Ignoring Hardware Differences

Marc Olano
University of Maryland, Baltimore County

One of the great promises of real-time shading is the potential to have a single shading program that can run across a wide range of graphics hardware. While we don't yet have a single cross-platform shading language to satisfy everyone, there is ample evidence that it *is* possible. In this chapter, we discuss what is necessary to create a cross-platform shading language, how shading languages allow us to ignore hardware differences, what range of hardware can reasonably be represented by a single shading language, and what evidence exists now that it will really work.

1. The key to cross-platform shading

The key to a cross-platform shading language is to work with a common model of shading hardware rather than specifics of the hardware itself. The model is a mental picture of what's going on that shader-writers use to make sense of the code they write. The further you get into hardware specifics, the less general your model becomes.

Designing a good model for shading is the balance of three competing goals. The model should be simple enough for novice users to understand. It should be a good model of the problem domain, accurately describing what the shader is trying to do rather than exactly how to do it. This will allow the shading language compiler to map the shader onto the hardware in the way that is best for the specific hardware platform. Of course, it should also be possible to map it efficiently onto all the desired platforms.

The second goal is particularly important — the purpose of shading code (or any code) is to describe **what** you want done. The compiler can and will made different choices about **how** (within limits — it can't change the algorithms you use, but it can rearrange the execution order, unroll loops, decide if a certain operation should be computed or looked up in a texture, etc.).

1.1. Single Program, Multiple Data

Shading is inherently a very parallel task. Whether we are talking about vertices in an object, a surface diced into micropolygons, ray-traced intersection points, or screen pixels, there is always some relatively common set of operations being applied to a set of samples on the surface. It is this parallel nature that makes shading so approachable by hardware and allows us to even consider real-time shading.

The model that almost every shading system adopts is Single Program/Multiple Data (SPMD), with no processor-to-processor communication. You write a shader to describe what happens to a single sample on the surface (single program). That same single program is run at every sample on the surface (multiple data). SPMD is closely related to the Single Instruction/Multiple Data (SIMD) model of parallel computation, but SIMD implies more about how the program will be executed. With SIMD, a set of parallel processors will runs the same set of instructions in lockstep, but with different data at

each processor. SPMD runs the same program, but without any implication of whether the same path through the program must be taken by all processors. On a pure SIMD array of processors, conditional code is handled by disabling a subset of the processors, who must wait while the others process the conditional instructions. Contrast this with the Multiple Instruction/Multiple Data (MIMD) model, where every processor can be running a completely different program.

SPMD is sometimes referred to as “SIMD on MIMD” or “effective SIMD”, as it is uses a SIMD style of programming, but can include programs to run on a single processor, MIMD machine or SIMD machine.

1.2. No communication

One of the aspects of shading that has allowed the explosion of fast shading hardware is the independence of each shading sample from every other shading sample. One of the most difficult and expensive aspect of general-purpose parallel machines is the communication network allowing the processors or nodes to communicate with each other. If the need for this communication is removed, the need for synchronization between the processors disappears, as does the need for physical connections between processors. The processors can be packed much more densely and are free to execute on batches of samples, samples in a pipeline, samples one at a time — whatever is most efficient.

Communication costs are also generally so high relative to computational costs, and so dependent on the machine architecture, that introducing processor to processor communication into a SPMD model greatly reduces the kinds of hardware a program can use effectively. The longer we can avoid communication, the more general our shaders will be.

Shaders don’t need sample to sample communication because shaders are typically restricted to computing only local lighting models. Anything that makes the appearance of one point on the surface depend on points elsewhere on the surface introduces the need a sample to sample communication. Shadows, global illumination and subsurface scattering are all on the list of effects that break the model to some degree if they are allowed in a shader.

General purpose computations, on the other hand, often require significant processor to processor communication. As graphics hardware becomes more powerful and flexible, there is an increasing desire to use it for other general purpose parallel computation. This comes at a cost in flexibility of the resulting code. I would argue that we need **two** computational models. A model including communication for general purpose computation on NVIDIA and ATI-style hardware, and a model for shading (possibly targeting the general model on hardware that supports it) that is task oriented and unifies vertex and fragment computations.

In the mean time, many people have succeeded in creating general purpose algorithms on GPUs with inter-processor communication. They achieve this feat through the use of multiple passes. On one pass, you write data into textures or buffers in the graphics card. On the following pass, **any** processor can read **any** data from this texture, not just its own. Even if you are willing to accept multiple passes through the hardware, this

communication method isn't perfect for all uses. The reader decides what other processor's data to read, and can read at most a handful of data per pass. Some computational algorithms map well to this model, while others would prefer to have the *writer* decide where the data should go. All of that will be covered in more depth later – for now, we'll restrict the discussion to shading.

2. Languages for hardware abstraction

One of the best examples of a shading language for hardware abstraction is the RenderMan shading language. Shaders written in this language have been successfully targeted to a huge range of different hardware. Pixar's PhotoRealistic RenderMan targets a single processor running each step of the shader in a loop over the micropolygons in a diced-up surface as generated by the REYES algorithm [Cook 1987]. BMRT also targeted a single processor, but as a ray-tracer it ran each shader in its entirety on one ray-intersection sample before moving on to the next sample [Gritz and Hahn 1996]. SGI created a RenderMan implementation targeting multiple rendering passes on graphics hardware, assuming hardware with a fast render-to-texture/read-from-texture or copy framebuffer-to-famebuffer [Peercy et al. 2000]. ATI has created a RenderMan language compiler targeting current shading hardware as part of their ASHLI toolkit.

