Cg: A system for programming graphics hardware in a C-like language

William R. Mark* R. Steven Glanville† Kurt Akeley† Mark J. Kilgard†

The University of Texas at Austin* NVIDIA Corporation†

Abstract

The latest real-time graphics architectures include programmable floating-point vertex and fragment processors, with support for data-dependent control flow in the vertex processor. We present a programming language and a supporting system that are designed for programming these stream processors. The language follows the philosophy of C, in that it is a hardware-oriented, general-purpose language, rather than an application-specific shading language. The language includes a variety of facilities designed to support the key architectural features of programmable graphics processors, and is designed to support multiple generations of graphics architectures with different levels of functionality. The system supports both of the major 3D graphics APIs: OpenGL and Direct3D. This paper identifies many of the choices that we faced as we designed the system, and explains why we made the decisions that we did.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture; D.3.4 [Programming Languages]: Processors – Compilers and code generation I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors; I.3.6 [Computer Graphics]: Methodology and Techniques—Languages

1 Introduction

Graphics architectures are now highly programmable, and support application-specified assembly language programs for both vertex processing and fragment processing. But it is already clear that the most effective tool for programming these architectures is a high level language. Such languages provide the usual benefits of program portability and improved programmer productivity, and they also make it easier develop programs incrementally and interactively, a benefit that is particularly valuable for shader programs.

In this paper we describe a system for programming graphics hardware that supports programs written in a new C-like language named Cg. The Cg language is based on both the syntax and the philosophy of C [Kernighan and Ritchie 1988]. In particular, Cg is intended to be general-purpose (as much as is possible on graphics hardware), rather than application specific, and is a hardware-oriented language. As in C, most data types and operators have an obvious mapping to hardware operations, so that it is easy to write high-performance code. Cg includes a variety of new features designed to efficiently support the unique architectural characteristics of programmable graphics processors. Cg also adopts a few features from C++ [Stroustrup 2000] and Java [Joy et al. 2000], but unlike these languages Cg is intended to be a language for "programming in the small," rather than "programming in the large."

Cg is most commonly used for implementing shading algorithms (Figure 1), but Cg is not an application-specific shading language in the sense that the RenderMan shading language [Hanrahan and Lawson 1990] or the Stanford real-time shading language (RTSL) [Proudfoot et al. 2001] are. For example, Cg omits high-level shading-specific facilities such as built-in support for separate surface and light shaders. It also omits specialized data types for colors and points, but supports general-purpose user-defined compound data types such as structs and arrays.

As is the case for almost all system designs, most features of the Cg language and system are not novel when considered individually. However, when considered as a whole, we believe that the system and its design goals are substantially different from any previously-implemented system for programming graphics hardware.

The design, implementation, and public release of the Cg system has occurred concurrently with the design and development of similar systems by 3Dlabs [2002], the OpenGL ARB [Kessenich et al. 2003], and Microsoft [2002b]. There has been significant cross-pollination of ideas between the different efforts, via both public and private channels, and all four systems have improved as a result of this exchange. We will discuss some of the remaining similarities and differences between these systems throughout this paper.

This paper discusses the Cg programmer interfaces (i.e. Cg language and APIs) and the high-level Cg system architecture. We focus on describing the key design choices that we faced and on explaining why we made the decisions we did, rather than providing a language tutorial or describing the system's detailed implementation and internal architecture. More information about the Cg language is available in the language specification [NVIDIA Corp. 2003a] and tutorial [Fernando and Kilgard 2003].

Figure 1: *Screen captures from a real-time Cg demo running on an NVIDIA GeForce™FX. The procedural paint shader makes the car's surface rustier as time progresses.*

2 Background

Off-line rendering systems have supported user-programmable components for many years. Early efforts included Perlin's pixel-stream editor [1985] and Cook's shade-tree system [1984].

*Formerly at NVIDIA, where this work was performed.
email: billmark@cs.utexas.edu, {steveg,kakeley,mjk}@nvidia.com

Today, most off-line rendering systems use the RenderMan shading language, which was specifically designed for procedural computation of surface and light properties.

In real-time rendering systems, support for user programmability has evolved with the underlying graphics hardware. The UNC PixelFlow architecture [Molnar et al. 1992] and its accompanying PFMan procedural shading language [Olano and Lastra 1998] and rendering API [Leech 1998] demonstrated the utility of real-time procedural shading capabilities. Commercial systems are only now reaching similar levels of flexibility and performance.

