7 CONCLUSION

The thesis stated in Chapter 1 was:

> The decomposition of the graphics pipeline into a coherent set of user-programmable procedures provides valuable new tools to the interactive graphics programmer. In addition, a special-purpose language for writing the procedures insulates the programmer from the details of the graphics system while providing the system designer the opportunity to perform optimizations. This can be implemented efficiently on a graphics machine using current technology to yield a system that maintains interactive frame rates

Chapter 2 presented our abstract pipeline, a decomposition of the graphics pipeline satisfying the requirements of the thesis. The stages of this pipeline are model, transform, primitive, interpolate, shade, light, atmosphere, and warp. These stages provide complete coverage of the traditional graphics pipeline. The stages are orthogonal, so changes in one do not require changes in another. The utility of each these stages has been shown by the prior work in procedural systems, surveyed in Chapter 2 using the abstract pipeline as a taxonomy.

PixelFlow allows user-written code for the primitive, interpolator, shading, lighting, atmospheric and image warp stages of the abstract pipeline (as well as several extra PixelFlow-specific stages not covered by this dissertation). New code for all of these stages can be added using a testbed-style interface, without requiring a copy of the rest of the PixelFlow libraries.

PixelFlow also provides a working demonstration of shading and lighting procedures, written in a high-level special-purpose shading language. The details were discussed in Chapter 4. Surface shaders and lights written in this language, pfman, run in real-time.

Several people have written shading procedures in pfman. Their work and experiences were covered in Chapter 6. PixelFlow also provides a preliminary demonstration of the a special-purpose language for the primitive stage, discussed in Chapter 5.

Finally, PixelFlow provides support in the graphics API for user-written versions of the primitive, lighting and shading stages. These extensions were discussed in Chapters 4 and 5. With these extensions, applications can use all three stages, without regard to whether the code for the stage is built-in, written by a user using the low-level testbed interface, or written by a user in the pfman language.

The PixelFlow graphics system demonstrates that programmable stages can run in real-time on a graphics system built using current technology. PixelFlow is a large and complex system, the end result of years of labor by many talented people. The author's contribution to this system was the definition of the pfman language; the majority of the pfman compiler; and active participation in the design of the overall system organization, API extensions, and testbed interface.

7.1. Conclusions and future research

We have used PixelFlow to demonstrate programmability, at varying levels, in some stages of the abstract pipeline. PixelFlow is an immensely flexible machine, which could be capable of demonstrating the remaining procedural stages. There are unexplored avenues and further work that could be done in every stage. We summarize some of these possibilities below.

7.1.1. Models

Several procedural modeling languages have been proposed (see the prior work discussed in Chapter 2). In general, there is some question whether a procedural modeling language should be separate from the graphics API, and if so whether it should sit above or below the API.

On PixelFlow, a procedural modeling stage could run below the API on the node microprocessors. This would avoid the communication bottleneck between the host and Pix-

elFlow, and distribute the computation for the procedural models over multiple processors and multiple nodes.

7.1.2. Transformations

Procedural transformations have not been widely explored. Some of the issues have been addressed by [Fleischer88]. Others have been addressed by the work on deformations [Barr84][Sederberg86]. Open questions include how to specify a procedural transformation, how to control it from the graphics API, how it interacts with the normal transformations, and how to correctly render arbitrary primitives when their shape becomes distorted.

PixelFlow does not have procedural transformations, either through a high-level language or testbed interface. Procedural transformations were not a high priority, but PixelFlow should be capable of supporting them. While we have addressed the basics in our definition of a procedural transformation stage, many details of language interface and implementation are not covered by this dissertation. Since each primitive may require a number of transformations, code efficiency of transformations is even more critical than it was found to be for primitives in Chapter 5.

The standard interface for linear transformations uses a transformation matrix stack. The top of this stack is the current composite transformation matrix. The lower levels of the stack save earlier transformation states. The idea of keeping a stack of transformations is still applicable for procedural transformations, but the top of the stack can no longer be represented as a single composite matrix. Instead, it is represented by a chain of functions to be composed, one after the other.

Linear transformations and matrices remain important, since they easily represent the physical concepts of rigid motion and perspective projection. Any interactive machine supporting procedural transformations will need to preserve the efficiency of these common transformations. One way to achieve this would be to use a special representation for linear transformations. Back-to-back linear transformations could then be composed into a single matrix. A more general solution would allow combination of any back-to-back transformations that are closed under composition. Some analysis should be done before

this latter form is implemented – it is not clear that transformations of other types will appear back-to-back as often as the linear transformations.

7.1.3. Primitives/Interpolators

We have a full implementation allowing testbed procedures to be created for the primitive stage. We also have a preliminary implementation of a high-level language for creating primitives and interpolators. Several problems that prevent the current high-level language interface from being more useful and usable were discussed in Section 5.5. This section also covered some of the solutions that could be pursued.

Given the problems of creating a high-level language interface for primitives, it appears the state of the art in graphics hardware, as embodied by the PixelFlow machine, is not yet mature enough to support a full-blown special-purpose language for primitives. Primitives could be made more efficient through more aggressive optimization in the compiler and research in domain-specific compiler techniques. Advances in the state of the art of graphics hardware systems, particularly in speed and available memory, will also ease the current resource limitations.

7.1.4. Shading/Lighting

We have a fairly complete implementation of a language for writing new procedures for surfaces and lights. For better portability, this language should have been as close as possible to the RenderMan shading language. If the language were a superset of RenderMan (i.e. with the addition of a fixed point type), it would be capable of running RenderMan shaders unchanged.

