similar to the one described for A-buffer architectures in Section 5.2.3. Primitives will be bin-classified with the minimum amount of calculation possible, a fraction of the overall front-end task.

After the primitives have been sorted, they are converted into IGC instructions, and rasterized one bin at a time. The IGC and EMC array rasterize the primitives for each region. If several samples are required for each pixel, the scene is rasterized several times, with slightly different screen-space offsets for each pass. When a region has been rasterized, it is copied into the EMCs' transfer buffer, and composited over the image-composition network.

Shaders. Shaders, if they are used, intercept composited, rasterized regions, and compute a shading model on each pixel in the region. When antialiasing with several samples per pixel, successive regions sent to each shader contain successive samples. The samples are blended together using precomputed blending coefficients stored at each shader.

When all of the samples have been blended, the antialiased, shaded region of pixels is transferred over the composition network to the frame buffer.

Frame Buffer. The frame buffer receives shaded, antialiased pixels and stores them in the appropriate portion of its refresh memory. When all of the regions for a frame have been received, it switches buffers, and displays the new frame.

Note that rendering occurs in a discrete fashion: one 160x128 region is rasterized/shaded at a time. Each board in the system must know the type and location of each region to handle it in the appropriate way: whether to composite it, load it into a shader, unload it, forward it on without processing, and so forth. This is done by distributing a *rendering recipe* to each board before rendering begins. The rendering recipe is a list of the regions to be transferred over the image-composition network and the corresponding action to be taken for each region. For some boards, the rendering recipe is simple. For example, the frame buffer's rendering recipe simply states whether the region is to be stored and where, and at which point it should switch buffers. The rendering recipe for a shader is more complicated. It must load regions, blend other regions in, forward some regions on without processing, and unload other regions.

Performance. We have written a software timing simulator for the prototype system. It is parameterized by the number of renderers and shaders, and can be loaded with statistics for any desired image. It has an accurate model of the time required for different types of processing and data transfers, and provides appropriate rendering recipes for each system component. Details on the simulation method and simulation results are contained in Chapter 9.

CHAPTER 7

IMAGE-COMPOSITION NETWORK

The image-composition network is the backbone of the PixelFlow system. It carries pixels from renderer to renderer on their way to the frame buffer, compositing each renderer's pixels in as it goes. It performs the per-pixel sort, allowing multiple renderers to work in parallel on the same image with very little overhead. This provides the architecture with its property of near-linear scalability.

The image-composition network is implemented as an extremely high bandwidth, unidirectional datapath that extends across the entire system backplane (or backplanes—in a multiple-rack system). The backplane provides point-to-point wiring between boards, but the active circuitry is contained on each board.

The composition network is 160 bits wide and runs at 80 MHz, providing 1.6 Gbyte/second of bandwidth between adjacent boards. This raw bandwidth determines the maximum screen resolution, frame rate, and number of samples per pixel in the system. These parameters can be traded off against each other, but their product must be less than the total bandwidth.

Individual compositor nodes in the network are configurable under program control. This enables the same hardware to function as a compositor in a renderer board and as a network interface in shader boards.

The composition network is composed of two parts: a *data path*, which carries pixel data from board to board, and a *control path*, which synchronizes and sequences transfers over the network. We now describe these parts in more detail.

7.1 DATA PATH

The datapath is a 160-bit wide point-to-point communication network that connects each pair of adjacent boards in the system. The interconnection wires are part of the system backplane. High-density connectors bring the data path wires out to each board, along with associated control signals.

All of the active circuitry for the data path is contained on individual renderer/shader and frame-buffer boards. Although logically part of the image-composition network, they are placed on application boards for convenience. The data-path portion of the composition node on a renderer/shader is composed of 80 surface-mount compositor chips, placed near the backplane edge of the board. The compositors interface directly to the EMCs on the board, and are controlled by the control path, which we will describe in Section 7.2.

7.1.1 Compositor PLD

The compositor chips are small, surface-mount, devices that implement the composition function on two bit-serial pixel streams at once. Each chip composites the two pixel streams from its corresponding EMC's serial port with two bit-serial pixel streams from the upstream renderer (see Figure 6.8). The EMC interface runs at 40 MHz; the compositor input and output interfaces run at 80 MHz (the 80 • 2 wires of the input and output ports account for the 160-bit width of the data path).

Clock speeds of 80 MHz require special attention. This is near the limits of the fastest TTL logic families, and precludes wire lengths of more than a few inches. Fortunately, programmable logic devices (PLDs) have recently become available with propagation delays as low as 5 nsec. Gazelle Microcircuits recently introduced a line of fairly complex PLDs fabricated with a special gallium-arsenide process with speeds in this range. Although they are not field programmable, they can be programmed rapidly by laser in the factory and are available at reasonable prices (approximately $25 in quantity in 1991) [GAZE88].

The Gazelle 7 nsec 22VP10, a 24-pin PLD with 22 signal pins and up to 10 outputs, is sufficient to perform the composition operation. Figure 7.1 lists the timing specifications for the Gazelle 22VP10-7.

Symbol	Description	Min	Max
t_{PD}	Input to non-registered output	3 nsec	7.5 nsec
t_{EA}	Input to output enable	3	7.5
t_{ER}	Input to output disable	3	7.5
t_{CO}	Clock to output	3	6
t_S	Input setup time	3	–
t_H	Input hold time	0	–
f_{MAX}	Maximum frequency	111 Mhz	–

Figure 7.1: Timing specifications for the Gazelle 22VP10-7 PLD (from [GAZE88]).

The compositor's operating mode is determined by two mode inputs. These select one of the following three operations (see Figure 7.2):

- *Composite*. Composite the two pixel streams from the EMC with the two pixel streams at the input port. Transmit the result over the output port.
- *Load/Forward*. Load the input stream into the EMCs and forward it to the output port without modification.
- *Unload*. Send data from the EMC over the output port. Ignore pixels arriving at the input port.

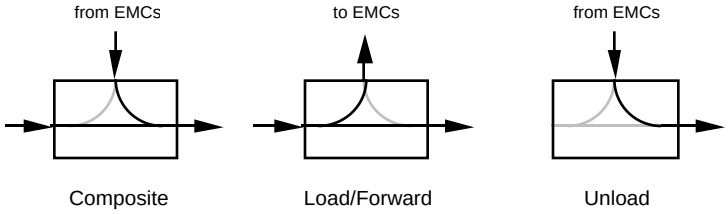


Figure 7.2: Compositor operating modes.

The *Composite* mode requires that pixels have their *z* value first and its bits ordered from most-significant bit (MSB) to least-significant bit (LSB) (see Section 6.3.4). When the nearer pixel has been determined, the compositor sets state bits recording which pixel is in front and that the priority decision has been made for this pixel. These two state bits determine which pixel is transmitted for the remaining *z* bits and all of the other data bits in the pixel.

The *Composite* mode is the only mode needed for renderer boards. Shaders need the other modes, however, to load regions into their EMCs, unload regions that have been shaded, and to forward regions that pertain to other shaders. *Load* and *Forward* modes are really distinct operations as far as the shader is concerned; it either loads data from the composition network or ignores the data and forwards it onward. The two operations are combined into the single *Load*/*Forward* compositor mode for simplicity.

7.1.2 Wiring Delays and Clock Distribution

To minimize signal propagation delays, compositor chips are placed as close to the backplane edge of the board as possible. An 80 MHz clock period is only 12.5 nsec. The output propagation delay from the compositors is 6 nsec. The input setup time is 3 nsec. Adding approximately 1 nsec for clock skew leaves approximately 2.5 nsec for signal propagation—assuming there are no reflections or ringing. This corresponds to a maximum signal-path length of 15 inches or 38 cm.. This requires that all data-path signals be routed carefully and terminated properly. Series termination at the transmit end reduces the power consumption to a tolerable level and reduces forward crosstalk.

High-density card-edge connectors are available with pin spacing of 100 mils laterally and 50 mils longitu-

dinally [TERA89]. A four-rank connector provides 40 connections per inch. The 320 datapath connections for each board, therefore, require 320 / 40 = 8 inches of board edge. This is feasible with a 9u (14.437 inch) height board, even allowing connector space for the control path, the message passing network (approximately 100 pins), and power, ground, and clock connections.

High-speed signaling requires a low-skew clock-distribution scheme. Vernon Chi invented a clever method called *salphasic* clock distribution for distributing clocks with very low skew over a large system. This method was demonstrated in the Pixel-Planes 5 system [CHI90]. The basic notion is to distribute clock signals as a standing wave of the proper frequency. A standing wave has the property that every point along it has the same phase. Differential amplifiers capture the zero crossings of the standing wave and convert them into clock signals. A 160 MHz clock distributed across three 19-inch racks in Pixel-Planes 5 has a measured clock skew of less than 200 picoseconds between any two boards. We can readily employ this scheme, or some variation of it, in the prototype PixelFlow system.

7.1.3 Region-by-Region Transfers

Transfers over the image-composition network have a quantum of one region size. The number of bits in a pixel is variable: it can be either 64 or 128 bits. Once a transfer begins, however, all the pixels in a region—whatever the pixel size—are transmitted from the source board(s) and loaded into the destination board(s).

The clock rate and pixel size determine the transfer time for a region. With 64-bit pixels, the transfer time is 160 • 128 pixels • 64 bits/pixel / (160 • 80 Mbit/sec) = 102.4 μsec. For 128-bit pixels, the transfer time is twice as long, 204.8 μsec.

Region transfers typically perform one of two tasks. They either composite regions of pixels from renderers or transfer shaded pixels to the frame buffer. (Other operations are possible, such as transferring a region from one renderer to another, but these are not needed in standard polygon rendering). Each region transfer has a particular meaning to each board in the system, and the format of the pixel data may differ, depending on the type of transfer. Each board must know the purpose of the transfer to configure itself correctly. The schedule of transfers for an individual board is called a *rendering recipe*. The rendering recipes on each board are the distributed controllers for the parallel system. Rendering recipes will be discussed in detail in Section 9.1.2. Individual transfers are synchronized by the control path, which we will discuss next.

7.2 CONTROL PATH

Transfer operations must be synchronized with great precision and low overhead to make maximum use of the image-composition network. When a transfer begins and the upstream renderer begins to transmit pixels, downstream renderers must be ready to receive them at the correct 80-MHz clock cycle. The composition network also must determine when all of the boards are ready to begin the next transfer. This decision must be made rapidly, since any delay affects the net bandwidth that can be achieved over the image-composition network, hence, the system's maximum speed.

The *control path* implements these two functions. In addition, it synchronizes transfers with other components on each board.

7.2.1 Ready and Go Chains

The *ready chain* and *go chain* are hardware interlocks for synchronizing transfers over the composition network. A *ready* token propagates upstream from board to board, indicating when boards are ready for the next transfer. A *go* token propagates downstream, marking the start of a transfer.

Each compositor has a *ready/go controller* with three inputs and three outputs. The *XferReady* input comes from the rasterizer. It indicates when the rasterizer is ready for the next transfer to begin. *XferReady*, *ReadyIn* and *ReadyOut* implement the ready chain; the *ready* token is encoded by low-to-high transitions of *ReadyOut* on each board. *GoIn*, *GoOut*, and *XferGo* implement the go chain; the *go* token is encoded by low-to-high transitions of *GoOut* on each board. Figure 7.3 shows a block diagram of the ready/go controller.

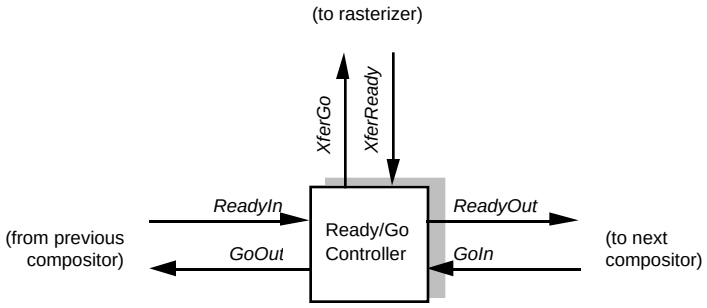


Figure 7.3: Ready/go controller.

The frame buffer asserts *ReadyOut* when it is ready for the next transfer to begin. The ready/go controller on each board receives *ReadyIn* from the downstream compositor and outputs *ReadyOut* to the upstream compositor. It asserts *ReadyOut* after *ReadyIn* and *XferReady* have both been asserted. The transfer begins when *ready* reaches the upstream renderer. Thus, each board can delay the transfer until it is ready to begin.

Go propagates through the composition network in the opposite direction of *ready*. The upstream compositor asserts *XferGo* and *GoOut* to begin a transfer. When each compositor receives *GoIn*, it asserts *XferGo* to the board and *GoOut* to the downstream compositor. The boards cannot veto *go* in the same manner they can veto *ready*; they lost the privilege when they asserted *XferReady*. The arrival of *go* means that *n* clock cycles later (*n* is determined by the startup delay of the output sequencing circuitry), pixels will flow over the composition network. When the transfer has completed, the controller deasserts *XferGo* and *GoOut*, the frame buffer (when it is ready) asserts *ReadyOut*, and the next transfer cycle begins.

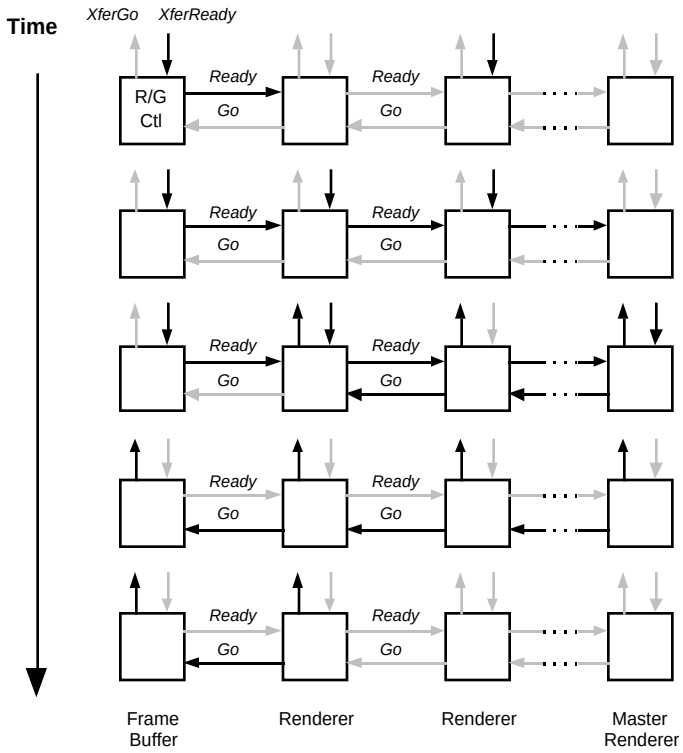


Figure 7.4: Ready and go chains at various stages during a transfer.