For many years, mainstream commercial graphics hardware was configurable, but not user programmable (e.g. RealityEngine [Akeley 1993]). SGI's OpenGL shader system [Peercy et al. 2000] and Quake III's shading language [Jaquays and Hook 1999] targeted the fragment-processing portion of this hardware using multipass rendering techniques, and demonstrated that mainstream developers would use higher-level tools to program graphics hardware.

Although multipass rendering techniques can map almost any computation onto hardware with just a few basic capabilities [Peercy et al. 2000], to perform well multipass techniques require hardware architectures with a high ratio of memory bandwidth to arithmetic units. But VLSI technology trends are driving systems in the opposite direction: arithmetic capability is growing faster than off-chip bandwidth [Dally and Poulton 1998].

In response to this trend, graphics architects began to incorporate programmable processors into both the vertex-processing and fragment-processing stages of single-chip graphics architectures [Lindholm et al. 2001]. The Stanford RTSL system [Proudfoot et al. 2001] was designed for this type of programmable graphics hardware. Earlier real-time shading systems had focused on fragment computations, but RTSL supports vertex computations as well. Using RTSL, a user writes a single program, but may specify whether particular computations should be mapped to the vertex processor or the fragment processor by using special data-type modifiers.

The most recent generation of PC graphics hardware (*DirectX 9* or *DX9* hardware, announced in 2002), continues the trend of adding additional programmable functionality to both the fragment and the vertex processors (Figure 2). The fragment processor adds flexible support for floating-point arithmetic and computed texture coordinates [Mitchell 2002; NVIDIA Corp. 2003b]. Of greater significance for languages and compilers, the vertex processor in some of these architectures departs from the previous SIMD programming model, by adding conditional branching functionality [NVIDIA Corp. 2003c]. This branching capability cannot be easily supported by RTSL for reasons that we will discuss later.

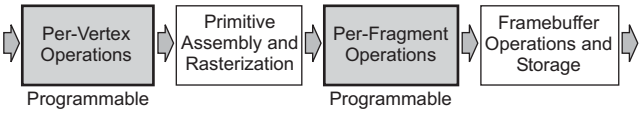

Figure 2: *Current graphics architectures* (DX9-class architectures) *include programmable floating-point vertex and fragment processors.*

Despite these advances in PC graphics architectures, they cannot yet support a complete implementation of C, as the SONY PlayStation 2 architecture does for its vertex processor that resides on a separate chip [Codeplay Corporation 2003].

Thus, by early 2001, when our group at NVIDIA began to experiment with programming languages for graphics hardware, it was clear that developers would need a high-level language to use future hardware effectively, but that each of the existing languages had significant shortcomings. Microsoft was interested in addressing this same problem, so the two companies collaborated on the design of a new language. NVIDIA refers to its implementation of the language, and the system that supports it, as Cg. In this paper, we consider the design of the Cg language and the design of the system that surrounds and supports it.

3 Design Goals

The language and system design was guided by a handful of high-level goals:

- **Ease of programming.**
  Programming in assembly language is slow and painful, and discourages the rapid experimentation with ideas and the easy reuse of code that the off-line rendering community has already shown to be crucial for shader design.
- **Portability.**
  We wanted programs to be portable across hardware from different companies, across hardware generations (for DX8-class hardware or better), across operating systems (Windows, Linux, and MacOS X), and across major 3D APIs (OpenGL [Segal and Akeley 2002] and DirectX [Microsoft Corp. 2002a]). Our goal of portability across APIs was largely motivated by the fact that GPU programs, and especially "shader" programs, are often best thought of as art assets – they are associated more closely with the 3D scene model than they are with the actual application code. As a result, a particular GPU program is often used by multiple applications (e.g. content-creation tools), and on different platforms (e.g. PCs and entertainment consoles).
- **Complete support for hardware functionality.**
  We believed that developers would be reluctant to use a high-level language if it blocked access to functionality that was available in assembly language.
- **Performance.**
  End users and developers pay close attention to the performance of graphics systems. Our goal was to design a language and system architecture that could provide performance equal to, or better than, typical hand-written GPU assembly code. We focused primarily on interactive applications.
- **Minimal interference with application data.**
  When designing any system layered between applications and the graphics hardware, it is tempting to have the system manage the scene data because doing so facilitates resource virtualization and certain global optimizations. Toolkits such as SGI's Performer [Rohlf and Helman 1994] and Electronic Arts's EAGL [Lalonde and Schenk 2002] are examples of software layers that successfully manage scene data, but their success depends on both their domain-specificity and on the willingness of application developers to organize their code in conforming ways. We wanted Cg to be usable in existing applications, without the need for substantial reorganization. And we wanted Cg to be applicable to a wide variety of interactive and non-interactive application categories. Past experience suggests that these goals are best achieved by avoiding management of scene data.
- **Ease of adoption.**
  In general, systems that use a familiar programming model and can be adopted incrementally are accepted more rapidly than systems that must be adopted on an all-or-nothing basis. For example, we wanted the Cg system to support integration of a vertex program written in Cg with a fragment program written in assembly language, and vice-versa.
- **Extensibility for future hardware.**
  Future programmable graphics architectures will be more flexible than today's architectures, and they will require