RenderMan may not turn out to be the best language for hardware shading, but it has done an admirable job at being adaptable to a wide range of hardware. In the model presented by RenderMan, the shader writer tags data as being either *uniform* or *varying*. Uniform data is the same across a set of samples being shaded at once[1]. Varying data may change from sample to sample.

For compilation of RenderMan shaders, the most important uniform and varying designations are for the inputs to the shader. The shading compiler must derive for itself which intermediate results within an expression are uniform and which are varying. Expressions using only uniform arguments will have uniform results; expressions with any varying arguments will have a varying result. The compiler can use similar logic to decide whether any local variable in a shader is really uniform or varying regardless of how it was specified in the shading code.

Given an accurate idea of exactly which quantities vary across the shaded surface and which don't, the shading compiler can make several choices for actual execution. It can decide to still evaluate every computation at every sample (not using the uniform/varying distinction). It can evaluate the uniform computations once for a set of samples and for each varying computation, loop over the samples to evaluate it. It can execute the varying computations as SIMD instructions across a parallel array of processors. It can execute the entire shader or just the varying computations across a set of MIMD processors. It can create a pipeline of stream processors, each executing one or a few varying instructions on one sample before passing that sample on to the next.

---

[1] One RenderMan trick that will tell you something about how many samples are shaded at once, breaking the illusion that all hardware is the same, is to assign a random color into a uniform variable in a RenderMan shader.

3. Where should we break the portability?

There are several facets of the RenderMan shading language that are not well suited for graphics hardware. We can expect several of these to be the foundation of differences between real-time languages and the RenderMan shading language, or limitations of hardware-accelerated RenderMan implementations.

Since PRMan version 3.8, the RenderMan shading language has included the ability to call arbitrary code from within a shader. This code can do anything, from compute a specialized noise function to spawn a different style of renderer to download an image from a live camera on the south pole. Until graphics hardware has the ability to run arbitrary code, this won't really be an option for real-time shading.

RenderMan's has just one scalar data type, *float*. Graphics hardware supporting floating point data is now ubiquitous, but the size and precision of the floating point numbers vary. Fixed point or reduced-precision floating point numbers are also provided on some hardware as a faster option than pure 32-bit floating point. With no way to indicate the range or precision of computations, a RenderMan compiler cannot know when to use these faster operations. Many of the candidates for a real-time shading language include some reduced precision types for efficiency: the OpenGL Shading Language [Kessenich et al. 2003], Direct3D HLSL [Microsoft 2002], and Cg [Mark et al. 2003].

RenderMan shaders have two computational frequencies (how often a computation happens), uniform and varying. Shading hardware has at least three — compute on the CPU, compute per vertex and compute per fragment, with no interleaving of computation between the levels. All of the languages mentioned above have chosen to break shading computation into separate procedures executing at each of these levels. That choice makes those shaders a poor fit to any hardware or software rendering system that does not follow the CPU/Vertex/Fragment breakdown. However, any alternative language that targeted all three stages must include new types for the new types of computation [Proudfoot et al. 2001].

The RenderMan shading language also includes no real means of communication from sample to sample. This is one of the strengths that allow RenderMan shaders to run on such a wide range of rendering systems, but is a serious restriction for the general computations that are becoming popular on graphics hardware. Communication between processors in current hardware seems best supported by rendering partial results to a texture then using the random access provided by the texturing hardware to find values from other processors in a later pass.

This form of communication is currently limited to fragment shaders and comes at a very high price of communication to instructions. Similar communication at the vertex shader level is possible, though considerably more complicated. The all but the final vertex shader pass can operate on a regular grid of vertices, allowing all vertex and fragment operations to be used (including any vertex and fragment texturing and rendering to texture for communication). In the next-to-last vertex shader pass, the vertex locations (or data necessary to do the final computation) can be rendered in to a vertex array for use in a final vertex shader pass. If multiple passes of fragment shading are

needed, they must follow after all vertex shading passes, but need not repeat the multi-pass vertex shading computation.

Obviously, stretching the hardware beyond its intended limits like this introduces a significant burden on the shader developer! Because users want to write algorithms that use communication, better facilities will appear in real-time shading languages, but as they do they will limit the applicability of those shaders to the class of similar hardware.

4. Predictions

In this space last year, I predicted that within at most a year or two, we would see one or two languages, split along the Vertex/Fragment lines and supporting some simple communication model for general purpose computation. The first part (and most obvious) part of that prediction has come through, with Cg, HLSL and GLSlang filling the role of general and cross-platform high-level languages. I still expect some additional language support for communication to appear – too many people are struggling to map general purpose algorithms onto the GPU.

Following that, I believe we will see compilers for shading-specific languages like RenderMan so those who want to do shading can use the same shader on both hardware and software renderers. For the same reason that C or C++ is not a satisfying interface for writing shaders on software renderers and the RenderMan shading language isn't a satisfying interface for writing general purpose CPU code, we will see a specialization of languages for graphics hardware. General-purpose but targeted to a specific class of hardware for general-purpose programming and shading-specific but usable anywhere for shading use.