Implementing PixelFlow Shading

Marc Olano

In the previous sections of this chapter, we covered the interface seen by both application and shader writers. In this section, we cover the basic knowledge of the PixelFlow hardware required to understand the implementation issues. For more details on the PixelFlow architecture, see [Molnar91][Molnar92][Eyles97]. We also cover some intermediate levels of abstraction between PixelFlow and an abstract graphics pipeline and explain how our procedural stages fit into the real PixelFlow pipeline.

Our abstract pipeline consists of procedures for each stage in the rendering process. Since these can be programmed completely independently, it is possible (and expected) that a particular hardware implementation may not have procedural interfaces for all stages. While PixelFlow is theoretically capable of programmability at every stage of the abstract pipeline, our implementation only provided high-level language support for surface shading, lighting, and primitives. The underlying PixelFlow software includes provisions for programmable testbed-style atmospheric and image warping functions, but we did not supply any special-purpose language support for these.

1. High-level view

PixelFlow consists of a host workstation, a number of rendering nodes, a number of shading nodes, and a frame buffer node. The hardware and lower level software handle the scheduling and task assignment between the nodes, so we can consider the flow of data in the system as the pipeline shown in Figure 1. This view is based on the passage of a single displayed pixel through the system. Neighboring pixels may have been operated on by different physical nodes at each stage of this simplified pipeline. This will be covered in more detail later in this chapter. For the purposes of mapping the abstract pipeline onto PixelFlow, the simplified view of the physical PixelFlow pipeline is sufficient.

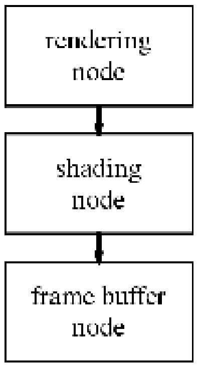

Figure 1. Simplified view of the PixelFlow system

1.1. Applying the abstract pipeline

The mapping of an abstract pipeline onto PixelFlow is shown in Figure 2. This abstract pipeline is divided into stages based on a set of logical rendering tasks. Contrast this with the abstract model presented later in Chapter 8, in which a single shader spans several computational units.

The modeling, transformation, primitive, and interpolation stages are handled by the rendering node. The shading, lighting, and atmospheric stages are handled by the shading node. Finally, the image warping stage is handled by the frame buffer node.

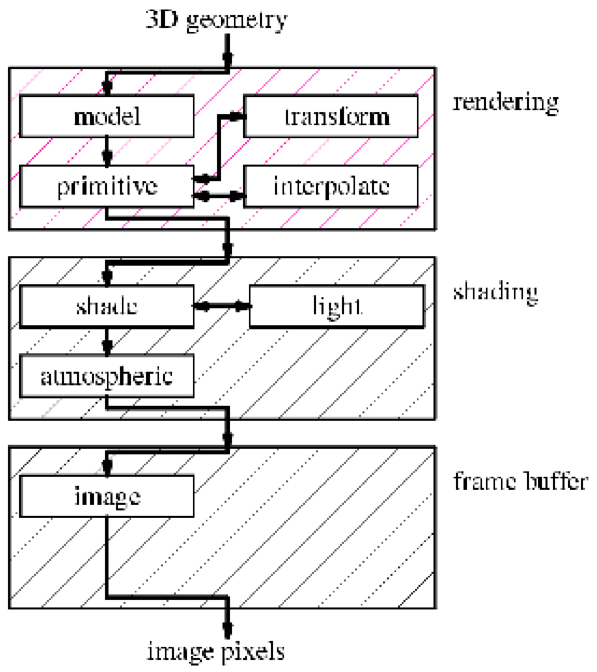

Figure 2. Procedure pipeline.

When mapping the abstract pipeline onto PixelFlow, we maintain the interfaces to the pipeline stages. Thus, the procedures written for PixelFlow should look exactly the same as the procedures written for a different machine with a different organization. The code for each stage is written just as if it were part of some arbitrary rendering system implementing the abstract pipeline.

It is important to notice that the abstract pipeline only provides a conceptual view for programming the stages. It allows the procedure programmer to pretend that the machine is just a simple pipeline instead of a large multicomputer. The real stages do not need to be executed strictly in the order given (and, in fact, are not). The user writing code for one of the stages does not need to know the differences between the execution order given in the abstract pipeline and the true execution order. The mapping of the abstract pipeline onto PixelFlow exhibits several different forms of this.