The ready/go controller on the upstream renderer has a slightly different function. Rather than passing ready upstream and go downstream, it simply asserts *GoOut* after it receives *ReadyIn* and *XferReady*. Since this controller determines when a transfer actually begins, it is called the *master controller* . The controller on each board can function as either a slave or master. The mode is selected by a bit in the compositor configuration register (see Section 8.2.2). Figure 7.4 shows the ready and go chains with master and slave controllers at various stages during a transfer cycle.

The incoming pixel stream is directed to an array of 54 high-speed 22V10 PLDs, similar to the compositors on a renderer/shader board. These PLDs demultiplex the 160 wires running at 80 MHz to 320 wires running at 40 MHz (see Figure 6.11). In addition, they forward the original pixel stream back onto the image-composition network so that more than one frame-buffer board can be included in a system.

Note that all of the signals in the ready and go chains (except *XferReady*) operate at 80 MHz to keep up with the composition-network data path and to reduce overhead. Even so, the overhead from the ready and go chains can be noticeable in large systems. In a 40-board system operating at 80 MHz, the overhead is 40 • 2 • 12.5 nsec = 1 μsec., which is approximately 1% of the transfer time for 64-bit pixels.

7.2.2 Controlling the Compositors

The *XferGo* signal from the control path indicates the beginning of a transfer. This signal may arrive at any 80-MHz clock cycle after *XferReady* has been asserted. The *compositor sequencer* configures and sequences the compositors and the EMCs' serial ports so that they are ready to composite, load, or unload pixels *n* clock cycles after *XferGo* is asserted.

The compositor sequencer has two parts: a configuration register and a timer. The configuration register stores the control bits that determine the compositor's operating mode (*Composite*, *Load*//*Forward*, or *Unload*) and the EMC serial port's transfer mode (*read* or *write*). It is loaded by a special IGC instruction called *IGC_COMP_CONFIG* (Section 8.2.2 describes the IGC interface to the image composition network).

The timer keeps track of the number of pixels that have been composited and enables and disables the EMC serial port at the appropriate time. It is preset before the transfer begins by the IGC instruction *IGC_COMP_LEN* to the appropriate length for 64 or 128-bit pixel transfers. After *XferGo* is asserted, it asserts the *XferEnab* signal to the EMC serial port for 8,192 80-MHz cycles for 64-bit transfers and 16,384 80-MHz cycles for 128-bit transfers, then deasserts *XferEnab*. This is the precise length of time required to transfer all of the pixels in one region.

7.3 PERFORMANCE MODEL

We now briefly estimate the image-composition network's performance. We first consider a simple system consisting of renderers and frame buffer only. Then we consider more complicated systems, which include shaders and perform supersampling. We will be concerned with image-composition–network performance only in this section. Overall system performance depends on other system components as well. This requires simulation and is analyzed in Chapter 9.

7.3.1 Simple Rendering Algorithm

A simple system configuration consists of a host interface, renderers, and a frame buffer. Without shaders, such a system cannot antialias and is restricted to simple rendering algorithms, such as Gouraud shading.

The image-composition network operates in the single mode, *Composite*, in this system. Regions of pixels are rendered on each of the renderers and composited over the composition network on the way to the frame buffer. 64-bit pixels are ample to encode the *z* value and color value for each pixel.

The performance of the image-composition network in this case is its raw bandwidth minus the overhead for setting up transfers. Consider a single transfer. The transfer itself requires 102.4 μsec. Setting up the transfer requires copying 64 bits of data from a rendering buffer in the EMC to the transfer buffer (64 • 2 ALU operations) plus, perhaps, 10 more operations to change configurations, etc. With a clock speed of 40 MHz, this is a total of (128+10) • 25 nsec = 3.5 μsec. The total time for a transfer is, therefore, 102.4 + 3.5

= 105.9 μsec or 9,443 regions per second. This is sufficient to update a 1280 x 1024-pixel high-resolution image at 147.5 updates per second!

7.3.2 Shaders and Supersampling

Next consider a system with shaders for deferred-shading calculations and supersampling. The image-composition network's behavior is more complicated in this case: rendering and composition occurs as before (though, perhaps with more data transmitted per pixel). Composited pixels are loaded into shaders, rather than the frame buffer, however.

Assume that we have a system with *s* shaders and are supersampling with *k* samples per pixel. Since each shader is responsible for one region at a time, the system will process *s* regions at a time. The first step is to rasterize the first sample for the first *s* regions. These are loaded into the shaders (*s* transfers). Then the remaining *k*-1 samples are rasterized and loaded into the shaders ($s \cdot (k-1)$ transfers). Each shader shades each region-sample and blends it in with the previous samples while the other shaders are being loaded. After a total of $s \cdot k$ cycles, each shader contains the blended color values for its region, and sends its color values to the frame buffer.

Unfortunately, there is only a single path into the EMCs from the compositor chips. This means that shaders cannot send blended pixels to the frame buffer at the same time they receive a new rasterized region. We can overlap these transfers almost completely by having the renderers perform a null transfer while the first shader unloads its pixel data to the frame buffer (see Figure 7.5). We call these null transfers *burps*. Once the first shader has unloaded its pixels, the next shader can unload its pixels while the first shader is reloaded. This continues until all of the shaders have unloaded their pixels.

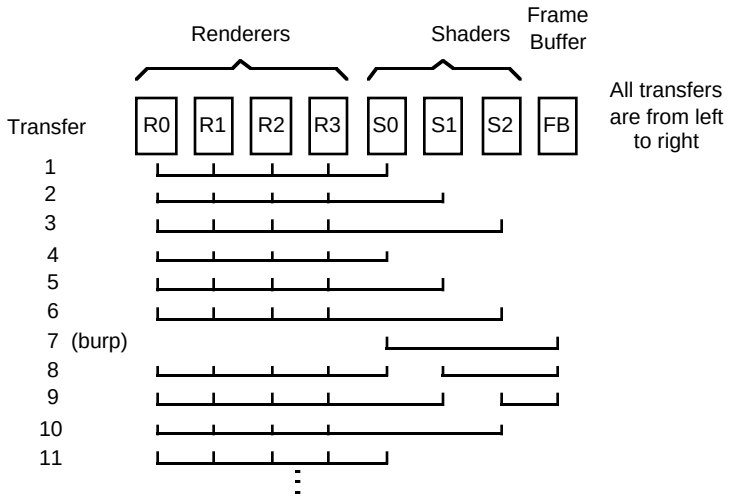


Figure 7.5: Transfers for system with 4 renderers and 3 shaders, computing 2 samples per pixel.

Burp transfers require an extra transfer every *s* supersampled regions (every $s \cdot k$ transfers). This overhead is significant if either *s* or *k* is small. Phong shading requires at least 3 shaders, and reasonable antialiasing requires at least 5 samples. The overhead for burps in this case is 1/15 = 6.67%.

A further source of overhead exists. When the last region of the image has been rasterized and composited, it still must be shaded and transferred to the frame buffer. This could be pipelined with compositing for the next frame—if the next frame can be rendered immediately (this would be the case in a real-time application like flight simulation). If there may be a pause before the next frame, the current frame must be finished without delay, since the delay may be arbitrarily long. Shading the last regions requires *s* region times. Transferring the shaded regions to the frame buffer requires an additional *s* region times. Unfortunately, these cannot be overlapped with other region transfers. If we are rendering a 1280 x 1024 image (64 regions) with 5 samples per pixel, and 3 Shaders, this amounts to $6/(64 \cdot 5) = 1.88\%$ overhead.

We can derive a formula for the total number of transfers required to render an image with r regions, s shaders, and k samples per pixel. It is:

transfers	=	$(s + 1) \cdot r / s +$	// first sample (includes burp)
		$(k – 1) \cdot s \cdot r / s +$	// other samples (no burp)
		$2 \cdot s$	// empty pipe
	=	$rk + r/s + 2s$	// total

This formula makes intuitive sense: the rk transfers are the ideal number for r regions and k samples; the r/s transfers are the number of burps; and the $2s$ transfers are the number of transfers to finish the last regions. If the number of regions is not a multiple of the number of shaders, some shaders will be idle as the last regions are computed. The formula in this case is more complicated. Let $b = ceiling\ (r/s)$. The number of transfers in this case is given by:

transfers	=	$(s + 1) \cdot b +$	// first sample (includes burp)
		$(k – 1) \cdot s \cdot b +$	// other samples (no burp)
		$2 \cdot s$	// empty pipe
	=	$skb + b + 2s$	// total

Again, the result is intuitive: skb is the theoretical number of regions (taking the idle shaders into account), b is the number of burp transfers, and $2s$ is the number of transfers for the last regions.

Figure 7.6 shows the expected frame rate, considering image-composition–network performance alone, for a range of rendering and system parameters. The transfer time, as mentioned above is 102.4 μsec for 64-bit regions and 204.8 μsec for 128-bit regions.

Screen resolution	Pixel size	Shaders	Samples	Percent overhead	Total transfers	Frames/ second	Application
640x512	64 bits	0	1	0	16	610.3	Gouraud
640x512	64	2	5	13.0	92	106.1	Gouraud, antialiased
640x512	128	4	1	42.8	28	174.4	Phong
640x512	128	4	5	13.0	92	53.1	Phong, antialiased
1280x1024	64	0	1	0	64	152.6	Gouraud
1280x1024	64	2	5	10.1	356	27.4	Gouraud, antialiased
1280x1024	128	4	1	27.3	88	55.5	Phong
1280x1024	128	4	5	7.0	344	14.2	Phong, antialiased

Figure 7.6: Composition-network performance under varying conditions.

CHAPTER 8

RENDERER/SHADER BOARD

A renderer/shader board implements an entire *z*-buffer rendering engine capable of rendering 70,000 *z*-buffered triangles per second and shading 160x128 pixels in parallel. A PixelFlow system can be configured with any number of renderer/shaders. Renderers are responsible for storing the display database, traversing it, transforming and rasterizing primitives, and compositing the rendered pixels with pixels from other renderers. Shaders receive rendered pixels from the image-composition network, buffer them, evaluate a shading model for all of the pixels in a 160x128-pixel region in parallel, and forward the pixels to the frame buffer. Although a single board type is used to implement both functions, individual renderer/shader boards must be designated as renderers or shaders when each application program begins.

A renderer/shader board has three main parts: a *graphics processor*, a *rasterizer*, and a *compositor*. The graphics processor is a fast floating-point processor based on the Intel i860 microprocessor. It is similar to the Graphics Processor board in Pixel-Planes 5 [MOLN89]. It contains its own local memory and has access to the message-passing network and the rasterizer. The graphics processor is responsible for traversing a partition of the display dataset (stored in its local memory), transforming primitives from object coordinates into screen coordinates, computing instructions to be passed to the rasterizer, and sorting these into bins corresponding to individual screen regions.

The rasterizer is a 160x128-pixel SIMD logic-enhanced memory array. It is similar to the Renderer board in Pixel-Planes 5 [EYLE90]. It rasterizes using the Pixel-Planes approach of processing every pixel in the array in parallel. The array can be moved from region to region, processing all of the primitives that fall into each region, until the entire image has been calculated [FUCH89].

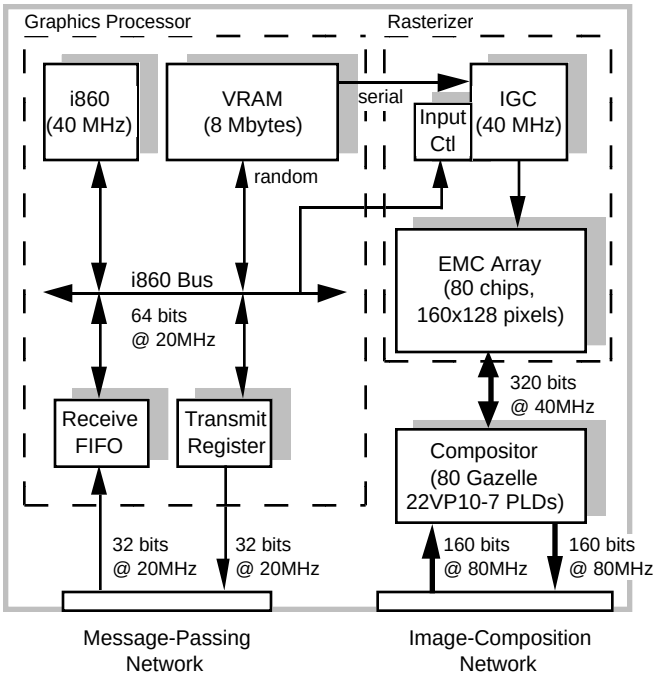


Figure 8.1: Block diagram of a renderer/shader board.

Associated with the rasterizer is a compositor. The compositor merges pixel data from the rasterizer with pixel data flowing over the image-composition network. It combines pixels arriving on the 160 input wires (from the upstream renderer boards) with the corresponding pixels calculated at the current renderer. The visible RGB and *z* values are passed on to downstream renderers, shaders, or a frame buffer on 160 outgoing wires.

Figure 8.1 shows a block diagram of a renderer/shader board. The following sections describe its components in more detail.

8.1 GRAPHICS PROCESSOR

The *graphics processor* (GP) consists of an Intel i860 microprocessor, 8 Mbytes of VRAM memory, and a message-passing–network interface. The network interface occupies one node on the ring network, and allows devices on other ring nodes to send programs and data to the GP, and to receive data from it. The GP consists of six main modules: the i860, its VRAM memory, a ring transmit interface, a ring receive interface, a status register, and an EPROM for booting the i860. The GP is very similar to the Graphics Processor board in Pixel-Planes 5. The main differences are the use of VRAMs instead of DRAMs and the two-way interleaved-memory scheme explained in Section 8.1.2. Figure 8.2 shows a block diagram of a GP.

