6 USER EXPERIENCES

We have had several users of the procedural shading capabilities of PixelFlow. This chapter will present some of their results and experiences.

6.1. Results

Our first and most tolerant user is Arthur Gregory. He began writing shaders well before the hardware or pfman compiler were complete. He did most of his development on the PixelFlow simulator. This simulator was an effective tool for early software development. It was also incredibly slow. Through his extraordinary patience, we were able to find and remove many bugs in the pfman compiler. Over the course of a year and a half, he wrote a number of pfman shaders, including two for an animated version of the bowling image on the cover of [Upstill90] (Figure 6.1).

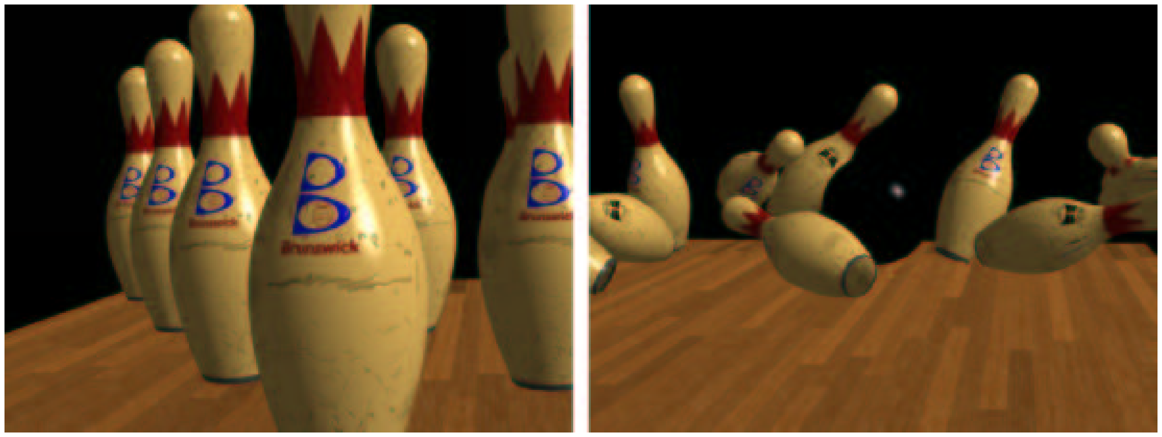

Figure 6.1. Bowling images by Arthur Gregory.

Our second major set of users are members of the UNC nanoManipulator Project. The nanoManipulator project, in collaboration with physicists from UNC, provides a virtual reality interface for an atomic force microscope. This microscope can resolve surface details down to individual atoms and, with interactive control, allows the physicist to make nanometer-scale modifications to the surface. In addition to the surface height measured by the microscope, the physicists would like to see other measured attributes like friction and surface adhesion. The UNC nanoManipulator Project has been experimenting with procedural shaders to show these extra surface attributes.

Chun-Fa Chang wrote the initial nanoManipulator shaders. His shaders show surface attributes using a checkerboard pattern, texture map, or bump map. All three are modulated by per-vertex shader parameters. An example of the checker board shader is shown in Figure 6.2a. Sang-Uok Kum continued this work with shaders that replace the per-vertex shader parameters with a texture map.

Renee Maheshwari created another shader using a *spot noise* function [vanWijk91]. Spot noise places spots of varying shapes on the surface in some random but even distribution (a Poisson or pseudo-Poisson distribution). She is also working on a line-integral convolution shader [Cabral93]. Alexandra Bokinsky created a shader using a bi-directional reflectance density function (BRDF) instead of the more common, but less accurate shading approximations. Aron Helser is combining these to create a large shader that will use multiple shading effects to visualize several surface attributes at the same time.

