

A Survey of Programmable Graphics Hardware Languages

Video card technology has been one of the fastest growing areas of computer hardware over the last five years. It has not only surpassed transistor improvements, but also it has far exceeded Moore's law as well, seemingly following roughly Moore's law cubed. As a result of these improvements and mainstream computer graphics exposure, the demand for graphical realism has skyrocketed. Besides the basic speed improvements, graphics card manufacturers have added new types of technology to their video cards to render even more realistic images in real time. One of these new types of technology that has been developed is the idea of programmable hardware. Graphical Processing Units (GPU) have been made general purpose enough such that bits of code can now be sent to the video card and executed entirely on the GPU, freeing up the CPU for other calculations. Furthermore, because GPUs follow a very small instruction set, extreme optimizations can be made to these calculations so for certain programs, GPUs far outclass even the fastest CPUs. In the realm of computer graphics, these video card programs, also called shaders, have grown to serve two main purposes, vertex and fragment (pixel) shading.

In standard pre-programmable hardware fixed-pipeline graphics, the vertex positions are calculated on the software side (CPU), and sent to the graphics card for rasterization (converted from 3d geometry into a 2d image that gets sent to the monitor). However, vertex-shaders allow programs to add further levels of complexity to this pipeline. With programmable shaders, the vertices can now be sent over to the graphics card, and then have further functions applied to them before they get rasterized. Then, fragment shaders allow the pixels resulting from the rasterization to have even further algorithms applied to them. Because these steps all occur on the video board, they are processed almost immediately, increasing the complexity of scenes that can be rendered in

real-time. Some of the common uses for vertex-shaders and fragment-shaders are for effects like lighting and textures.

For a while, the only way to take advantage of these new hardware features was through extensions provided by the individual hardware manufacturers themselves. No general-purpose language existed for use. This was a huge problem, because it basically eliminated hardware portability among programs, which is something that is crucial in the realm of consumer-side computer graphics. An application that displayed correctly only on one specific hardware setup would be useless in most cases. In addition, coding for these specific hardware extensions was only doable with assembly language, which, while being very fast and efficient to execute, is neither fast nor efficient to program. Therefore, it became necessary for someone to develop a high level and more portable language to take advantage of these new features that existed on the hardware. This idea was tackled by three separate research teams. NVIDIA created C for Graphics (Cg), Microsoft created High Level Shading Language (HLSL) and 3dLabs created the OpenGL Shading Language (GLSL). Cg is slightly more general than the rest as it is designed to function on top of both the OpenGL and DirectX APIs, while GLSL and HLSL are designed to be used with their respective APIs. However, the differences between these languages seem to be more a result of design choices rather than different solutions to a similar problem.

While these three languages are not the first shading languages invented, they are the first that attempt to have a non-application specific solution to the problem of programmable shaders. The first stream processing with user programmable components was written in 1984 by Ken Perlin. However, this idea didn't reach the mainstream until the last five or so years when commercially available hardware incorporated support for it. Pixar and Stanford also co-developed a programmable shading language. However, this language was entirely application specific, as it was designed for use only with Pixar's RenderMan suite. In addition, it functioned on the software level, rather than on commercially available hardware. In addition, RenderMan is designed for off-

line rendering, whereas OpenGL and DirectX are both designed for real time rendering and graphics (online applications).

In addition to these software shaders, the OpenGL API incorporated a very basic, quickly constructed high level shading language, but it was not used as an industry standard, and was not very complete. People tended to write their shaders for applications with assembly languages that were defined by the specific hardware manufacturer's graphics chipsets. There have been several other application specific high level shading languages developed by game designers to take advantage of the shading features as well. Quake 3 was programmed with its own proprietary high level shading language. Sony's Playstation 2 even supports a full implementation of C for its shader languages, which is even processed on an entirely different graphics chip. This parallel development of application specific shading languages shows a strong commercial demand for a high level alternative to assembly.