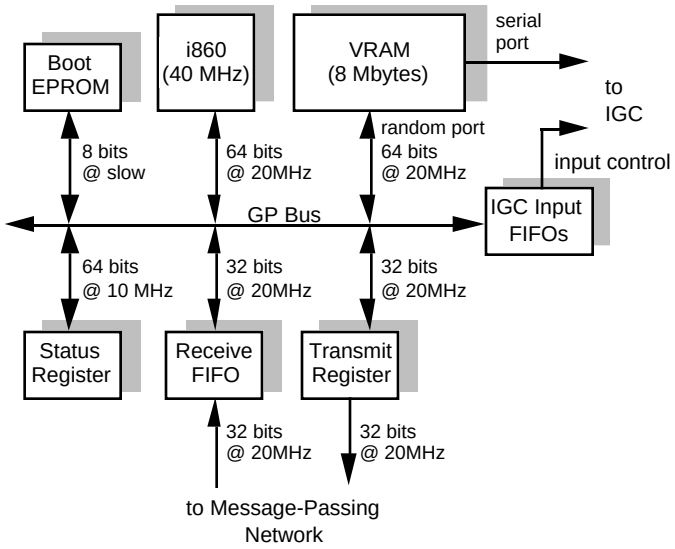


Figure 8.2: Block diagram of the graphics processor.

8.1.1 Intel i860 Microprocessor

This subsection gives a brief overview of the Intel i860 microprocessor, the heart of the graphics processor. For more information, see the *i860 64-Bit Microprocessor Data Sheet* [INTE90a], the *i860 64-Bit Microprocessor Hardware Reference Manual* [INTE90b], or the *i860 64-Bit Microprocessor Programmer's Reference Manual* [INTE90c].

The i860 is a 64-bit RISC microprocessor with on-chip code and data caches. Internally, it contains a RISC integer processor and a 64-bit floating-point processor. The processor runs at 40 MHz and can perform an integer operation and a floating-point operation (potentially a multiply/accumulate) every 40 MHz clock cycle. This gives it a peak processing speed of 40 integer MIPS and 80 MFLOPS. Figure 8.3 shows a block diagram of the i860 microprocessor.

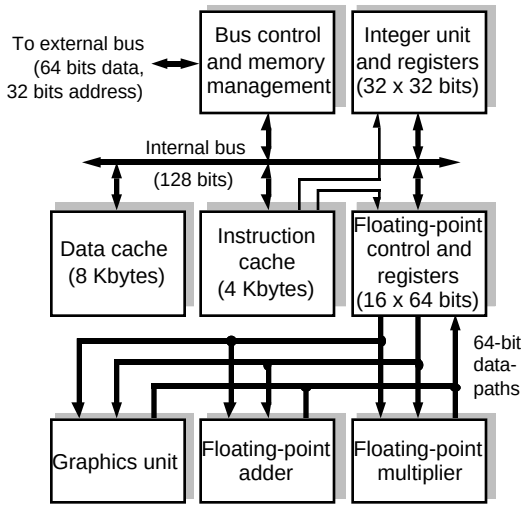


Figure 8.3: Block diagram of the i860 microprocessor.

The integer processor controls overall operation of the i860. It executes load, store, integer, and control-transfer instructions and fetches instructions for the floating-point processor. It contains 32 32-bit registers. It can respond to software-initiated and externally-initiated traps and interrupts. It supports virtual memory with a 64-entry, four-way set-associative *translation lookaside buffer* (TLB). When paging is enabled, the i860 uses the TLB to translate logical addresses to physical addresses and to check for access violations.

The floating-point processor contains a 64-bit floating-point adder, multiplier, and parallel graphics unit. It has its own set of registers, which can be accessed as 32 32-bit registers or 16 64-bit registers. Special load and store instructions allow four adjacent 32-bit registers to be loaded from or stored to cache simultaneously. All floating-point and graphics instructions use these registers as their source and destination operands. Graphics instructions use a special, parallel integer unit to perform multiple additions and comparisons in parallel. Floating-point and graphics instructions have a latency of 3–4 cycles, but generate a new result every clock cycle.

The instruction cache is a 4 Kbyte two-way set-associative memory with 32-byte blocks. It transfers up to 64-bits per clock cycle to the instruction unit. The data cache is an 8 Kbyte two-way set-associative memory, also with 32-byte blocks. It transfers up to 64 bits per clock cycle to the integer unit and up to 128 bits per clock cycle to the floating-point unit. The data cache uses a write-back scheme. Caching can be inhibited by software when desired.

The processor communicates with external memory and I/O devices via a 64-bit bus called the *local bus*. The local bus runs at a 25 nsec pitch with a maximum speed of one cycle every 50 nsec (two 40-MHz clock cycles). It is pipelined, allowing up to three bus requests to be outstanding at once.

8.1.2 VRAM Memory System

The graphics processor contains 8 Mbytes of VRAM memory. The memory is 64 bits wide (to match the local bus) and contains four banks of sixteen 256K x 4 VRAM chips. The VRAMs' random ports are connected to the i860's local bus. Their serial ports are mapped onto the IGC's input port. The VRAMs, together with a DMA controller, form an input FIFO for the IGC (this will be described in detail in Section 8.2.1).

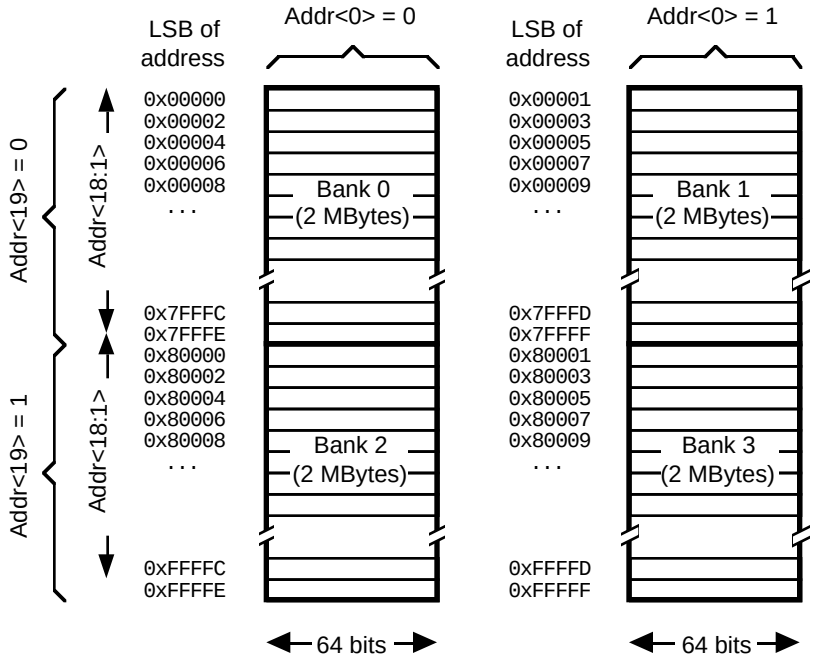


Figure 8.4: Four-bank, two-way interleaved memory organization.

The GP memory uses two-way interleaving to match the fast bus speed of the i860 (50 nsec cycle time) with the slower speed of the VRAMs. Each of the two interleaved pairs of banks has its own memory controller and can execute one fast-page-mode memory cycle in 100 nsec. The interleaved bank pairs or *partitions* are arranged so successive 64-bit memory locations lie in different partitions (see Figure 8.4). Reads and writes to consecutive memory locations (as in cache misses) hit alternate partitions. A high percentage of memory operations are of this type. The average memory cycle time, therefore, can approach the speed of the i860's local bus. This memory design differs from the design used in Pixel-Planes 5, which does not use interleaving. It has been successfully implemented at speed in Sun Microsystems' VX/MVX Visualization Accelerator [SUN91].

Dynamic memories require periodic refreshing to prevent stored data from being corrupted. The GP's memory controller contains a timer that automatically initiates refresh cycles to meet this requirement.

8.1.3 Message-Passing Network Interface

Since there is no high-bandwidth communication traffic over the message-passing network for most applications, the GP's transmit and receive interfaces are fairly simple. The transmit interface is a single 32-bit register. The receive interface is a 1024-word FIFO (a FIFO is needed for the receive interface to meet the ring protocol of being able to accept a message immediately upon arrival).

Transmit Interface. The GP's transmit port consists of a 32-bit register on the i860's data bus. The GP latches data into the register using *shadow reads*—reads from multiply-mapped instances of its data memory. The i860's address space has 2^{32} addresses—more than are necessary to address its VRAM memory and I/O devices. The VRAM memory is mapped into the address space three times: once for normal accesses, once to interpret read data as a destination address, and once to interpret read data as outgoing message data.

When the i860 reads from the destination-address block, the transmit register latches the read data and initiates a message transfer using the data in the register as the destination address. It asserts *TxReady* while placing the destination address on its transmit port. When *TxGo* is received (meaning that the receiver and a ring channel have been acquired), a bit in the status register is set. After loading a destination address, the i860 simply loops waiting for this status bit to be set. It then transmit the data words of the message by reading from the transmit data address block. Since the VRAM memory system

operates at 20 MHz, its speed matches the speed of the ring channel.

When the entire message has been sent, the i860 writes to the *end message* command register (see Section 8.1.4), terminating the message. This shadow-read method of transmitting data is twice as fast as using a memory-mapped register, since only one bus cycle is needed per data word, rather than two, in the case of a memory-mapped register.

Receive Interface. The receive interface buffers incoming messages in the receive FIFO. When the message has been received in its entirety, the receive interface interrupts the i860 with a *RxMsgInt* interrupt. The i860 then reads the message from the FIFO, which is mapped into its address space. The message should contain a word count in one of the first data words, so the i860 knows how many words to read. The i860 signals that it has finished reading a message by writing to the *clear message* command register.

8.1.4 Status and Command Registers

Read and write registers for status and synchronization are memory-mapped onto the i860's local bus. The status register is a read-only register that contains a 16-bit global synchronization timer and several status bits. The command registers are write-only registers for configuring and controlling various board-level devices.

Status Register. The synchronization timer is a 16-bit counter that increments every 50 nsec and interrupts the i860 every time it overflows. It is reset to zero when a system reset occurs, and begins counting immediately after reset is deasserted. As a result, this timer is nearly synchronous from board to board. By keeping the high-order bits of the counter in memory (incremented whenever a timer interrupt occurs), a global system timestamp can be maintained.

The status bits indicate the status of interrupts and other board-level information. The interrupt bits are sticky bits which are set when an interrupt condition occurs and remain set until explicitly cleared by writing to the appropriate command register (described in the following section). The remaining bits indicate board-level information, such as whether:

- IGC FIFOs have room for more data
- IGC FIFOs are empty
- the *TxGo* signal has been received
- a message has been received over the message-passing network

Command registers. Command registers are implemented as special write-only addresses, which reset interrupts bits and implement various board-level commands. Some of the important command-register functions are:

- Reset various interrupts
- Signal the end of an outgoing message
- Signal that an incoming message has been processed

8.1.5 IGC Input FIFOs

The GP sends commands to the rasterizer using two FIFO queues, a *rendering command FIFO* (RFIFO) and a *transfer command FIFO* (TFIFO). The RFIFO stores IGC commands for rasterization. The TFIFO stores copy and transfer commands for the image-composition network.

When the GP prepares commands for a region, it stores them in its own VRAM memory. It queues commands up by writing a control word, consisting of the command's address and length, into a 36-bit hardware FIFO for either the RFIFO or TFIFO. The queue, then, consists of two hardware FIFOs that store control words in order, and the actual data, which resides in VRAM memory.

Both hardware FIFOs are 1K, synchronous FIFOs that are memory-mapped into the i860's address space. Operation of the FIFOs will be explained further in Section 8.2.1.

8.1.6 Graphics-Processor Algorithm

The GPs on each Renderer communicate with the host computer over the message-passing network and the IGC via the IGC input structure. The GP interacts with the host for two purposes: 1) to receive new viewing transformations, editing commands, and the *render* signal, and 2) to inform the host when it has processed these commands and is ready to accept commands for the next frame.

When a GP receives the *render* signal from the host, it transforms all of the primitives in its portion of the database into screen coordinates and classifies them with respect to screen regions. Transformed primitives are stored in blocks of memory called *bin chunks*, which are allocated as needed to store primitives that fall into each region. Note that a given primitive may fall into more than one region. This is a minor source of inefficiency and is analyzed in Section 8.3.1.

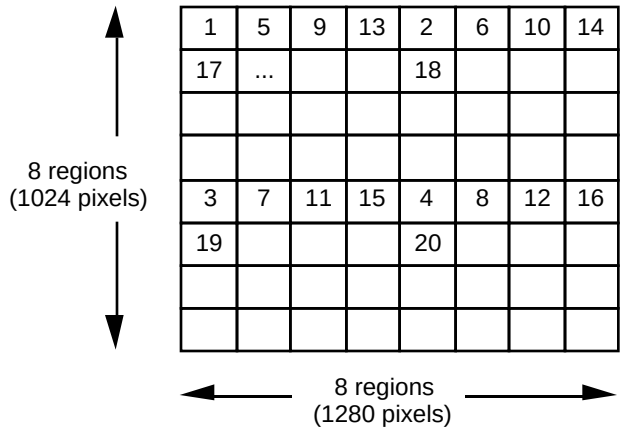


Figure 8.5: Interleaved region processing order.

When all of the primitives have been bin-classified, the GPs go through each screen region, converting primitives in the corresponding bin into IGC instructions for the rasterizer. To improve dynamic-load balance, regions are generally processed in an interleaved order, as shown in Figure 8.5 (and discussed in Section 6.2.5). When every screen region has been processed, the GP is finished with the frame and sends the *finished* signal to the host. Figure 8.6 gives pseudocode for this outer-level algorithm.

```
repeat {
    process editing commands from the host;
    wait for render command from the host;
    bin-classify primitives;
    for each screen region
        process primitives in region
    send finished signal to the host;
}
```

Figure 8.6: Outer-level GP algorithm.

Processing screen regions. To process a screen region, the GP converts all of the primitives in the corresponding bin into IGC commands, stores them in its VRAM memory, and stores a pointer to the commands in the RFIFO.

When all of the commands (primitives) for a region have been loaded into the VRAM and RFIFO, the GP places an additional command, *IGC_REGION_DONE*, in the RFIFO, and two commands, *IGC_REGION_COPY* and *IGC_REGION_XFER*, in the TFIFO. The GP then moves to the next region and repeats the process.