additional language functionality. We wanted to design a language that could be extended naturally without breaking backward compatibility.

- **Support for non-shading uses of the GPU.**
  Graphics processors are rapidly becoming sufficiently flexible that they can be used for tasks other than programmable transformation and shading (e.g. [Boltz et al. 2003]). We wanted to design a language that could support these new uses of GPUs.

Some of these goals are in partial conflict with each other. In cases of conflict, the goals of high performance and support for hardware functionality took precedence, as long as doing so did not fundamentally compromise the ease-of-use advantage of programming in a high-level language.

Often system designers must preserve substantial compatibility with old system interfaces (e.g. OpenGL is similar to IRIS GL). In our case, that was a non-goal because most pre-existing high level shader code (e.g. RenderMan shaders) must be modified anyway to achieve real-time performance on today's graphics architectures.

4 Key Design Decisions

4.1 A "general-purpose language", not a domain-specific "shading language"

Computer scientists have long debated the merits of domain-specific languages vs. general-purpose languages. We faced the same choice – should we design a language specifically tailored for shading computations, or a more general-purpose language intended to expose the fundamental capabilities of programmable graphics architectures?

Domain-specific languages have the potential to improve programmer productivity, to support domain-specific forms of modularity (such as surface and light shaders), and to use domain-specific information to support optimizations (e.g. disabling lights that are not visible from a particular surface). Most of these advantages are obtained by raising the language's abstraction level with domain-specific data types, operators, and control constructs.

These advantages are counterbalanced by a number of disadvantages that typically accompany a language based on higher-level abstractions. First, in contrast to a low-level language such as C, the run-time cost of language operators may not be obvious. For example, the RenderMan system may compute coordinate transformations that are not explicitly requested. Second, the language's abstraction may not match the abstraction desired by the user. For example, neither RenderMan nor RTSL can easily support OpenGL's standard lighting model because the OpenGL model uses separate light colors for the diffuse and specular light terms. Finally, if the domain-specific language abstraction does not match the underlying hardware architecture well, the language's compiler and runtime system may have to take complete control of the underlying hardware to translate between the two abstractions.

These issues – when considered with our design goals of high performance, minimal management of application data, and support for non-shading uses of GPU's – led us to develop a hardware-focused general-purpose language rather than a domain-specific shading language.

We were particularly inspired by the success of the C language in achieving goals for performance, portability, and generality of CPU programs that were very similar to our goals for a GPU language. One of C's designers, Dennis Ritchie, makes this point well [Ritchie 1993]:

> "C is 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."

These reasons, along with C's familiarity for developers, led us to use C's syntax, semantics, *and* philosophy as the initial basis for Cg's language specification. It was clear, however, that we would need to extend and modify C to support GPU architectures effectively.

Using C as the basis for a GPU language has another advantage: It provides a pre-defined evolutionary path for supporting future graphics architectures, which may include CPU-like features such as general-purpose indirect addressing. Cg reserves all C and C++ keywords so that features from these languages can be incorporated into future implementations of Cg as needed, without breaking backward compatibility.

As will become evident, Cg also selectively uses ideas from C++, Java, RenderMan, and RTSL. It has also drawn ideas from and contributed ideas to the contemporaneously-developed C-like shading languages from 3Dlabs (hereafter *3DLSL*), the OpenGL ARB (*GLSL*), and Microsoft (*HLSL*).