The first example is the overall organization of the processes on PixelFlow. PixelFlow completes all of the modeling, transformation, primitives, and interpolation in the rendering nodes before sending the shading parameters for the visible pixels on to a shading node. PixelFlow then completes all of the shading, lighting, and atmospheric effects before sending the completed pixels on to the frame buffer node for warping. On a different graphics architecture, it might make more sense to complete all of the stages for every pixel in a primitive before moving on to the next primitive. Either choice appears the same to users who write the procedures. The abstract pipeline does not include information about the stage scheduling to allow just such implementation flexibility.

The procedures running on the PixelFlow rendering nodes provide another example. The abstract pipeline presents transformation, primitive, and interpolation as if they were a sequential chain of processes. On PixelFlow, the primitive stage drives transformation and interpolation. A procedural primitive function is invoked for each primitive to be rendered. This function calls both transformation and interpolation functions on demand as needed. The results stored for each pixel include its depth, an identifier for which procedural shader to use and the shading parameters for that procedural shader. Once again, the user writes procedures as if they were independent sequential stages and is not aware of the true ordering within the PixelFlow implementation.

The final example is with the shading and lighting stages. The abstract pipeline presents shading and lighting as if the shading stage called the lighting stage for each light. On PixelFlow, the linkage between these stages is not as direct. These two stages run with an interleaved execution scheduled by the PixelFlow software system. This interleaving is explained in more detail in [Olano98]. And again, the interleaved scheduling is hidden from anyone who writes a shading or lighting procedure.

1.2. Parameter manager

Supporting this pipeline is a software framework that handles the details of the rendering process and the communication between the programmable procedures. That communication is assisted by a global parameter manager, implemented on PixelFlow by Rich Holloway. The parameter manager allows each node in the system to find values or pixel memory addresses of the parameters. It also keeps track of other attributes of each parameter - its type and size, default values, whether it needs to be transformed (and how), etc. Whenever a procedure is compiled, an extra *load function* is generated. This load function is run when the procedure is loaded by the application. The load function registers all of the parameters used or produced by the procedure. The parameter manager collects this information and makes sure each parameter is available when it is needed. This global parameter space is similar to the shared memory "blackboard" idea used by MENV [Reeves90].

2. Low-level view

The PixelFlow system data-flow was show in Figure 1. A view of the hardware at that level was sufficient to understand how the abstract pipeline maps onto PixelFlow. We must delve deeper to understand some of the issues that impacted our implementation. Where Figure 1 showed only a single stage for rendering and shading, PixelFlow may have many nodes (see Figure 3). There are also two networks connecting the nodes in the PixelFlow system, the geometry network and composition network. The rendering nodes and shading nodes are identical, so the balance between rendering performance and shading performance can be decided on an application by application basis. The frame buffer node is also the same, though it includes an additional *daughter card* to produce video output.

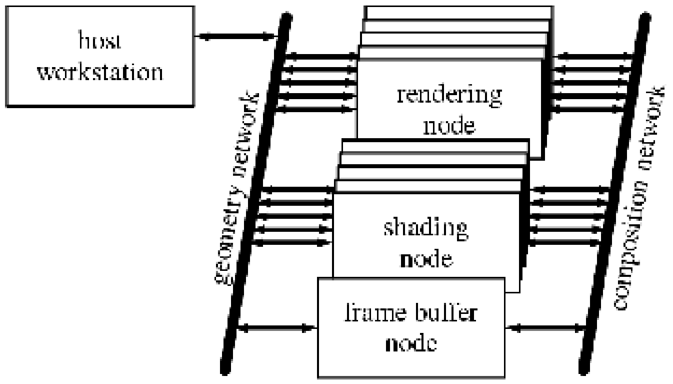

Figure 3. PixelFlow machine organization.

Each rendering node is responsible for rasterizing an effectively randomly chosen subset of the primitives in the scene. The rendering nodes work on one 128x64 pixel *region* at a time (or 128x64 image samples when antialiasing). Many of our examples and tests are based on either an NTSC video screen size of 640x512 pixels with four samples per pixel, or a high-resolution screen size of 1280x1024 pixels. There are 40 regions in an NTSC image with no antialiasing. With antialiasing using four samples per pixel, the NTSC image has 160 regions. Without antialiasing, the high-resolution image also has 160 regions. Therefore, our target is to be able to handle 160-128x64 regions at NTSC video rates of 30 frames per second.