IGC_REGION_DONE tells the rasterizer that it has processed all of the commands for a region;

IGC_REGION_COPY tells the rasterizer to copy the region's pixel data to the transfer buffer in the EMC array; and *IGC_REGION_XFER* tells the rasterizer to composite the region.

If more than one sample is required per pixel, the GP must load additional sets of rendering commands into the IGC FIFO for each new sample point. This can be done by repeating the operations above (using a slightly modified transformation matrix), or by reusing the commands for the first sample point. Since the only difference between commands for different sample positions is a slight offset in *x* and *y* screen coordinates, and *A* and *B* tree coefficients are independent of *x* and *y* displacements, the existing commands can be reused if C is replaced with $C - \Delta x \cdot A - \Delta y \cdot B$.

Adjusting coefficients in this fashion is approximately twice as fast as generating the commands from scratch (see Section 8.3.2). A minor complication is that the initial bin-classification must take all of the sample points into account. This can be done by bloating each primitive's bounding box by the maximum *x* and *y* offsets for any sample point (less than a pixel diameter for typical reconstruction filters). Adjusting coefficients, rather than calculating successive samples from scratch, allows the GP to support supersampling with less than a factor-of-*k* increase in computation.

```
repeat {
    // First sample
    Convert transformed primitives into IGC
        instructions and store in RFIFO;
    Place IGC_REGION_DONE command in RFIFO;
    Place IGC_REGION_COPY command into TFIFO;
    Place IGC_REGION_XFER command into TFIFO;

    // Additional samples
    for each remaining sample
        Adjust coefficients and store IGC instructions
            in RFIFO;
        Place IGC_REGION_DONE command in RFIFO;
        Place IGC_REGION_COPY command into TFIFO;
        Place IGC_REGION_XFER command into TFIFO;
    }
```

Figure 8.7: GP region-processing algorithm.

The GP performs flow control by checking the FIFOs to be sure they will not overflow before writing commands to them. Figure 8.7 gives pseudocode for the region-processing algorithm. Note that this version of the algorithm assumes that one region is processed at a time. Systems with shaders require that multiple regions be processed simultaneously (see Section 7.3.2). This requires minor modifications to this algorithm, which we will illustrate in Section 9.1.1.

8.2 RASTERIZER

The rasterizer converts primitive geometry into pixels when the board operates as a renderer and computes the shading model for each pixel when the board operates as a shader. Its main components are the *Image Generation Controller* (IGC), its associated input FIFOs and control circuitry, and the array of 80 PixelFlow *Enhanced Memory Chips* (EMCs).

The IGC interprets instructions from the GP that are queued in its input FIFOs. It is a microcoded engine that executes high-level IGC instructions, converts floating-point coefficients into the serial form required by the EMCs, and sequences the cycle-by-cycle operation of the EMCs.

The EMCs are CMOS logic-enhanced memory chips, each containing 256 pixel processors. Each pixel processor has a 1-bit ALU, 512 bits of associated memory, and a leaf cell of the linear-expression tree. The pixel processors are arranged as a 160x128-pixel SIMD computing surface, that operates on a 160x128-pixel region in parallel. Each EMC has a 4-bit, bidirectional, 40-MHz serial port. Either 64 or 128 bits of

pixel memory (depending on the configuration) form a *transfer* buffer for communication over the EMC's serial port. The serial port provides the EMCs with access to the image-composition network.

Figure 8.8 shows a block diagram of the rasterizer. The following sections describe the rasterizer components in more detail.

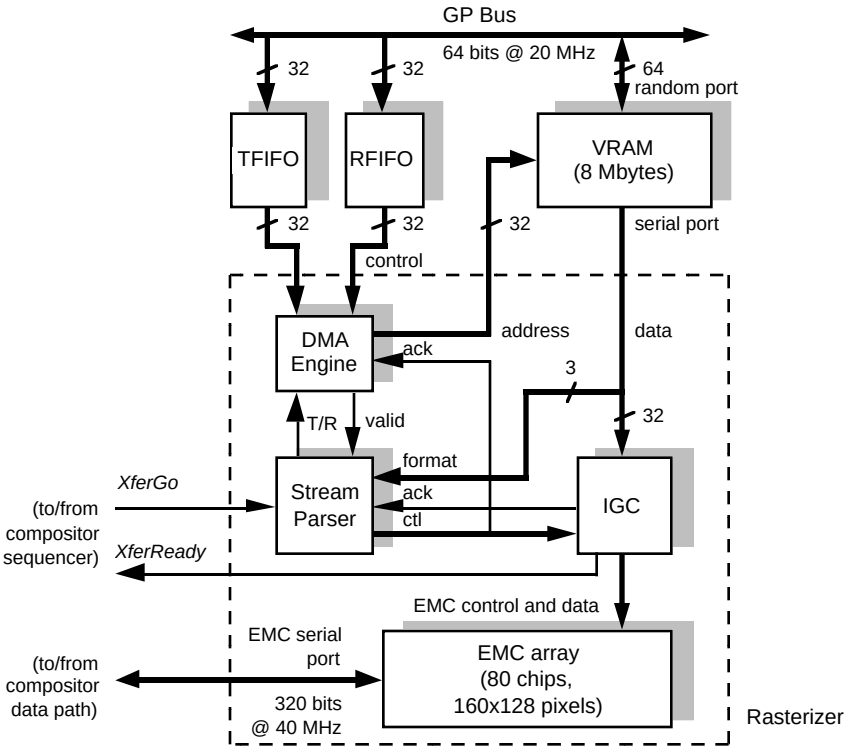


Figure 8.8: Block diagram of the rasterizer.

8.2.1 Input FIFOs and Stream Parser

The rendering command FIFO (RFIFO) and *transfer command FIFO* (TFIFO) provide communication and buffering between the GP and the rasterizer. The two hardware FIFOs store pointers to blocks of data in VRAM and the length of each block. Both FIFOs are needed because the rasterizer performs two types of processing: rasterization, which is the background task, and region transfers, which occur at unpredictable times (when the image-composition network is ready).

A DMA controller reads pointers from the hardware FIFOs and fetches the data from the VRAMs' serial ports. The serial ports of all of the VRAM banks are connected together to a common 32-bit bus and connected to the IGC input port. A three-bit field of the IGC command opcode is fed into an input controller called the *stream parser*. The stream parser decodes these bits, tells the DMA engine which FIFO to read next, and loads IGC commands into the appropriate IGC input register.

Each hardware input FIFO can store up to 1024 control words. Each control word can point to up to 16K words of IGC command data, so the total FIFO capacity is 15 million words per FIFO—twice the GP's memory size! The IGC processes commands at an average rate of about 5 Mword/second for typical algorithms. If the VRAMs' serial ports run at 20 MHz, they can feed the IGC at four times the typical rate, which should be ample for most applications.

The stream parser interleaves commands from the RFIFO and TFIFO based on the state of the image-composition network. It initially reads commands from the RFIFO, but begins processing commands from the TFIFO when a new transfer needs to be (and is ready to be) initiated.

The stream parser accepts two special commands that synchronize transfers over the image-composition network: *IGC_REGION_DONE* and *IGC_REGION_XFER*. It also has three internal variables that regulate the processing of commands from the two input FIFOs and interact with these special commands: *BuffCnt*, *BuffWait*, and *XferWait*.

IGC_REGION_DONE indicates that a screen region has been rasterized and is ready to be transferred over the image-composition network. It is generally loaded into the RFIFO after the commands for rasterizing a region have been loaded. *IGC_REGION_XFER* actually initiates the transfer. It is generally loaded into the TFIFO after the commands which copy pixel data from a buffer in pixel memory into the transfer buffer.

BuffCnt keeps track of the number of occupied region buffers in pixel memory (regions that have been rasterized but have not been transferred). It is incremented when an *IGC_REGION_DONE* command is received and decremented when an *IGC_REGION_XFER* command is received.

BuffWait is a semaphore that blocks the processing of commands from the RFIFO. It is set when all of the buffers in pixel memory are occupied (i.e. *BuffCnt* = number of physical buffers available). It is cleared when the next *IGC_XFER_REGION* command is received, indicating that a transfer can begin and a region buffer is now available.

XferWait is a semaphore that blocks the processing of commands from the TFIFO when set. It is set when an *IGC_XFER_REGION* command initiates a transfer. It is cleared when the transfer finishes (when *XferGo* makes a transition from high to low).

Section 9.1.3 describes the complete rasterizer algorithm. Section 9.1.1 gives a step-by-step illustration of how regions are processed, how the input stream shifts from one input FIFO to the other, and how the *BuffWait* condition is handled. One detail to note is that the DMA engine and IGC must be able be stop processing a set of commands from one FIFO at any IGC command boundary in a block of commands (commands referenced by a single control word). This is necessary to minimize the overhead between region transfers. For example, if a large block of commands from the RFIFO is being processed when *XferDone* is received, the DMA engine must be able to shift to the TFIFO before finishing the block of commands—otherwise it would have to delay the next transfer for an arbitrary length of time, degrading the performance of the image-composition network. The soonest the DMA engine can shift input streams is at an IGC-command boundary. The DMA engine must be able to resume processing the previous block of commands when the transfer commands have been processed.

8.2.2 Image Generation Controller

The *Image Generation Controller* (IGC) controls and sequences the EMCs and synchronizes the rasterizer with the composition network. The IGC is a custom chip with its own microcode store and a serializer to convert floating-point input coefficients into the fixed-point, bit-serial form required by the EMCs. It processes commands from the input FIFOs and stream parser and drives the various control and data inputs of the EMC array.

The IGC's input stream is a series of commands, which can span one or more 32-bit words. The first word in each command is an instruction word, which specifies the starting microcode address for the command and, implicitly, the format of the operands (the operands typically are *A*, *B*, and *C* coefficients for the linear-expression tree).

Each IGC command is executed by an IGC microcode routine that can take several cycles to execute. Sample IGC instructions are: *IGC_LOAD*, which loads tree results into pixel memory; *IGC_MEMpluseqTREE*, which adds tree results to a location in memory; *IGC_SETENABS*, which sets the enable registers in every pixel processor; and *IGC_TREEgeZERO*, which sets the enable registers in pixel processors whose tree results are greater than or equal to zero. The full set of IGC instructions for Pixel-Planes 5 is described in [EYLE90]. Additional instructions, such as *IGC_COMP_CONFIG*, *IGC_COMP_LEN*, *IGC_XFER_REGION*, and *IGC_REGION_DONE*, are required to interface with the image-composition network.

The IGC also has two condition-code inputs that allow it to test external conditions, and several control outputs that allow it to communicate with external devices besides the EMCs. One of these outputs is used for the *XferReady* signal, which tells the compositor that it can begin the next region transfer (see Section

7.2.1). The other control outputs are used to load the compositor configuration register and timer using the *IGC_COMP_CONFIG* command (see Section 7.2.2).

8.2.3 EMC Array

Rasterization is actually performed in the array of 80 PixelFlow EMCs. Each EMC contains a 1-bit ALU and 512 bits of local memory for 256 pixels. The EMCs form a 2-dimensional SIMD processor array that covers a 160x128-pixel region of the screen. This logical array can be "moved" to operate on any region of the display screen.

Each pixel is provided with its own 1-bit processor, an output of the linear-expression tree, 512 bits of local memory, and a 64-bit or 128-bit communication register or *transfer buffer*. Figure 8.9 shows a logical diagram of an EMC.

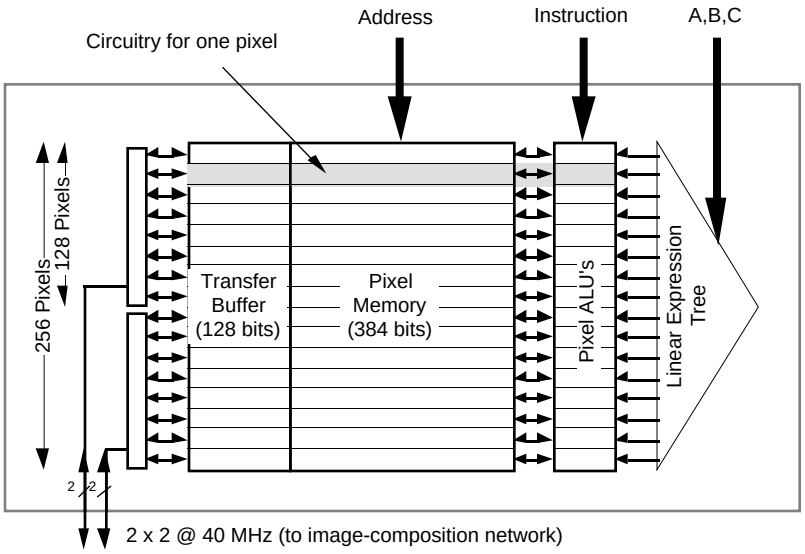


Figure 8.9: Logical diagram of a PixelFlow EMC.

Pixel processors. Pixel processors are general-purpose 1-bit processors with carry registers and enable registers, which allow operations to be performed on a subset of the pixels. The pixel processor can use tree results or local memory as operands and can write results back to local memory. It can also transfer data between memory, the carry register, and the transfer buffer.

Linear-expression tree. The linear-expression tree evaluates bilinear expressions $Ax + By + C$ for each pixel of the array in parallel. A, B, and C are coefficients loaded from the IGC and (x, y) represent the pixel's screen coordinates. Many graphics calculations can be cast into the form of bilinear expressions. For example, the edges, depth, and color values of Gouraud-shaded triangles can be cast into this form [FUCH85].

The IGC controls the operation of the EMC array. IGC instructions and coefficients are serialized and broadcast to all of the EMCs in parallel. The SIMD pixel processors execute these instructions in lock-step. The enable registers in each pixel processor are used to control which subset of the pixel processors are active at any given time.

Pixel memory. The 512 bits of local memory at each pixel are sufficient to store depth, color, and, perhaps, other intermediate data for several screen regions. This allows regions to be buffered in the EMCs to prevent bottlenecks due to uneven distributions of primitives on the screen. When a region of pixels is ready to be composited, the corresponding data is copied from its position in regular pixel memory to the transfer buffer at each pixel.

Transfer buffer. The transfer buffer is a distinguished, 128-bit portion of pixel memory. It can be

accessed in two ways: 1) by the pixel processor (*normal mode*), and 2) via an external serial port (*transfer mode*). In normal mode, the ALUs can access the transfer buffer just like any other part of pixel memory. This allows the ALUs to copy data into or out of the transfer buffer. In transfer mode, the transfer buffer is isolated from the ALUs and connected to the EMC's serial transfer port.