It should be important to note that while these programmable shaders allow for stunning new real-time graphics, most of the same visual effects can actually be achieved with multipass techniques. Multipass techniques involve rendering the same scene multiple times slightly differently and then combining results. These techniques therefore require a high ratio of memory bandwidth to Arithmetic Logic Units (ALUs). However, VLSI technology has tended to go in the opposite direction and as a result; programmable processors have become much more widely used than multipass techniques.

All three of these languages are based very strongly on the syntax of C. This is because the C syntax has permeated the computer science world, so people will already be familiar with the syntax and won't have to learn an entire new language. Ease of adoptability is a very important goal for all of the language designers, if they hope for their language to become an industry standard in the computing community.

“C is a quirky, flawed and an enormous success. While accidents of history surely helped, it evidently satisfied a need for a system implementation language efficient enough to displace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments” [Ritchie 1993].

However, there are some places where these languages differ from C. These differences tend to either come out of the hardware differences between CPUs and GPUs, or because of different design goals for programs that are expected to be executed on these video cards. These two underlying differences get expressed very clearly in the language design.

While GPUs perform similar actions to CPUs, there are some key differences. The most fundamental of these is that GPUs are stream processors, which is very different from the sequential processing of a CPU. The difference between a stream processor and a sequential processor is that a stream processor takes a constant input stream, and works on that producing an output stream. This means that only one program will be executed at the same time, in a linear fashion. CPUs on the other hand, have to be designed to run multiple programs with multiple memory states interspliced with each other. This is also the reason why GPUs have been progressing at a rate far greater than Moore’s Law. While sequential processors have to have a cache and various other techniques to run multiple programs at a time, stream processors do not have to worry about this. A stream processor in the context of a GPU in essence means that each of these small vertex programs is run through the GPU many many times no competition for GPU cycles from other shader programs. This has a direct effect on the passing of information between programs. In a stream processor, the interaction between programs (vertex and fragment shaders) has to be one of data flow rather than procedural random access that CPUs support, and C is designed to take advantage of. On a sequential processor, in order for the vertex shader to communicate with a fragment shader, it has to write the data it wants to pass to the fragment shader into specified registers. Then, when the fragment shader gets executed in the next stage of the

processor pipeline, it will read these registers. These registers are not restricted to being physical registers as they can be virtual registers that map to memory, for expandability. Therefore, Cg, GLSL, and HLSL all have extended C's variable types to express this difference. All languages make a distinction between Varying, Attribute and Uniform values. Uniform values are ones that change at most once per primitive. These are set by the outside application and cannot be written inside the vertex or fragment shader code. Attribute variable types change at most once per vertex. These are used for frequently modified data as it is passed down the pipeline, such as the vertex position or normal vector. The last variable type is called varying. Varying variables change at most once each fragment. This last category is how the vertex shaders can communicate with the fragment shader. Vertex shaders can write to varying variables as they go through the stream, and then the fragment programs which come afterwards will be able to read the values (Fragment shaders cannot write to varying variables). Each one of these small shading program is designed to be executed a huge number of times, generally for each vertex (possibly hundred of thousands) and for each pixel (possibly millions). The difference between executions involves the changing of certain registers that are read at the beginning of the programs. This scheme is very different from the procedural accessing between programs in C.

Another attribute of a stream processor is that GPUs do not support indirect addressing, while CPUs do. This means that CPUs can read and write anywhere in memory with pointer dereferences, while GPUs only can pass values with data flow. As a result, none of the high level languages support pointers or pointer dereferencing. However, because it is expected that GPUs may one day evolve to support indirect addressing, Cg has included pointers, but simply disabled it in the current implementation. Another effect of this lack of indirect accessing is that it is impossible to implement a runtime stack for temporary variables, and so recursive functions are prohibited in shading languages as well. Furthermore, high level shading languages support a call-by-value scheme as allowed by this paradigm. This means that parameters are copied rather than passed in by reference. However, Cg implemented this using a bind-by-position scheme rather

than a bind-by-name scheme as GLSL, HLSL (and C) have. This means that in Cg, all the bindings are performed at compile time.