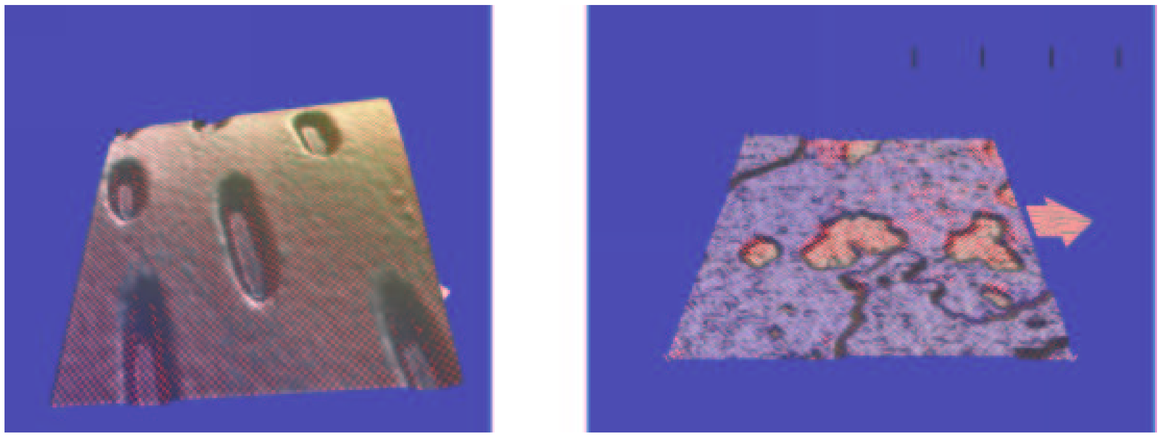

Figure 6.2. nanoManipulator shaders.

a) Atomic force microscope scan of a CD-ROM, checker pattern fades in and out with surface adhesion. b) Surface with three layers, bottom layer has a different adhesion and friction.

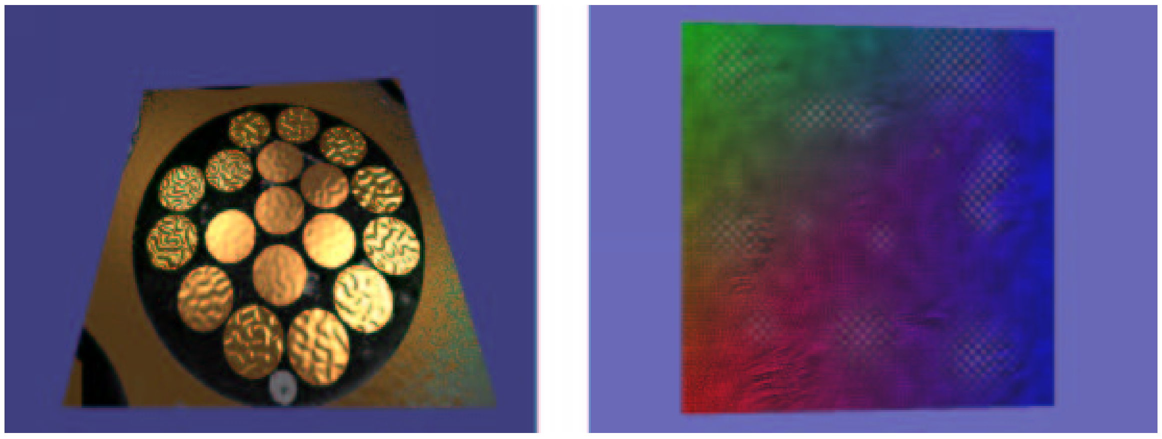

Figure 6.3. More nanoManipulator shaders.

a) Scanning electron microscope image of a set of copper-gold alloy samples. Each is rendered using an appropriate BRDF. b) Combined shader using color, fading checkerboard, and spot noise.

Our newest user is Chris Wynn. All of our other users have used only one or two shaders at a time. He will be applying procedural shaders to a larger architectural model.

6.2. Lessons

We were fortunate to have these users to exercise the PixelFlow shading system. Their needs and problems helped to focus our efforts in developing and extending pfman. In this section, we look at some of the lessons we learned from our users. We were able to act on some of these, while others will have to stand as guides for future systems.

6.2.1. Early users

Having a friendly user of the system early in the development is enormously helpful. Arthur Gregory wrote and tested shaders using the PixelFlow simulator. He wrote complex shaders that stretched the capabilities of the early system. He was forced to contort his code to use less memory before the pfman memory allocation was done, proving that automatic memory allocation was going to be a necessity. He did array manipulations element-by-element before either array operations or non-constant array indices were working. Without his efforts, pfman would have probably remained as difficult to use as the testbed interface it replaces.

6.2.2. Research project

While Arthur Gregory did help us to focus our pfman development, it is still a research project. This fact is manifest by missing features, like the derivative operators or a uniform fixed-point type. Our users have also learned to consider the possibility that problems they experience are the result of undiscovered compiler bugs instead of bugs in their own shader code. We also lack the documentation of a more polished commercial product. Documentation for pfman consists of [Upstill90] for information on RenderMan, a pfman language definition document, and a handful of network news postings to a local news-group highlighting the differences between RenderMan and pfman. The API on PixelFlow is documented by the OpenGL reference manual [Neider93] and a technical report on the extensions and differences on PixelFlow [Leech98]. To fill in the absence, our users have started to collect and share their own experiences. Using this information, Chris Wynn, the user who has most recently started to use pfman, was able to write and run a simple procedural checker board shader in half an hour.