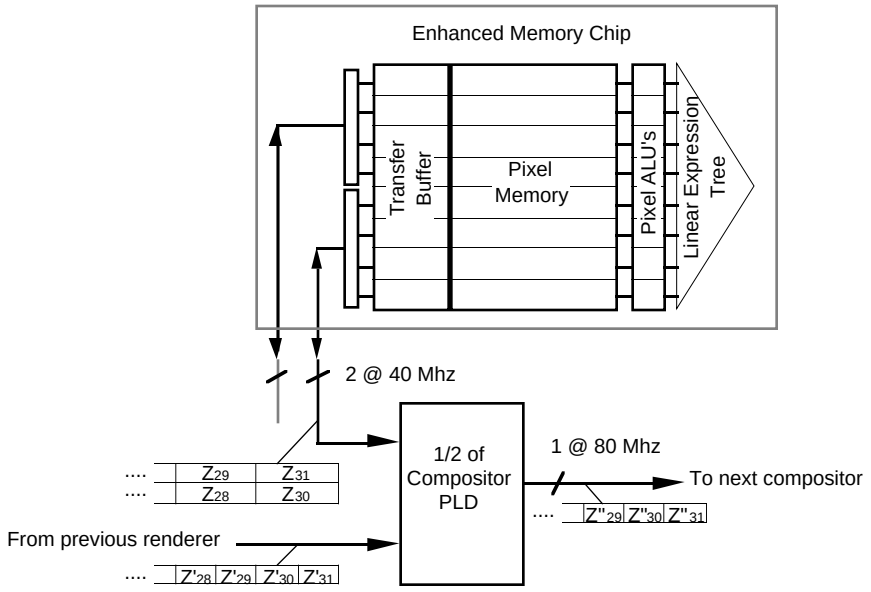


Figure 8.10: Snapshot of the EMC transfer port in operation.

Figure 8.10 shows a snapshot of the transfer port in operation. The transfer buffers in each pixel are connected to a double 2-bit-wide external port on the EMC called the *serial* or *transfer port.* Two bits of each of two pixels are presented at the port simultaneously. On the next cycle, the next most significant two bits of each pixel are presented. The port cycles through every bit-pair of every pixel contained in the EMC's transfer buffer during a transfer. The amount of data transferred is either 64 bits or 128 bits per pixel, depending on an external input called *XferLen*. The transfer port runs at 40 MHz, so 64-bit transfers take 128•32•25 nsec = 102.4 msec and 128-bit transfers take 204.8 msec. The transfer port provides a communication bandwidth of 20 MByte/second per EMC.

Each EMC's transfer port mates with a compositor chip. This provides the high-bandwidth connection between the rasterizer and image-composition network.

EMC feasibility. The PixelFlow EMC described here is very similar to the EMC used in Pixel-Planes 5 [FUCH89]. The three differences are: 1) additional memory per pixel, 2) higher-speed transfer port, and 3) larger, configurable transfer buffer. None of these modifications are difficult to implement.

The Pixel-Planes 5 EMC was built using 1.6μ CMOS technology. A similar 1.2μ CMOS process is available now, providing higher density and faster transistors. The Pixel-Planes 5 EMC used a 6-transistor static cell to implement the pixel memory. A 4-transistor dynamic-memory cell can be substituted easily, doubling the memory density. The faster process makes the 40 MHz transfer port easy to implement. The larger transfer buffer is a minor design change.

8.2.4 Rasterizer Algorithm

The rasterizer processes IGC instructions describing the primitives in the scene and converts them into pixel values. The rasterizer operates in region-sized batches, rasterizing the primitives for one region (one region-sample, when supersampling) at a time. It then copies the pixel data into the EMCs' transfer buffer to await composition.

The background task of the IGCs and EMCs is to rasterize regions. The EMCs contain buffering for up to

four regions in addition to the region being scanned out of the transfer buffer. It continues rasterizing as long as commands are present in the RFIFO and space remains in pixel memory to rasterize the current region.

From time to time rasterization is "interrupted" by high-to-low transitions of *XferGo* from the compositor sequencer, indicating that a transfer has completed. This tells the stream parser to read commands from the TFIFO, which contains *IGC_REGION_COPY* and *IGC_REGION_XFER* commands. These commands copy pixel data from the appropriate buffer in the EMC to the EMC transfer buffer and set the *XferReady* signal, indicating that the next transfer is ready to begin. When these two commands have been processed, the IGC resumes processing commands from the RFIFO.

```
repeat {
     if (!XferWait && !TFifoEmpty) {
          // Process a command from the TFIFO
          read FifoCommand from TFIFO;
          switch (FifoCommand)  {
               case IGC_REGION_COPY:
                    copy region into Transfer Buffer;
                    break;
               case IGC_REGION_XFER:
                    --BuffCnt;
                    XferReady = TRUE;
                    XferWait  = TRUE;
                    BuffWait  = FALSE;
                    break;
               }
          }
     else if (!BuffWait && !RFifoEmpty) {
          // Process a command from the RFIFO
          read FifoCommand from RFIFO;
          switch (FifoCommand) {
               case (any rendering command):
                    do some rendering;
                    break;
               case IGC_REGION_DONE:
                    ++BuffCnt;
                    if (BuffCnt ≥ MAX_BUFFS)
                         BuffWait = TRUE;
                    break;
               }
          }
     else
          do nothing;

     // Reset XferReady and XferWait if necessary
     if (XferGo becomes TRUE)
          XferReady = FALSE;
     if (XferGo becomes FALSE)
          XferWait  = FALSE;
     }
```

Figure 8.11: Rasterizer algorithm.

The state variable *BuffWait* prevents rasterization commands from being processed when no buffer space is available for rasterized pixels. The state variable *XferWait* prevents a new region from being copied into

the transfer buffer until the previous transfer is finished (Section 8.2.1 explains the function of these variables in more detail). Figure 8.11 gives pseudocode for the rasterizer algorithm.

8.3 PERFORMANCE MODEL

In this section we develop a performance model for the renderer/shader board. We must consider the raw processing speed of each module and inefficiencies that may occur during operation. The results developed here are used as parameters for the system simulator described in Chapter 9.

8.3.1 Bin Classification

Since the SIMD rasterizer contains a processor for each pixel in a 160x128-pixel region, the processing time for a primitive does not depend on the size of the primitive, as long as it falls entirely within a region. Large primitives, or primitives that cover region boundaries can affect two or more regions. This increases the number of primitives that must be processed by the rasterizers. We call this phenomenon *bin replication*. We define the *bin-replication factor* for a given scene to be the number of primitives actually rasterized divided by the number of visible primitives.

How large are the bin-replication factors for typical scenes? We will approach this problem in two steps: 1) developing a simplified model, which we will solve analytically, and 2) validating the model with measurements on actual datasets.

Analytic model. We make the following simplifying assumptions: Assume that every polygon in the dataset has an identical screen-space bounding box of width w and height h. Assume, further, that each polygon has an equal probability of falling anywhere on the screen. Let W be the width and H be the height of a screen region.

We can compute the average number of regions covered by a given polygon by the following integral:

$$bin\ \ replication\ \ factor\ \ = \int_{screen} \int p(x\ ,y\ )r(x\ ,y\ )dxdy$$

where $p(x,y)$ is the probability that the center of the bounding box falls at point (x,y) and $r(x,y)$ is the number of regions affected by this placement of the bounding box. If we ignore the effect of the edge of the screen (11% of the region boundaries are screen edges on a 1280x1024 screen), we can reduce this to an integral over a single screen region with $p(x,y) = 1/HW$ over the entire region:

$$\int_0^H \int_0^W p(x\ ,y\ )\, r(x\ ,y\ )\ dxdy = \int_0^H \int_0^W \frac{1}{HW} r(x\ ,y\ )\, dxdy$$

We can determine r(x,y) using the geometric construction shown in Figure 8.12. The screen region is divided into three parts: corner areas, edge areas, and a center area. If the center point of a w x h bounding box falls in a corner area, the primitive must be processed in four screen regions. If it falls in an edge area, the primitive must be processed in two screen regions. If it falls in the center area, the primitive must be processed in only one screen region.

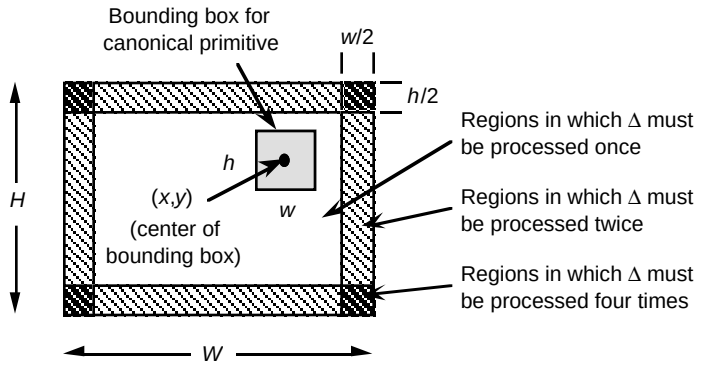


Figure 8.12: Geometric construction for evaluating *r*(*x*,*y*).

Using this geometric construction, the integral reduces to the following:

$$\frac{4 \cdot \textit{area of corner regions} + 2 \cdot \textit{area of edge regions} + 1 \cdot \textit{area of center region}}{HW}$$

$$= \frac{4 \cdot 4\left(\frac{w}{2}\right)\left(\frac{h}{2}\right) + 2 \cdot \left(2\left(\frac{h}{2}\right)(W - w) + 2\left(\frac{w}{2}\right)(H - h)\right) + 1 \cdot (W - w)(H - h)}{HW}$$

$$= \frac{4 \cdot (wh) + 2 \cdot (hW + wH - 2hw) + 1 \cdot (WH - Wh - Hw + wh)}{HW}$$

$$= \left(\frac{w + W}{W}\right)\left(\frac{h + H}{H}\right)$$

This equation was first derived by John Eyles as part of the performance model for Pixel-Planes 5, which has similar inefficiencies due to bin replication. Figure 8.13 plots the predicted bin-replication factor for several square region sizes and square bounding-box sizes. The predicted bin-replication factor for the 160x128-pixel region size of the prototype system is 1.15 for 10x10-pixel bounding boxes and 1.30 for 20x20-pixel bounding boxes.

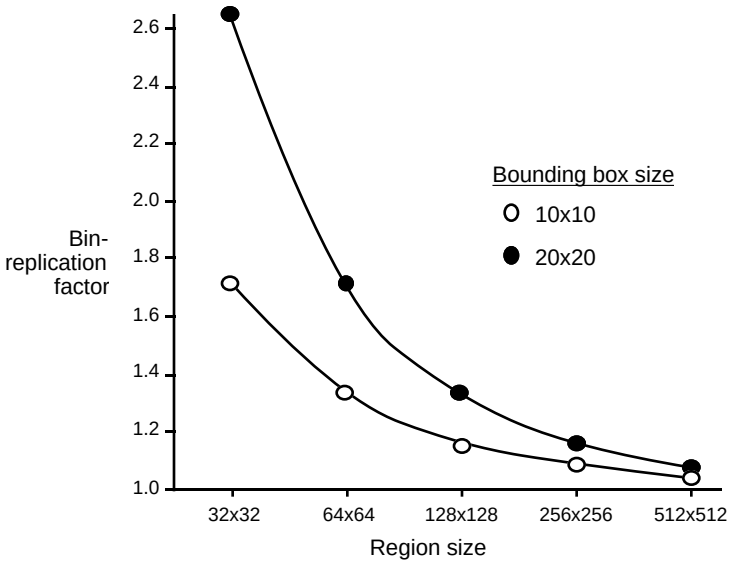


Figure 8.13: Bin replication factor as function of region size.

Empirical measurement. Appendix A contains the actual bin-replication factors for the six sample databases. Figure 8.14 lists the average triangle area and the predicted and actual bin-replication ratios for these databases. To compute *w* and *h*, we assumed that primitives cover one third of the area of the bounding box (an assumption based largely on intuition—it can be quite poor for databases with long, thin primitives). Therefore, $w = h = \text{sqrt}(3A)$, where *A* is the average screen-space area of the primitives in the dataset.

Dataset	Visible triangles	Average triangle area	Bin-replication ratio	
			Predicted	Measured
Space station and shuttle	6,549	264.08	1.43	1.53
Poliovirus	369,819	42.07	1.16	1.17
Radiosity lobby	4,786	1,117.23	1.98	2.21
House	51,922	146.26	1.32	1.25
Earth	128,880	22.94	1.12	1.14
Pipes	137,747	32.28	1.15	1.15

Figure 8.14: Predicted and actual bin-replication factors for sample datasets rendered at 1280x1024-pixel resolution.

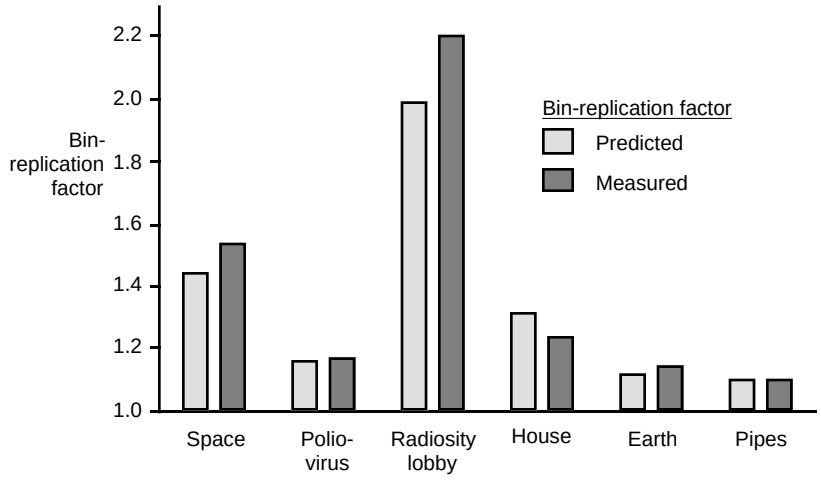


Figure 8.15: Graph of results in Figure 8.14.