Since each rendering node has only a subset of the primitives, a region rendered by one node will have holes and missing polygons. The different versions of the region are merged using *image composition*. PixelFlow includes a special high-bandwidth network called the *composition network* with hardware support for these comparisons. As all of the rendering nodes simultaneously transmit their data for a region, the network hardware on each node compares, pixel-by-pixel, the data it is transmitting with the data coming in from the upstream nodes. It keeps only the closest of each pair of pixels to send downstream. By the time all of the pixels reach their destination, one of the shading nodes, the composition is complete.

Once a shading node has received the data, it does the surface shading for the entire region. The technique of shading after the pixel visibility has been determined is called *deferred shading* [Deering88][Ellsworth91]. Deferred shading only spends time shading the pixels that are actually visible, and allows us to do shading computations for many more pixels in parallel. With non-deferred shading, each primitive is shaded separately. With deferred shading, all primitives in a region that use the same procedural shader can be shaded at the same time.

In a PixelFlow system with `n` shading nodes, each shades every n^{th} region. Once each region has been shaded, it is sent over the composition network (without compositing) to the frame buffer node, where the regions are collected and displayed.

3. PixelFlow node

The nodes on PixelFlow all look quite similar (See Figure 4). Each node of the PixelFlow system has two RISC processors (HP-PA 8000's), a 128x64 custom SIMD array of pixel processors, and a texture memory store. Only the rendering nodes make use of the second RISC processor, where the primitives assigned to the node are divided between the processors. The existence of the second RISC processor does not impact our implementation, so we can take the simplified view that there is only one processor on the node and let the lower level software handle the scheduling between the physical processors. The RISC processors share 128 MB of memory, while each pixel processor has access to 256 bytes of local memory. The texture memory exists in several replicated banks for access speed, but the apparent size is 64 MB.

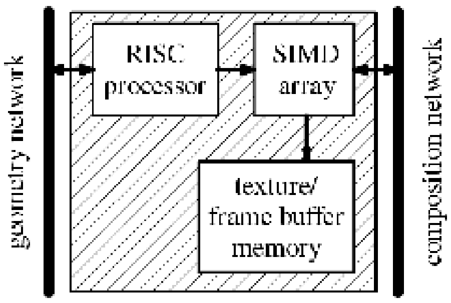

Figure 4. Simple block diagram of a PixelFlow node

Each node is connected to two communication networks. The geometry network, carries information about the scene geometry and other data bound for the RISC processors. This network is four bytes wide and operates at 200 MHz. It can simultaneously send data in both directions, giving a total bandwidth of 800 MB/s in each direction. The composition network handles transfers of pixel data from node to node. It also operates in simultaneously in both directions at 200 MHz. However, the composition network is 32 bytes wide, giving a bandwidth of 6.4 GB/s in each direction. Four bytes of every transfer is reserved for the pixel depth, reducing the effective bandwidth to 5.6 GB/s.

3.1. Compiler target

Every procedural stage on PixelFlow has a testbed-style interface, which allows new stage procedures to be created using the internal libraries of the PixelFlow system. Writing code new procedures using this interface requires a deep understanding of the implementation and operation of PixelFlow, more than will be provided in this dissertation. We provide a high-level, special-purpose language so the users who write new procedures will not need to have that level of understanding of PixelFlow. It also makes rapid prototyping and porting procedures to other systems possible.

The compiler for our special-purpose language produces C++ code that exactly conforms to the testbed interface. This code consists of two functions, a load function (mentioned in section 1.2), and the actual code for the procedure. The code for the procedure is run on the RISC processor and includes embedded *EMC functions*. Each EMC function puts one SIMD instruction into an instruction stream buffer. The EMC prefix that appears on all of these functions stands for *enhanced memory controller*, from the Pixel-Planes SIMD array's origin as a processor-enhanced memory; we use it here just to identify the functions that generate the SIMD instruction stream.

When the C++ code for a procedure is run, the result is a buffer full of instructions for the SIMD array. This instruction stream buffer can be sent to the SIMD array several times without requiring the original C++ code to be re-executed.

There are two forms of EMC function used in PixelFlow. The form used on the shading nodes checks the available space in the instruction stream buffer with each instruction and can re-allocate the buffer on the fly. The form used in the rendering nodes requires a buffer of sufficient size to be allocated at the beginning of the procedure. The reason for this difference, and the issues that result, are discussed in Section [Olano99].