Texture transformation is a second useful feature present in pfman, but missing in RenderMan. Texture transformations are present in OpenGL, and have proved useful for a variety of texturing techniques [Segal92]. It would be interesting to generalize the use of different transformations beyond the geometric transformations and a texture transform. For example, the OpenGL API already can switch between three transformation stacks [Neider93]. It could easily be extended to handle an arbitrary number. RenderMan also

has the ability to name specific transformation states [Upstill90]; this feature could also be used to pick the type of transformation.

Within the framework of the pfman language, the main feature that is missing is the ability to compute derivatives. RenderMan has an operator that can be used to take the derivative of any value or expression that varies over the surface. The RenderMan renderer finds the derivative of an expression by first evaluating the expression across the surface. An approximation of the derivative is computed using the results of the expression in adjacent surface samples.

Other hardware implementations may be able to use this approach, but the results from adjacent samples are not available on PixelFlow. Since PixelFlow does not run the shading procedures until after all surfaces have been rendered, there is no guarantee that any of the adjacent pixels will be part of the same surface or use the same shader. It is possible to interpolate the derivatives of the shading parameters at the same time as the parameters themselves. A possible way to allow general derivatives is to use a static compile-time analysis to compute the desired derivative from the derivatives of the shader's input parameters using the derivative chain rule. At each intermediate computation, derivative computations would be added. The downfalls of this method are that it may add a considerable number of computations to the shader, and it may increase the memory use significantly.

There is some interesting possible future work extending some of our optimization techniques. In particular, we have barely scratched the surface of automatic combined execution of portions of different shaders. We do only the most basic of these optimizations automatically. Some others we do with hints from the shader-writer, whereas other possible optimizations are not done at all. For example, we currently run every shader instance independently. It would be relatively easy to identify and merge instances of the same shader function that do not differ in any uniform parameters. For a SIMD machine like ours, this would give linear speed improvement with the number of instances we can execute together. Even more interesting would be to use the techniques of [Guenter95] to combine instances of the same shader function with differing uniform parameter values.

7.1.5. Fog and atmospheric effects

The high-level language issues for procedural fog and other atmospheric effects have been well explored by the RenderMan language. PixelFlow supports a testbed interface for procedural atmospheric effects. Pfman atmospheric procedures could be added with relative ease, but there may be new opportunities for optimization along similar lines to the optimizations we do for surface and light procedures. The RenderMan API includes functions to specify the atmospheric procedure and its parameters. Similar extensions could be added to the OpenGL API.

7.1.6. Image warping

PixelFlow has a testbed interface for new procedural code to run on the frame buffer board. However, there is only a single procedural hook to handle both image warping and the operations to load pixels into the frame buffer itself. This latter code is non-trivial and make new versions of this stage difficult to write. For usable support of procedural image warping, this stage should be split into a procedural warping stage and a separate frame buffer stage, hidden from the user.

There are several more general issues for procedural image warping that merit further investigation. High-level implementation-independent language support for image warping has not been well explored. The Adobe Photoshop plugin interface uses C code and a Photoshop-specific library [Knoll90a]. Similarly, the GIMP (Gnu Image Processing Manipulation Program) plugin interface also uses C code and a GIMP-specific library. Even the GIMP scheme scripting language relies on GIMP-specific functions.

All of the existing work for procedural image warping is in the context of image manipulation software. The combination of image warping with a 3D renderer is an open question.

7.1.7. Other areas

Chapter 2 included examples of what has been done in each stage. With a machine that supports several stages of the abstract pipeline, new techniques can use cooperating procedures for several stages. For example, frameless rendering is a technique that updates

pixels in a random order [Bishop94]. The machine could render a pixel or region of pixels at a time. A primitive procedure could shuffle the rendered pixel locations to random places in the larger final image, and an image warping procedure could reverse the shuffle to place the pixels in their correct locations in the output image.

The abstract pipeline presented in this dissertation was created for Z-buffered interactive graphics systems, a common choice for high-end graphics machines. We have not attempted to address other rendering methods. For example, Arie Kaufman and his colleagues at SUNY Stony Brook have developed a series of flexible volume rendering architectures [Kaufman88][Kanus96]. There has also been work on flexible ray-tracing machines [Potmesil89]. Some polygon rendering machines have used non-Z-buffer algorithms, for example [Niimi84] used a scan-line algorithm. Each of these systems may motivate an abstract pipeline different from the one we have created. Furthermore, some stages may be shared between these different pipelines, while other stages may be specific to one pipeline or another.

7.2. Contributions

In the process of proving the thesis, this dissertation introduces a number of new contributions to the field of computer graphics. The major contributions are listed below:

- A new decomposition of the rendering process into an abstract pipeline. This pipeline consists of an orthogonal set of procedural stages. A graphics system can allow user-programmable procedures at any or all of these stages.
- A demonstration that such a system can be implemented efficiently enough to run in real-time on interactive graphics hardware built with current technology.
- A demonstration that such a system can use a high-level application-specific language for the definition of the procedures and still achieve real-time performance. Such a special-purpose language allows a user to write new procedures without knowing internal details of the graphics system. A special-purpose language also allows procedures written for one graphics system to run, without modification, on a completely

different graphics system supporting the same language – this has been shown by [Upstill90][Slusallek94][Gritz96], but is not demonstrated as part of this dissertation.

- The definition of a special-purpose language, pfman, based on Pixar's RenderMan shading language, for writing procedural stages for real-time graphics hardware
- An analysis of which features in the pfman language, but not in the RenderMan shading language, were necessary for the interactive implementation. This will be important for future implementations since it is our belief that the RenderMan language would be no more difficult to implement than the pfman language. Our experience and the comments of our users indicate that compatibility with the RenderMan is preferable to a new, incompatible, language.
- An analysis of the optimizations we used to allow pfman shaders to run in real-time.
- The definition of extensions to the pfman language for writing procedures for the primitive and interpolator stages.
- A preliminary demonstration of primitives written in pfman.