Figure 8.15 plots this same information. We can see that the simplified model correlates well with the measured replication factors. For all of the sample databases with large numbers of primitives (excluding the space and lobby databases), the bin-replication factor is less than 1.2. This makes intuitive sense, since high-complexity databases tend to have small primitives. This corresponds to a relatively small amount of rasterization overhead, and should not significantly degrade the performance of the system. Bin-replication affects Pixel-Planes 5, as well, and turned out to have only minor impact on the system's performance.

8.3.2 Renderer Performance

To build a performance model for the renderer board we need to know how fast the different board components bin-sort, transform, adjust, and rasterize primitives. Pixel-Planes 5 uses i860s for transformation and Pixel-Planes EMCs for rasterization, so we used the measured performance of Pixel-Planes 5 as a starting point. In consultation with David Ellsworth, who has performed extensive performance analysis on Pixel-

Planes 5, we arrived at the performance estimates shown in Figure 8.16.

Task	Gouraud shading		Phong shading	
	Clock cycles	Time (nsec)	Clock cycles	Time (nsec)
Graphics processor				
Bin-classification time	225	4,500	275	5,500
Transformation time	250	5,000	300	6,000
Adjusting time	220	4,400	220	4,400
Rasterizer				
Rasterization time	230	5,750	300	7,500
Region copy time	132	3,300	260	6,500

Figure 8.16: Renderer performance model.

The bin-classification time, transformation time, adjusting time, and rasterization time all refer to the time required to process a single primitive. The region-copy time is the time required to copy a region's worth of pixel data from pixel memory to the transfer buffer in the EMCs.

We can use these statistics to test the balance between GPs and rasterizers. To compute a Phong-shaded image antialiased with 5 samples per pixel, the GP spends one bin-classify time (5,500 nsec), one transformation time (6,000 nsec), and four adjust times (4•4,400 nsec) per primitive, a total of 29,100 nsec. The rasterizer spends five rasterization times (5•7,500 nsec) per primitive, a total of 37,500. Thus, the GP and rasterizer are balanced to within 23% for antialiased triangle rendering. Bin-replication overhead affects the rasterizer more than the GP, so this evens the balance further.

8.3.3 Shader Performance

Shader performance is less critical than renderer performance, since we can provide the system with as many shaders as necessary to calculate a given shading model. This is a fixed cost for the system, independent of the number of renderers and the renderers' performance.

From measurements made on Pixel-Planes 5, we estimate that shaders require approximately 400-500 μsec to Phong-shade a region of pixels with a single light source. Since the transfer time for Phong-shaded pixels is 204.8 μsec, between two and four shaders are needed for the system to run at full speed when Phong shading. Additional light sources, or more complex shading models, such as procedural textures may double or triple the shading time.

There is an additional advantage to using more shaders: fewer burp transfers are required, so the effective composition network bandwidth increases. We anticipate that four shaders are sufficient for most deferred-shading applications.

CHAPTER 9

CONTROL ALGORITHMS AND PERFORMANCE

This chapter analyzes system-wide control and performance issues. Section 9.1 describes the distributed control algorithm used to synchronize the graphics processors, rasterizers, and compositors in the system. Section 9.2 investigates the amount of buffering required to minimize dynamic load imbalances. Section 9.3 describes the simulation method and gives performance estimates for a range of databases and system configurations.

9.1 SYNCHRONIZATION AND CONTROL

In previous chapters we described the boards in a PixelFlow system and the portion of the rendering algorithm performed on each board. This section describes the overall synchronization and control of the system. Section 9.1.1 discusses synchronization between the rasterizer and compositor on a renderer board. Section 9.1.2 discusses rendering recipes, the method of high-level control.

9.1.1 Rasterizer/Compositor Synchronization.

Transfers over the image-composition network can be considered to be the heartbeat of a PixelFlow system. They occur extremely rapidly; over 800 transfers are needed to compute a Phong-shaded, antialiased, high-resolution image. Transfers cannot begin until every board in the system is ready.

When the composition network begins transferring a region, the transfer continues without interruption until the entire region has been transferred. The control path of the composition network sequences transfers and synchronizes transfers with devices on each board.

Figure 9.1 shows a simplified view of the rasterizer and compositor components involved in synchronizing region transfers. Here we have lumped the IGC and pixel processors into a single entity called the rasterizer. Buffers 0–3 are portions of pixel memory that can store a region's worth of pixels. Pertinent state variables and synchronization signals are also labeled.

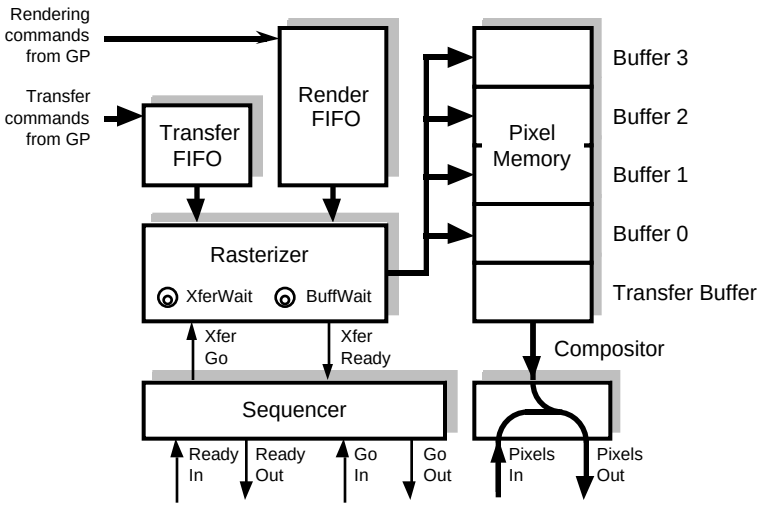
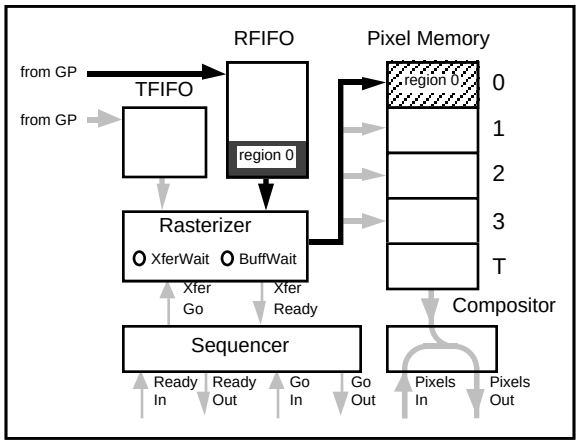


Figure 9.1: Synchronization components on a renderer board.

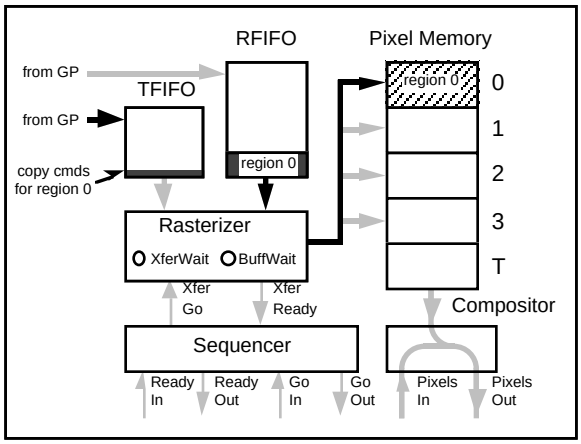
Figures 9.2(a)-9.2(f) show snapshots of the system while rasterizing and compositing several regions. Active signals and datapaths are indicated by solid black lines. Buffers that contain data are indicated by dark gray halftones.

In Figure 9.2(a), Region 0 is rasterized while further IGC instructions for the region are loaded into the RFIFO. In Figure 9.2(b), rendering commands for Region 0 have been completely loaded into the RFIFO; the GP is loading its copy and transfer commands into the TFIFO. In Figure 9.2(c), Region 1 has been completely loaded into the RFIFO and TFIFO; Region 0 is copied into the transfer buffer.

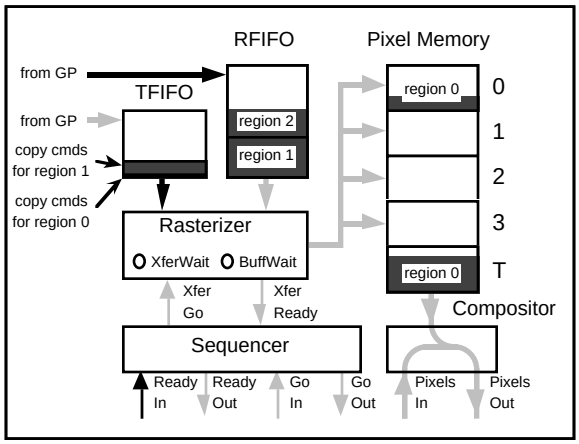
In Figure 9.2(d), the rasterizer processes Region 1; the GP is one region ahead of the rasterizer. Pixel data for Region 0 has been copied into the transfer buffer, causing *XferReady* to be asserted. In Figure 9.2(e), Region 2 is rasterized while Region 0 is composited. In Figure 9.2(f), Region 1 is composited. Rasterization has stopped because no buffer space is available in pixel memory. It can resume when a buffer becomes available. (This scenario is especially likely when rendering datasets with small numbers of primitives).



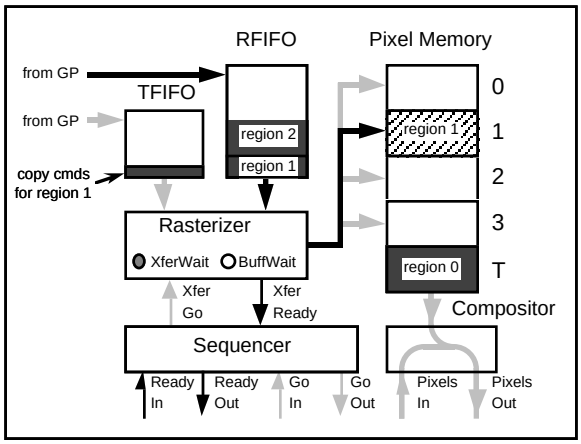
(a) GP loads rendering commands into RFIFO. Region 0 is rasterized.



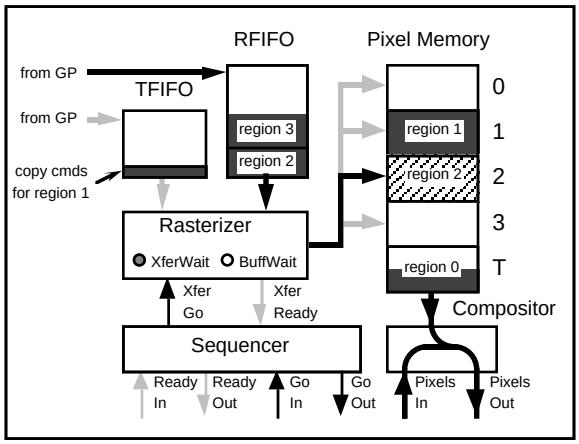
(b) GP loads copy commands into TFIFO. Region 0 is rasterized.



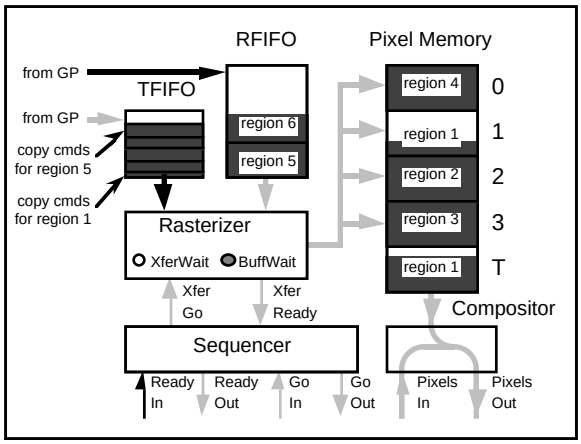
(c) Region 0 pixels are copied from Buffer 0 to transfer buffer.



(d) Region 1 is rasterized. Sequencer waits for 'Go' signal.



(e) Region 0 is composited. Region 2 is rasterized.



(f) Region 1 is copied into transfer buffer. Rasterization pauses because all buffers are full.

Figure 9.2: Snapshot of renderer in various stages of operation.

9.1.2 Rendering Recipes

The interlock scheme above handles low-level synchronization between system components. A high-level control scheme is needed to specify the semantics for region transfers and to control the overall progress of the rendering algorithm. We have adopted a distributed control method, in which each board keeps track of the context for each transfer. Each system module (graphics processor, rasterizer, compositor, frame buffer, etc.) keeps track of the action it must perform during each transfer time. These lists of actions implicitly define the rendering algorithm. We call these lists of instructions *rendering recipes*.

Prior to rendering each frame or series of frames, each module must be given an appropriate rendering recipe. The recipe must take into account information such as the resolution of the display screen, the number of shaders, and the number of samples per pixel. As the frame is computed, each board follows its respective recipe. For example, a shader's recipe may tell it to load pixels arriving over the image-composition network on a particular transfer. The renderers' recipes must tell them to put the appropriate data on the composition network.

During burp transfers, renderers must perform the normal handshake operations even though they are not transmitting data or performing useful work. This means that graphics processors must send *IGC_REGION_DONE* and *IGC_REGION_XFER* commands to the rasterizer, and the rasterizer must hand-shake with the compositor in the usual way, even if the renderer is not transferring pixels.

Polygon rendering. The rendering recipe for a renderer or shader specifies the GP's task for the region, the rasterizer's task, and the nature of the transfer (*Composite*, *Load*, *Unload*, etc.). The rendering recipe for a frame buffer specifies whether to load pixels and where they should be stored in the VRAM array.

Figure 9.3 shows a complete set of rendering recipes to compute a 6-region image without shaders or supersampling. Each row in the figure represents a component in the system. All of the renderers receive the same rendering recipe. Therefore, only one row is provided for the graphics processors and one for the rasterizers. Each column represents a region time during the algorithm (note that the first two region times do not requires transfers over the composition network). The entire algorithm requires only 6 transfers, since no burp transfers are needed.

	Transfer							
	*	*	0	1	2	3	4	5
Graphics Processor	r0	r1	r2	r3	r4	r5		
Rasterizer EMC's		r0	r1	r2	r3	r4	r5	
Rasterizer Comm. Port			r0	r1	r2	r3	r4	r5
Frame Buffer			r0	s1	r2	r3	r4	r5

r_i represents region *i*

* means no transfer needed over composition network

Figure 9.3: Rendering recipes to compute a 4-region image without shaders or antialiasing.