Another difference between GPUs and CPUs is that GPUs have a much smaller instruction set, as they are used for a much more specific purpose. As a result, high level shading languages have only incorporated a small subset of the C language. Because GPUs are designed for vector mathematics, they are able to perform vector operations incredibly fast. Therefore, shading languages all have implemented native support for vectors and vector operations, and in fact, scalars are even converted into vectors at compile time. Also, GPUs support a much smaller set of variable types. Most modern GPUs support only 32 bit floating point numbers, 16 bit floating point numbers, and 12 bit fixed floating point numbers. For a while, Cg also did not support the integer type, but they decided to re-implement it for increased compatibility with existing C programs. GLSL also decided to include integers in their language for their use in loops and array accesses. However, because this is the primary use of integers, they are all defined as low precision shorts. Because GPUs have different kinds of floats, different high level shading languages have chosen to expose this to different levels. GLSL and HLSL decided for the compiler to use hints about the use of the floats to determine which type the user intended, while Cg differentiates between them by different keywords (float/half/fixed). In addition, graphics hardware doesn't support any string or character data types, as these are not used in graphics. While GLSL does not support character arrays, Cg chose to implement them, also for compatibility issues.

Another slight difference can be seen as a result of the performance priority for these shading languages. Cg has disabled the type promotion that exists in C. This means that while `float X*2.0` casts the result to a double (often not what the user intended) in C, Cg determines the type of results based on the type of the variable, so `float X * 2.0` would result in a float. Therefore, possibly costly and undesirable conversions are avoided by the compiler.

Another difference in the structure of shader programs and general C programs is that shaders tend to be rather small, on the order of 10 to 100 lines. Therefore, the C and Java

constructs for “programming in the large” have been excluded from these languages. Specifically, if you can expect programs to be relatively small, much of the object-oriented design features of C++ and Java can be ignored. Classes, templates, exceptions and namespaces are all not supported in Cg, HLSL or GLSL. Another example of this is that while Cg supports most of the operator functionality of C, such as overloaded operators, structs and arrays, it has a very small subset of the control flow operations. Goto, switch statements and recursive functions are not supported. However to accommodate changes in GPUs, these keywords have been reserved in case they decide to implement them later.

There were many obstacles that these languages had to overcome. They have to support multiple generations of previous graphics architecture, as well as be general enough to be useful even with the rapid changes happening in the industry. In addition, it is a challenge for any language to make the jump from a research paper to industry standard, so these languages all had to choose their design goals very carefully so that it would be possible for their language to be adopted. These design goals reflect the implementations of the languages.

One of the more important design goals is program portability. These languages must be robust enough to function on different hardware setups and different graphics APIs. However, because of commercial constraints, GLSL and HLSL had to be designed for their owner’s APIs. Therefore, Cg has a distinct advantage over the other two languages in that it can function over either one.

In addition, programmer productivity is a very important design goal. The languages must be more useful than coding in assembly. One of the biggest problems with assembly language is that it is very hard to design or reason about code written in it. Therefore, the main draw of a high level shading language would be ease of construction. However, sometimes the assembly level can tell the programmer more about what the hardware is actually doing, giving them the possibility to tweak and improve the performance of the code. Therefore, it may be desirable to be able to access the assembly program that the compiler generates from the high level code. Cg and HLSL both

allow the user to either compile all of their Cg code into assembly and tweak it manually before sending it to the graphics card, or they can just send the Cg code itself, and the compiler will take care of all the assembly. However, GLSL takes the opposite approach and integrates the high level language directly into the API and driver, eliminating the need for an assembly language compilation at all.