4.2 A program for each pipeline stage

The user-programmable processors in today's graphics architectures use a stream-processing model [Herwitz and Pomerene 1960; Stephens 1997; Kapasi et al. 2002], as shown earlier in Figure 2. In this model, a processor reads one element of data from an input stream, executes a program (*stream kernel)* that operates on this data, and writes one element of data to an output stream. For example, the vertex processor reads one untransformed vertex, executes the vertex program to transform the vertex, and writes the resulting transformed vertex to an output buffer. The output stream from the vertex processor passes through a non-programmable part of the pipeline (including primitive assembly, rasterization, and interpolation), before emerging as a stream of interpolated fragments that form the input stream to the fragment processor.

Choosing a programming model to layer on top of this stream-processing architecture was a major design question. We initially considered two major alternatives. The first, illustrated by RTSL and to a lesser extent by RenderMan, is to require that the user write a single program, with some auxiliary mechanism for specifying whether particular computations should be performed on the vertex processor or the fragment processor. The second, illustrated by the assembly-level interfaces in OpenGL and Direct3D, is to use two separate programs. In both cases, the programs consume an element of data from one stream, and write an element of data to another stream.

The unified vertex/fragment program model has a number of advantages. It encapsulates all of the computations for a shader in one piece of code, a feature that is particularly comfortable for programmers who are already familiar with RenderMan. It also allows the compiler to assist in deciding which processor will perform a particular computation. For example, in RTSL, if the programmer does not explicitly specify where a particular computation will be performed, the compiler infers the location using a set of well-defined rules. Finally, the single-program model facilitates source code modularity by allowing a single function to include related vertex and fragment computations.

However, the single-program model is not a natural match for the underlying dual-processor architecture. If the programmable processors omit support for branch instructions, the model can be effective, as RTSL demonstrated. But if the processors support branch instructions, the single-program model becomes very awkward. For example, this programming model allows arbitrary mixing of vertex and fragment operations within data-dependent

loops, but the architecture can support only fragment operations within fragment loops, and only vertex operations within vertex loops. It would be possible to define auxiliary language rules that forbid intermixed loop operations, but we concluded that the result would be an unreasonably confusing programming model that would eliminate many of the original advantages of the single-program model.

As a result, we decided to use a multi-program model for Cg. Besides eliminating the difficulties with data-dependent control flow, this model's closer correspondence to the underlying GPU architecture makes it easier to for users to estimate the performance of code, and allows the use of a less-intrusive compiler and runtime system. The multi-program model also allows applications to choose the active vertex program independently from the active fragment program. This capability had been requested by application developers.

A language for expressing stream kernels

After we made the decision to use a multi-program model for Cg, we realized that we had the opportunity to both simplify and generalize the language by eliminating most of the distinctions between vertex programs and fragment programs. We developed a single language specification for writing a stream kernel (i.e. vertex program or fragment program), and then allowed particular processors to omit support for some capabilities of the language. For example, although the core language allows the use of texture lookups in any program, the compiler will issue an error if the program is compiled for any of today's vertex processors since today's vertex processors don't support texture lookups. We will explain this mechanism in more detail later, in our discussion of Cg's general mechanism for supporting different graphics architectures.

The current Cg system can be thought of as a specialized stream processing system [Stephens 1997]. Unlike general stream processing languages such as StreamIt [Thies et al. 2002] or Brook [Buck and Hanrahan 2003], the Cg system does not provide a general mechanism for specifying how to connect stream processing kernels together. Instead, the Cg system relies on the established graphics pipeline dataflow of GPUs. Vertex data sent by the application is processed by the vertex kernel (i.e. the vertex program). The results of the vertex program are passed to primitive assembly, rasterization, and interpolation. Then the resulting interpolated fragment parameters are processed by the fragment kernel (i.e. the fragment program) to generate data used by the framebuffer-test unit to update the fragment's corresponding pixel. Cg's focus on kernel programming is similar to that of Imagine KernelC [Mattson 2001]. However, if the Cg language is considered separately from the rest of the Cg system, it is only mildly specialized for stream-kernel programming and could be extended to support other parallel programming models.

A data-flow interface for program inputs and outputs

For a system with a programming model based on separate vertex and fragment programs, a natural question arises: Should the system allow any vertex program to be used with any fragment program? Since the vertex program communicates with the fragment program (via the rasterizer/interpolator), how should the vertex program outputs and fragment program inputs be defined to ensure compatibility? In effect, this communication constitutes a user-defined interface between the vertex program and the fragment program, but the interface is a data-flow interface rather than a procedural interface of the sort that C programmers are accustomed to. A similar data-flow interface exists between the application and inputs to the vertex program (i.e. vertex arrays map to vertex program input registers).