It is appealing to think of the entire system operating in lock step, with each module performing the same step in the rendering recipe at the same time, as shown in the figure. Although this is a useful conceptual model, it is not how the system operates. Only the image-composition network operates synchronously. The rasterizers are separated from the composition network by buffering and may be up to four regions ahead. The graphics processors are separated from the rasterizers by buffering and can be many regions ahead of the rasterizers.

Section 7.3.2 described the more complex sequence of calculations and transfers needed to render a frame with shaders and supersampling. Shaders require multiple region times to shade each region and renderers are required to burp to let shaders send pixels to the frame buffer.

Figure 9.4 shows a set of rendering recipes for a system with 2 shaders computing an image with 2-sample-per-pixel antialiasing. Each shader has a slightly different recipe, since shaders are loaded and unloaded at

different times. Notice that the renderers burp for one transfer every four region-times to allow the first shader to transmit its pixels to the frame buffer. The halftoned cells trace the progress of region 2 through the system.

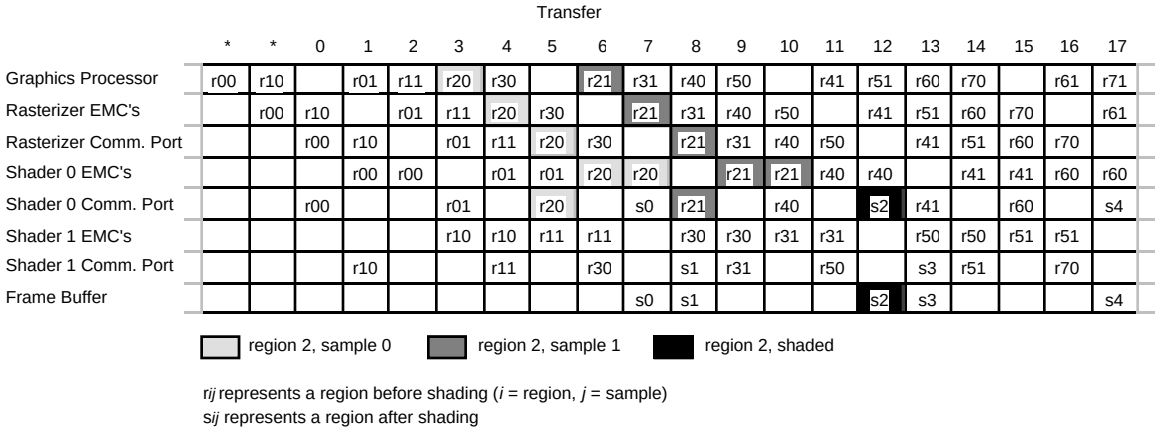


Figure 9.4: Rendering recipes for system with 2 shaders and 2-sample-per-pixel antialiasing.

Figure 9.5 shows rendering recipes for a more complicated system: one with 4 shaders computing 3-sample-per-pixel antialiasing.

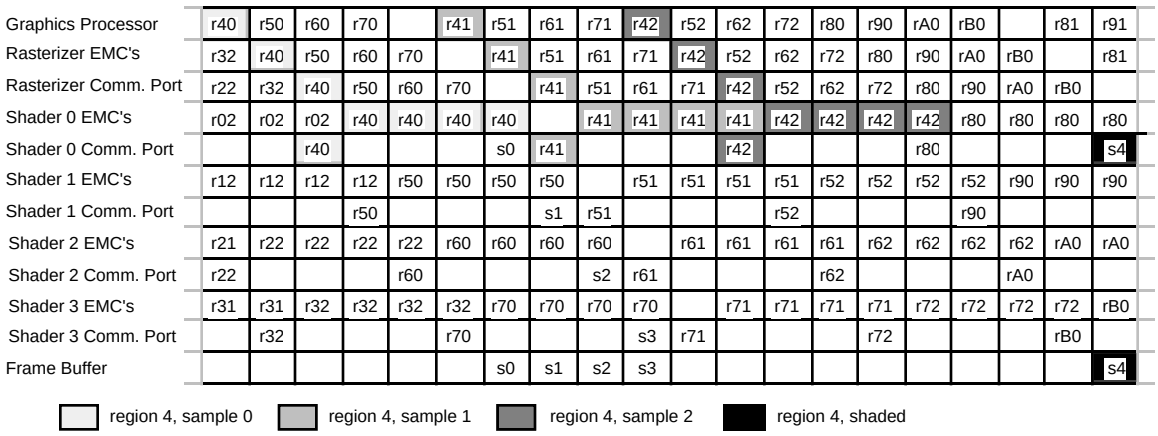


Figure 9.5: Rendering recipes for system with 4 shaders and 3-sample-per-pixel antialiasing.

The i860 on each system board is responsible for storing and administering the rendering recipe for the devices on the board. The recipe can be precomputed and stored as a table of actions, or it can be stored implicitly as an algorithm that runs on the i860. For the system simulator (described in Section 9.3.1), we encoded the rendering recipes into finite-state machines and verified their correct operation. Formulas for the number of region times required to execute the rendering recipes for different system configurations are given in Section 7.3.2.

9.2 DYNAMIC LOAD BALANCE

As described in Section 5.4, dynamic-load balance is a critical issue for image-composition systems that

rasterize and composite in image order. Although a PixelFlow system uses *z*-buffer renderers, it has load-balance properties similar to A-buffer systems.

A crucial design decision is the number of regions of buffering between the rasterizer and composition network. If the amount of buffering is too small, databases with high dynamic-load-balance ratios suffer severe performance penalties. If it is too large, EMC chip area (an expensive resource) is wasted. It is important to get this number right.

Dataset (Phong-shaded)	Dynamic LB. ratio	Regions of buffering				
		1	2	4	8	16
Lobby (4786 Δ's)						
Scattered	1.67	220,553	220,553	220,553	220,553	219,541
Clustered	6.77	138,725	159,533	181,288	192,209	192,209
House (51,922 Δ's)						
Scattered	1.16	1,251,133	1,307,859	1,388,289	1,410,924	1,407,100
Clustered	8.09	476,349	499,731	605,858	815,102	1,022,087
Pipes (137,747 Δ's)						
Scattered	1.05	1,866,238	2,015,023	2,074,191	2,076,066	2,076,066
Clustered	13.17	248,901	345,040	473,536	675,131	1,040,071
Poliovirus (369,819 Δ's)						
Scattered	1.09	1,927,848	2,020,538	2,063,031	2,063,031	2,063,031
Clustered	9.01	382,855	552,002	703,372	904,424	1,208,796

Figure 9.6: Simulated performance in visible triangles per second for system with 36 renderers and various amounts of buffering.

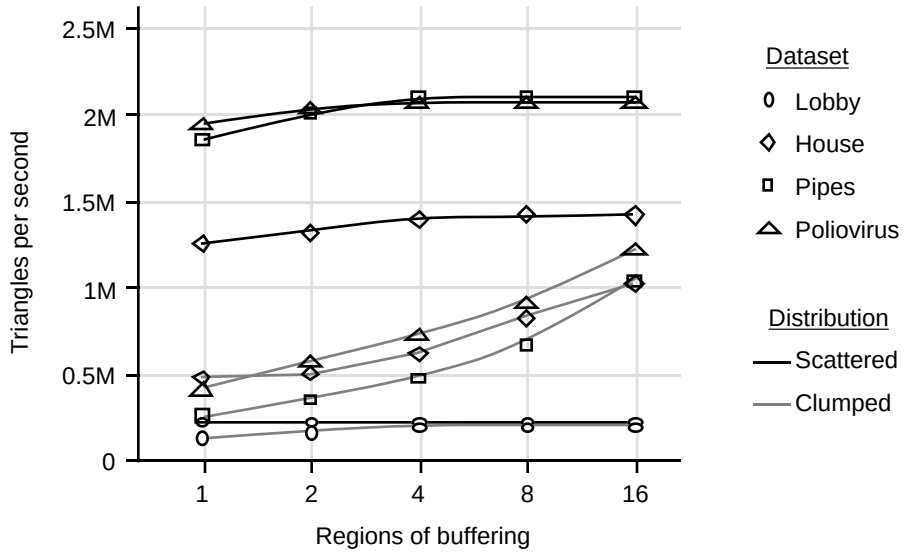


Figure 9.7: Plot of simulated performance (in visible triangles per second) vs. number of regions of buffering for a 36-renderer system.

Figure 9.6 gives the simulated rendering performance for four sample databases on a 36-renderer system with different amounts of buffering in the EMCs. The images, in each case, are Phong-shaded, unantialiased, and are computed at 1280x1024-pixel resolution. Rendering performance is given in visible

triangles per second. Figure 9.6 plots the same information.

We see from the graph that the amount of buffering makes almost no difference for databases with low dynamic load-balance ratios. It makes a substantial difference, however, for datasets with high dynamic load-balance ratios. The performance degradation is severe with clumpy datasets, even if a large amount of buffering is available. Eight regions of buffering appears to be a good choice, but is very expensive in terms of silicon area in the EMC. Four regions of buffering is the most that can conveniently fit in the new EMC. This means that system performance will suffer in distributed, immediate-mode applications or applications that distribute primitives poorly.

9.3 ESTIMATED PERFORMANCE

The performance of a deterministic system can be considered to be a vector function with many inputs and many outputs. The function's outputs are performance parameters that we care to measure, such as frame rate, latency, the number of triangles per second, etc. The function's inputs are system configuration and database parameters, such as the number of renderers, the number of shaders, the amount of buffering between rasterizers and compositors, the primitive-distribution method, etc.

Unfortunately, the performance function for the PixelFlow prototype design is too complex to analyze completely. We can estimate the function for particular inputs and outputs, however, using simulation. Section 9.3.1 describes the simulation method and simulators we used. Section 9.3.2 investigates the system's scalability as the number of renderers is increased. Section 9.3.3 estimates the system's performance for a range of datasets and a range of system parameters.

9.3.1 Simulation Approach

Two simulators were used to model the prototype system: a *rendering simulator* to develop rendering algorithms and to gather statistics for the sample databases, and a *timing simulator*, to model data-dependent system timing.

Rendering simulator. The rendering simulator is a general-purpose renderer that was used to develop and evaluate rendering algorithms, and to gather statistics for the sample databases. The simulator is, in essence, a scan-line renderer that can compute visibility and shading using a variety of algorithms. It is also instrumented to gather a variety of scene and rendering-algorithm statistics.

The simulator is written in C and accepts input files in Pixel-Planes 5 *archive* format, the scene-description format used in Pixel-Planes 5 [ELLS90b]. *Archive* format supports a variety of primitive types and allows one to specify a variety of rendering methods. This allowed us to setup and preview databases on Pixel-Planes 5, and to verify the correctness of our rendering algorithms. The simulator was used to render the sample images in Appendix A.

A scan-line renderer was used because of its flexibility. The heart of the simulator is a set of routines that read in a scene-description file, transform the primitives into screen coordinates, and build the sorted-polygon lists required for scan-line rendering. The simulator traverses pixels in order and, for each pixel, computes a list of primitives that are potentially visible at that pixel. To provide for antialiasing algorithms with wide filter kernels, this list includes primitives that lie up to two pixel diameters from the pixel center. The primitives can be sampled anywhere in a 4-pixel x 4-pixel square centered at the pixel center, providing a simple, efficient interface for a variety of visibility and antialiasing algorithms.

The rendering simulator supports the following rendering options:

- Antialiasing method: supersampling or A-buffer.
- Antialiasing kernel and number of samples per pixel.
- Database distribution method: scattering or clustering.
- Partitions: total number and which one to render.
- Output image dimensions.
- Screen region dimensions (to calculate statistics for the prototype system).

It is also profusely instrumented to gather image and rendering statistics, such as the average depth complexity at each pixel, the number of complex primitives, the bin-replication factor, static and dynamic load-balance ratios, etc.

The rendering simulator also can calculate and print the distribution of primitives falling into each screen region. This is used as input to the timing simulator, described next.

Timing simulator. The timing simulator models the time-dependent behavior of the prototype system. It is written in C++ in an object-oriented fashion. Classes were defined for each significant module in the system: GPs, IGC FIFOs, rasterizers, and compositors. The input and output parameters for each class correspond to the module's inputs and outputs. Each class contains a state machine that implements the rendering recipe for the system module and takes into account the module's time-dependent behavior. The modules are connected together when the simulator is initialized; the inputs of one module are connected to the corresponding outputs of other modules. Each module has a routine called *tick*, which advances time within the module one 25-nsec clock tick.

For example, the *Rasterizer* class simulates the rasterizer portion of the renderer board. Elements of class *Rasterizer* contain interfaces to two elements of class *IGCFifo* (one for the RFIFO and one for the TFIFO) and an element of class *Compositor*. Internally, *Rasterizer* elements contain a state machine whose transitions are determined by the inputs from connected modules and a performance model for the rasterizer. It accepts primitives from the IGC FIFOs at the rate that primitives are rasterized in an actual system. It also models the synchronization handshaking with the composition network, the TFIFO and RFIFO.

The simulator can be configured to simulate systems with different numbers of renderers and shaders, different amounts of buffering between components, different region-processing orders, and different transformation, rasterization, and composition speeds. The system accepts primitive-distribution information calculated in the rendering simulator. It does not compute actual images; it only models the time required to compute the image. The performance predicted by the timing simulator was cross-checked with performance estimates calculated by hand for several simple input configurations. The results agreed to within reasonable tolerances. The following sections give performance results obtained from the two simulators.

9.3.2 Scalability

To verify that the system's performance indeed scales linearly with the number of renderers over a large performance range, we ran the simulator on the Lobby, House, Pipes, and Poliovirus datasets with system configurations of 1, 4, 16, 64, and 256 renderers. Figure 9.8 gives the simulation results in visible triangles per second for a Gouraud-shaded, 1280x1024 image of each of these datasets. A 256-renderer system renders over 15 million triangles per second on the Poliovirus database.

Figure 9.9 plots the same information on a log/log scale. The rendering performance on each dataset initially rises linearly as renderers are added, then reaches a plateau as the image-composition network saturates. For system performance to continue to increase linearly, the dataset must be large enough that the rasterization time dominates the composition time—even for very large system configurations. Poliovirus is one such dataset. The Lobby dataset results in lower performance than the other three because its polygons are larger, resulting in more bin replication (the bin-replication factor is 2.21 for the Lobby dataset).