These high level shading languages must also be able to support the full hardware specifications, including functionality that might come in the future. In addition to having the same functionality as assembly code, they also have to be able to maintain performance on a similar level with it as well. This is very important in an interactive, graphics-oriented environment, where better looking scenes generally take longer to compute. This goal of designing the language to be fully functional even in several generations of graphics cards later without breaking backward compatibility is extremely hard. Given the rapid rate of hardware change, it is virtually impossible to predict the kinds of hardware features that will be implemented in graphics cards in two years time. One example of the kind of change is that DirectX 9.0 hardware supports floating point fragments, while DirectX 8.0 compliant hardware did not. In this case, it is possible to emulate floating point operations for DirectX 8.0, but that would result in a vast hardware slowdown. The solution that Cg decided to implement was to have what they refer to as a "profile". Each piece of hardware comes with a profile, which describes what subset of Cg it supports. Cg currently supports 18 different hardware profiles. GL approached this problem from the opposite side with its extension interface. Hardware vendors can write OpenGL extensions which provide access functions to specific hardware capabilities. In essence, whereas Cg subtracts functionality based on hardware, GL adds it based on hardware functionality, accomplishing virtually the same thing. HLSL on the other hand simply enforces hardware vendors to make their cards compliant with a specific version of the DirectX language. Therefore, any piece of hardware is guaranteed only to work with the versions of DirectX that it is compliant with.

There is another area where the design goals, and resulting implementations, of CG and HLSL and GLSL diverge. One of the main tenants of Cg is that it should have a minimal interface with application data. This means that the scene data should not be used in shading computations, or more generally, that Cg should not be hard-coded with graphics in mind. Ideally, Cg should be able to be used for any generic problem, and simply be a tool that lets a coder take advantage of the fast GPU. There is a huge potential in the application of shader languages beyond the graphics world. Because GPUs deal with a much smaller instruction set, and perform vector math exclusively, they can be highly optimized to do this. Many problems can be converted into vector math problems and sent to the GPU. This could free the CPU from some very hard calculations. GLSL and HLSL on the other hand were both designed to be high level enough to deal with the graphics domain, but specific enough to allow for graphics based optimizations.

This difference of general purpose vs. domain specific is a fundamental difference that separates Cg from the other languages, and there are advantages to each approach. One of the advantages of a domain specific programming language is that you can improve programmer productivity by having specific functions to accomplish things that are often used in graphics. For example, built in cross products, or common shading algorithms are both very useful for graphics programmers, but don't serve much function for other unrelated problems. In addition, it is much easier to optimize the compiler if you know the kind of problems that will be written. On the other hand, general purpose languages have the potential to appeal to a much larger audience.

One example of this difference is that GLSL and HLSL are targeted towards executing vertex and fragment programs specifically, so they in essence have two very closely related, but separate languages. Cg however uses the same language for both fragment and vertex shaders. In addition, Cg does not support high level shading functions, nor does it have colors or point data types or support vector swizzling (`position.xyz = position.xzy` swizzles around the y and z values). Rather, Cg provides this same functionality by giving abstract tools for the definition of user defined data types. Another example of this is that Cg does not have any built in graphics specific

global such as the modelview matrix (`gl_ModelviewMatrix` in GLSL). However, Cg is not entirely general purpose. The Cg development team has realized that it will be used mostly for graphics purposes, and a visible concession of this is the `float4x4` data type (added by developer request). Instead of hardwiring all the common graphics methods, Cg includes libraries to satisfy the graphics designers. These libraries provide access to functions like discrete-differentiating instructions for shader anti-aliasing and other graphics problems.

One hard task of graphics applications is communication between the shading and lighting functions. Cg lets the user do this through Java style interfaces while GLSL and HLSL both just support a specific hard coded way to resolve this communication.

It was more than just coincidence that caused three separate companies to design a very similar language at the same time. Rather, a large market demand motivated its development, and because of both cross-pollination of ideas and similar design choices, these three languages ended up looking very similar. Most actual differences stem from the differences in design choices rather than different solutions for the same design goal. For example, Cg's goal of constructing a general purpose language rather than an application specific one altered the course of its development. This combined with aggressive advertising by NVIDIA has led it to be much more widely used than GLSL or HLSL, and probably will make it become the industry standard for programmable shader languages.

Papers read:

3DLabs. 2002 *OpenGL 2.0 shading language white paper, version 1.2*. Feb.

Kessenich, J., Baldwin D. 2003 *The OpenGL Shading Language, version 1.05*. Feb

Mark W.R, Glanville R, Akeley K, Kilgard M, 2003 *Cg: A System for programming graphics hardware in a C-like language*. Jun.

Microsoft Corp. 2002. *High-level shader language. DirectX 9.0*. Dec.