When programming GPUs at the assembly level, the interface between fragment programs and vertex programs is established at the register level. For example, the user can establish a convention that the vertex program should write the normal vector to the TEXCOORD3 output register, so that it is available to the fragment program (after being interpolated) in its TEXCOORD3 input register. These registers may be physical registers or virtual registers (i.e. API resources that are bound to physical registers by the driver), but in either case the binding names must be chosen from a predefined namespace with predefined data types.

Cg and HLSL support this same mechanism, which can be considered to be a modified bind-by-name scheme in which a predefined auxiliary namespace is used instead of the user-defined identifier name. This approach provides maximum control over the generated code, which is crucial when Cg is used for the program on one side of the interface but not for the program on the other side. For example, this mechanism can be used to write a fragment program in Cg that will be compatible with a vertex program written in assembly language.

Cg (but not HLSL) also supports a bind-by-position scheme. Bind-by-position requires that data be organized in an ordered list (e.g. as a function-parameter list, or a list of structure members), with the outputs in a particular position mapping to inputs in that same position. This scheme avoids the need to refer to a predefined auxiliary namespace.

GLSL uses a third scheme, pure bind-by-name, that is not supported by either Cg or HLSL. In the pure bind-by-name scheme, the binding of identifiers to actual hardware registers must be deferred until after the vertex program and fragment program have been paired, which may not happen until link time or run time. In contrast, the bind-by-position approach allows the binding to be performed at compile time, without any knowledge of the program at the other side of the interface. For this reason, performance-oriented languages such as C that are designed for separate compile and link steps have generally chosen bind-by-position instead of bind-by-name.

4.3 Permit subsetting of language

Striking a balance between the often-conflicting goals of portability and comprehensive support for hardware functionality was a major design challenge. The functionality of GPU processors is growing rapidly, so there are major differences in functionality between the different graphics architectures that Cg supports. For example, DX9-class architectures support floating-point fragment arithmetic while most DX8-class architectures do not. Some DX9-class hardware supports branching in the vertex processor while other DX9-class hardware does not. Similarly, on all recent architectures the vertex processor and fragment processor support different functionality.

We considered a variety of possible approaches to hiding or exposing these differences. When minor architectural differences could be efficiently hidden by the compiler, we did so. However, since performance is important in graphics, major architectural differences cannot reasonably be hidden by a compiler. For example, floating-point arithmetic could be emulated on a fixed-point architecture but the resulting performance would be so poor that the emulation would be worthless for most applications.

A different approach is to choose a particular set of capabilities, and mandate that any implementation of the language support all of those capabilities and no others. If the *only* system-design goal had been to maximize portability, this approach would have been the right one. GLSL currently follows this approach, although it specifies a different set of capabilities for the vertex and fragment processor. However, given our other design goals, there was no reasonable point at which we could set the feature bar. We wanted

both to support the existing installed base of DX8-class hardware, and to provide access to the capabilities of the latest hardware. It could be argued that the presence of significant feature disparities is a one-time problem, but we disagree – feature disparities will persist as long as the capabilities of graphics hardware continue to improve, as we expect will happen.

Our remaining choice was to expose major architectural differences as differences in language capabilities. To minimize the impact on portability, we exposed the differences using a subsetting mechanism. Each processor is defined by a *profile* that specifies which subset of the full Cg specification is supported on that processor. Thus, program compatibility is only compromised for programs that use a feature that is not supported by all processors. For example, a program that uses texture mapping cannot be compiled with any current vertex profile. The explicit existence of this mechanism is one of the major differences between Cg and GLSL, and represents a significant difference in design philosophy. However, hardware vendors are free to implement subsets and supersets of GLSL using the OpenGL extension mechanism, potentially reducing the significance of this difference in practice.

The NVIDIA Cg compiler currently supports 18 different profiles, representing vertex and fragment processors for the DirectX 8, DirectX 9, and OpenGL APIs, along with various extensions and capability bits representing the functionality of different hardware. Although one might be concerned that this profile mechanism would make it difficult to write portable Cg programs, it is surprisingly easy to write a single Cg program that will run on all vertex profiles, or on all DX9-class fragment profiles. With care, it is even possible to write a single Cg program that will run on any fragment profile; the extra difficulty is caused by the idiosyncratic nature of DX8-class fragment hardware.