Dataset	Number of Renderers				
	1	4	16	64	256
Lobby (4,786 Δ's)					
Gouraud	37,744	141,472	346,310	426,180	448,968
Phong	30,850	110,864	198,425	228,816	237,678
House (51,922 Δ's)					
Gouraud	67,249	267,237	1,005,967	2,882,314	4,542,210
Phong	55,377	218,300	771,432	1,862,806	2,462,159
Pipes (137,747 Δ's)					
Gouraud	73,473	292,711	1,160,561	4,464,057	10,158,333
Phong	60,553	240,816	948,148	3,533,787	5,819,476
Poliovirus (369,819 Δ's)					
Gouraud	71,687	286,129	1,133,093	4,388,501	15,623,954
Phong	59,104	235,743	931,065	3,547,084	11,166,033

Figure 9.8: Simulation results (in visible triangles per second) for systems of various sizes rendering Gouraud-shaded, 1280x1024 images.

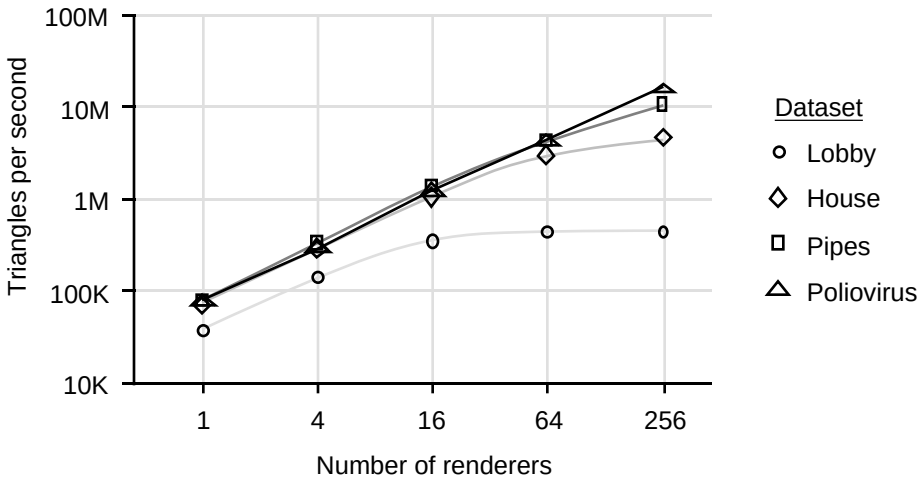


Figure 9.9: Rendering performance (visible triangles per second) vs. number of renderers (log/log scale).

We can determine how close the system is to being linearly scalable by solving for ß (the scale exponent) in the following expression:

$$perf_{N \text{ renderers}} = N^{\beta} \cdot perf_{1 \text{ renderer}}$$

For the Poliovirus dataset, over a range of 1 to 256 renderers, ß = 0.971, which is very near to the ideal value of 1. These results show that, for sufficiently large datasets, the system's performance scales approximately linearly with the number of renderers up to very high performance values—much higher than can be achieved on any existing or, to the author's knowledge, proposed system.

9.3.3 Results for Sample Datasets

Simulations were run for all the sample datasets using a variety of system configuration and image-quality parameters. Figures 9.10 through 9.13 show the results. All of the simulations assume a two card-cage system with 36 renderers and 4 shaders. Simulations were performed for low-resolution (640x512) and high-resolution (1280x1024) images and Gouraud- and Phong-shaded primitives.

Dataset	Visible triangles	1 sample		5 samples	
		Tris/sec	Frame rate	Tris/sec	Frame rate
Space station and shuttle	6,566				
Scattered		1,447,212	220.4	565,060	86.1
Clustered		918,707	139.9	255,407	38.9
Poliovirus	370,158				
Scattered		2,684,561	7.3	925,259	2.5
Clustered		1,403,975	3.8	338,266	0.9
Radiosity lobby	4,786				
Scattered		1,057,212	220.9	416,355	87.0
Clustered		672,191	140.4	215,401	45.0
House	51,937				
Scattered		2,389,102	46.0	836,445	16.1
Clustered		601,218	25.2	361,533	7.0
Earth	128,880				
Scattered		2,586,394	20.1	899,498	7.0
Clustered		1,596,630	12.4	562,574	4.4
Pipes	137,905				
Scattered		2,699,256	19.6	937,683	6.8
Clustered		1,231,845	8.9	313,937	2.3

Figure 9.10: Simulated rendering performance for 640x512-pixel, Gouraud-shaded images on a 36-renderer system with 4 shaders.

Dataset	Visible triangles	1 sample		5 samples	
		Tris/sec	Frame rate	Tris/sec	Frame rate
Space station and shuttle	6,566				
Scattered		847,882	129.1	306,693	46.7
Clustered		641,838	97.8	186,911	28.5
Poliovirus	370,158				
Scattered		2,201,343	5.9	718,534	1.9
Clustered		1,102,606	3.0	261,564	0.7
Radiosity lobby	4,786				
Scattered		611,919	127.9	222,864	46.6
Clustered		464,886	97.1	156,589	32.7
House	51,937				
Scattered		1,885,832	36.3	640,747	12.3
Clustered		1,014,122	19.5	277,551	5.3
Earth	128,880				
Scattered		2,097,315	16.3	696,573	5.4
Clustered		1,268,629	9.8	332,911	2.6
Pipes	137,905				
Scattered		2,192,448	15.9	726,198	5.3
Clustered		965,518	7.0	241,261	1.7

Figure 9.11: Simulated rendering performance for 640x512-pixel, Phong-shaded images on a 36-renderer system with 4 shaders.

Dataset	Visible triangles	1 sample		5 samples	
		Tris/sec	Frame rate	Tris/sec	Frame rate
Space station and shuttle	6,549				
Scattered		590,106	90.1	170,545	26.0
Clustered		420,293	64.2	104,288	15.9
Poliovirus	369,819				
Scattered		2,521,608	6.8	851,627	2.3
Clustered		907,264	2.5	189,021	0.5
Radiosity lobby	4,786				
Scattered		402,523	84.1	112,114	25.5
Clustered		284,715	59.5	85,580	17.9
House	51,922				
Scattered		1,986,486	38.2	661,366	12.7
Clustered		801,596	15.4	188,895	3.6
Earth	128,880				
Scattered		2,482,759	19.3	813,226	6.3
Clustered		970,044	7.5	219,894	1.7
Pipes	137,747				
Scattered		2,569,427	18.7	870,659	6.3
Clustered		614,832	4.5	124,776	0.9

Figure 9.12: Simulated rendering performance for 1280x1024-pixel, Gouraud-shaded images on a 36-renderer system with 4 shaders.

Dataset	Visible triangles	1 sample		5 samples	
		Tris/sec	Frame rate	Tris/sec	Frame rate
Space station and shuttle	6,549				
Scattered		315,174	48.1	87,156	13.3
Clustered		278,515	42.5	72,921	11.1
Poliovirus	369,819				
Scattered		2,063,031	5.6	658,310	1.8
Clustered		703,372	1.9	145,309	0.4
Radiosity lobby	4,786				
Scattered		220,180	46.0	62,924	13.1
Clustered		181,459	37.0	54,395	11.4
House	51,922				
Scattered		1,387,704	26.7	442,399	8.5
Clustered		605,949	11.7	143,969	2.8
Earth	128,880				
Scattered		1,996,592	15.5	622.849	4.8
Clustered		751,531	5.8	168,925	1.3
Pipes	137,747				
Scattered		2,074,191	15.1	669,780	4.9
Clustered		473,536	3.4	95,697	0.7

Figure 9.13: Simulated rendering performance for 1280x1024-pixel, Phong-shaded images on a 36-renderer system with 4 shaders.

The simulations measured the time to compute a single frame in its entirety. This means that calculations could not be overlapped from one frame to the next. In an actual system rendering multiple frames, bin-classification can be partially overlapped with rasterization of the previous frame, increasing the frame rate

by 10 to 15%, but not changing the latency. Because of this factor, the frame rates and rendering speeds for these images are approximately 10–15% pessimistic for 1-sample-per-pixel Gouraud and Phong images and approximately 3–5% pessimistic for 5-sample-per-pixel images.

CHAPTER 10

CONCLUSION

We have shown that image-composition forms the basis of a rich, new class of high-performance image-generation architectures. Their simple programming model and property of linear scalability make them a good choice for extremely high-performance systems. They have the disadvantage of requiring a high-performance composition network that spans the entire system and places some constraints on the rendering method. If a fixed-format pixel representation is used, however, this bandwidth is independent of the scene complexity.

In Part II, we described a prototype system design, PixelFlow, that demonstrates that these properties can be achieved in a realizable system. The PixelFlow design uses technology available in 1991. Simulation results indicate that a two-card–cage PixelFlow system can render up to 2.5 million triangles per second and 870,000 antialiased triangles per second, and that a large-configuration PixelFlow system can render up to 15 million triangles per second.

Perhaps the most important result for graphics-system architects is that image-composition provides a clean way to decompose the image-generation problem into an arbitrary number of pieces. This provides a way to build systems of higher performance when existing approaches reach their maximum levels of usefulness.

10.1 FUTURE RESEARCH

There are many difficult and interesting issues related to image-composition architectures that remain to be solved.

First, the field of distributed graphics architectures and algorithms is still new. Work is needed to develop algorithms to partition datasets over parallel processors so that graphics applications have a clean and efficient data-manipulation model. This is needed for any distributed-graphics architecture—not just image-composition architectures.

Second, the relationship between distributed graphics architectures and distributed general-purpose processors needs to be investigated. A distributed, retained-mode display structure is sufficient for viewing static scenes, but image-composition systems could be used as immediate-mode graphics servers for general-purpose multicomputers. The image-composition network provides a way to tie renderers embedded in different parts of the system together. No sorting is needed to route primitives to the correct renderer. As application processors compute data to be visualized, they can send it the nearest renderer.

Third, there is the question of how best to support realistic rendering methods, such as texturing, transparency, and shadows. Texturing is probably best implemented as a deferred-shading process. Transparency is difficult because it requires sorting, rather than selection, in the image-composition network. The A-buffer approach seems a natural way to implement transparency, but it requires high-speed A-buffer renderers and a more complicated image-composition network. Shadows require global information and appear more problematic.

10.2 TOWARD IMPLEMENTATION

We undertook this research with two aims: first, to demonstrate that the potential advantages of image-composition architectures can be realized in an actual system, and second, if the results looked promising enough, to actually build the system. The constraints for a proof-of-concept design and an actual system design are slightly different. We had to restrict ourselves to proven technology and components in the prototype-system design to demonstrate that the design is feasible; in an actual-system design, we can use unproven techniques as long as we can make them work—if the system works, the design is feasible.

This section lists the enhancements we intend to incorporate into the system we plan to build. Although some are risky, they greatly increase the power and usefulness of the system.

- **1024 bits per pixel.** Don Speck at Caltech recently demonstrated that one-transistor dynamic-memory cells can be implemented on standard CMOS processes at very high densities [SPEC91]. A 1-T DRAM cell allows us to place 1024 or more bits per pixel on the new EMC. This will provide 8 regions of buffering on rasterizers, improving rendering efficiency for datasets with high dynamic-load-balance ratios, and will provide the extra pixel memory required for complex lighting, shading, and texturing in the shaders.
- **Faster EMCs with 4-bit ALUs and linear-expression tree.** Using a newer CMOS process we can increase the system clock rate from 40 MHz to 50 MHz. The higher density allows more complex ALUs. We intend to replace the 1-bit ALUs and linear-expression tree of our previous EMC designs with 4-bit ALUs and a 4-bit linear-expression tree. The linear-expression tree generalizes to multiple bits in a straightforward way. A 4-bit ALU increases EMC performance by nearly a factor of 4.
- **Compositor within EMC.** To reduce board area and power consumption, we plan to incorporate the compositor function into the EMC itself. This requires a relatively small amount of logic, but requires extremely high-speed I/O pads. Tom Knight developed an impedance-matched I/O pad design that operates at extremely high speeds using reduced using signal levels [KNIG88]. Using this approach, the EMCs should be able to transmit pixel data between boards at 100 MHz. 1-volt switching levels and series termination reduce power dissipation and power and ground noise.
- **Faster, wider image-composition network.** Incorporating the compositor into the EMC allows us to double the number of composition network I/O wires for each EMC. We will reduce the number of EMCs per renderer/shader board from 80 to 64, but will double the number of composition network connections 2 in/out to 4 in/out for each EMC. This increases the composition network bandwidth to 3.2 Gbyte/second, allowing real-time frame rates for antialiased, high-resolution, Phong-shaded images.
- **Twin i860XP graphics processor.** To keep up with the higher-performance rasterizer, the graphics processor will be enhanced to a two-processor design using Intel's follow-on to the i860, the *i860XP* [INTE91]. This 64-bit microprocessor has a similar architecture to the i860, but has double-sized caches, a double-speed data bus, and a higher (50 MHz) clock speed. One of the i860XPs will be used to run the application program (in immediate mode) or will maintain the display structure (in retained-mode). The other will perform geometry calculations and feed primitives to the rasterizer.
- **Coefficient adjuster at IGC input.** Since the new EMC will be substantially faster than the EMCs in previous designs, and general-purpose floating-point processors are not keeping pace, we face a performance imbalance between the geometry-processing and rasterization portions of the renderer. We can take advantage of the extra rasterizer power by computing additional samples per pixel. A coefficient adjuster will be incorporated into the IGC to adjust C coefficients for x and y screen-space offsets required for supersampling. The i860s will only have to calculate the IGC instructions for a single sample point (at the center of the pixel). The IGC input structure will read the data multiple times, adjusting C coefficients as needed for additional samples.
- **Support for image-based textures.** An additional texture-board design will allow MIP-map textures to be computed at the full image-composition network bandwidth. The texture board should also be able to perform shading calculations, so the texture board can replace the shader in the prototype system.

These enhancements will multiply the performance of the renderer/shader board by approximately a factor of 5. If all goes well, we estimate that a two-card–cage, enhanced PixelFlow system will render 2 million 5-sample-per-pixel antialiased, Phong-shaded, MIP-map textured, 100-pixel triangles per second at 30 Hz frame rates at 1280x1024 screen resolution.