6.2.3. Machine details

One of the key advantages of writing shaders with a special purpose language is that the language hides details of the machine architecture. This does not always work. Aron Helser of the nanoManipulator project said, “Pfman successfully hides nearly all of the details of programming the EMCs. But we still have to be acutely aware of the machine and its strengths and limitations.”

Despite our memory allocation efforts, the nanoManipulator users have run into pixel memory limitations, and were forced to change from floating point to fixed point for many computations. They have hit memory limits on the number of times a loop can execute. Each time through the loop creates another copy of the SIMD instructions for the body of the loop, eventually creating too many. They have had to adapt their shader code to avoid using varying expressions to control the number of times a loop executes. Many of our users have found that if the correct set of options have not been set in the application, some quantities (like surface position) will not be computed even if they are required.

Some of these machine-specific details could be masked, or better masked, by the compiler. For example, the limitation on how many times a loop can execute would be reduced if the compiler moved loop-invariant code out of the loop, and the computation of surface position could be automatically forced whenever a shader needs it. Other problems are caused because the PixelFlow machine does not precisely fit the model presented by the shading language. For example, no matter how good the compiler is, the amount of pixel memory will never increase.

6.2.4. Similarity to RenderMan

Most of our users knew or learned RenderMan before trying pfman. In general, this was helpful since RenderMan and pfman are similar. Arrays are the most popular added feature, although two users also said that they liked the way that pfman made all of the parameters to a shader explicit. However, the universal opinion is that the parameters should have kept the same names as the RenderMan equivalents (e.g. `E` instead of `px_rc_eye`). These sentiments were well summarized by Chun-Fa Chang, who said, “It’s like learning Pascal helps learning C,” and, “I think the more pfman is like the RenderMan shading language, the better it is.”

6.2.5. Development cycle

Another common complaint is the amount of time required to compile and run a shader after each change. This is critical to the fast development and prototyping time traditionally associated with shading languages. The RenderMan shading language is compiled to an intermediate form which is interpreted during rendering [Hanrahan90].

For interactive performance on a machine using current technology, an interpreter is not practical. Fortunately, it is probably not necessary. In five trials compiling a directory of 20 shaders, the average time to compile an individual shader was 11 seconds. The shortest time was 8 seconds and the longest was 23 seconds. In comparison, linking these into executables to run on the PixelFlow nodes took about a minute and a half (the average time was 1:31 with times ranging from 1:26 to 1:35). The time to get the first frame after starting an application averaged ranged from 1:16 to 1:21 over five trials, with an average of one minute and 22 seconds.

Most of the time is spent linking and starting the PixelFlow machine, not compiling the shader. By pre-linking everything but the changing shaders and reducing the object size by stripping all unnecessary symbols, the linking time was reduced to an average of 53 seconds (from 48 to 60). Ideally, we would dynamically link shaders into a running PixelFlow application to avoid the overhead of both the linking and running. Our OpenGL extensions allow for dynamic linking and loading of shaders if we are later able to support it.

6.2.6. Learning curve

As the project and our users have progressed, learning to use shading on PixelFlow has become easier. The earliest experiences on the PixelFlow simulator were slow and painful. It took several weeks to a month to become proficient enough to generate a single frame from the simulator. Once we moved from simulation to hardware, the time to get started decreased significantly. Since the days of the simulator, we have also put some effort into allowing users to work without a full copy of the PixelFlow libraries. Now they need only their application and pfman shader source code.

The ramp-up time on PixelFlow has also improved as more users have gained experience. The old users help the new ones to avoid pitfalls of the machine, and have collected documentation for getting started and writing shaders. As a result, it can now take less than a day to get started, instead of a month or more.

Once started, writing and using new pfman shaders becomes much easier. In an informal poll of the current user community, most estimated that they could write equivalent shaders three times faster now than when they first started.

6.2.7. Application program interface

Most of the comments I have received from users deal with using the pfman language for writing shaders. This capability is what drew all of them to work on PixelFlow instead of a better-supported commercial machine. However, since the nanoManipulator project already had a working application using OpenGL, they have shown the advantages of using an API based on OpenGL. The same application now works on Silicon Graphics machines and on PixelFlow, though the added shading capabilities are only available on PixelFlow. As Chun-Fa Chang said, “It’s good that we have such a high-level language to take advantage of the programmable shading in PixelFlow and it still fits in the framework of OpenGL.”