4.4 Modular system architecture

Any system has a variety of modules connected by internal and external interfaces. Taken as a whole, these constitute the system architecture. Cg's system architecture (Figure 3) includes much more than the language itself. More specifically, it includes an API that applications can use to compile and manage Cg programs (the *Cg runtime*), and several modules layered on top of existing graphics APIs.

Cg's architecture is more modular than that of the SGI, GLSL and RTSL systems but similar to that of HLSL. The architecture provides a high degree of flexibility for developers in deciding which parts of the system to use. For example, it is easy to use the complete Cg system to program the fragment processor while relying on the OpenGL API's conventional fixed-function routines to control the vertex processor. The modular nature of the system does makes it difficult to implement some optimizations that would cross module boundaries; this tradeoff is a classic one in systems design.

Metaprogramming systems (e.g. [McCool et al. 2002]), which use operator overloading to embed one language within another, have a very different system architecture. In metaprogramming systems, there is no clear boundary between the host CPU language, the embedded GPU language, and the mechanism for passing data between the two. This tight integration has some advantages, but we chose a more modular, conventional architecture for Cg. The two classes of system architectures are sufficiently different that we do not attempt to compare them in detail in this paper.

4.4.1 No mandatory virtualization

The most contentious system design question we faced was whether or not to automatically virtualize hardware resources using software-based multi-pass techniques. Current hardware limits the

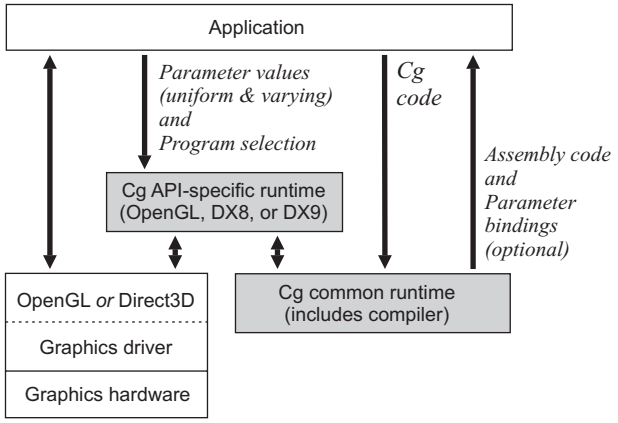

Figure 3: *Cg system architecture*

number of instructions, live temporary registers, bound textures, program inputs, and program outputs used by a program. Thus, without software-assisted virtualization a sufficiently complex program will exceed these limits and fail to compile. The limits on instruction count and temporary register count are potentially the most serious because the consumption of these resources is not clearly defined in a high-level language and may depend on compiler optimizations.

The SGI and RTSL systems demonstrated that it is possible to use multi-pass techniques to virtualize some resources for pre-DX8 hardware [Peercy et al. 2000; Proudfoot et al. 2001] and for later hardware [Chan et al. 2002]. However, we consider it to be impossible to efficiently, correctly, and automatically virtualize most DX8 architectures because the architectures use high-precision data types internally, but do not provide a high-precision framebuffer to store these data types between passes.

Despite the apparent advantages of automatic virtualization, we do not require it in the Cg language specification, and we do not support it in the current release of the Cg system. Several factors led to this decision. First, virtualization is most valuable on hardware with the fewest resources – DX8-class hardware in this case – but we had already concluded that effective virtualization of this hardware was impossible. Second, the resource limits on newer DX9-class hardware are set high enough that most programs that exceed the resource limits would run too slowly to be useful in a real-time application. Finally, virtualization on current hardware requires global management of application data and hardware resources that conflicted with our design goals. More specifically, the output from the vertex processor must be fed to the fragment processor, so multi-pass virtualization requires the system to manage simultaneously the vertex program and the fragment program, as well as all program parameters and various non-programmable graphics state. For example, when RTSL converts a long fragment program into multiple passes, it must also generate different vertex processor code for each pass.

Although Cg's language specification does not require virtualization, we took care to define the language so that it does not preclude virtualization. As long as the user avoids binding inputs and outputs to specific hardware registers, the language itself is virtualizable. For example, Cg adopts RTSL's approach of representing textures using identifiers (declared with special `sampler` types), rather than texture unit numbers, which are implicitly tied to a single rendering pass. Virtualization is likely to be useful for applications that can tolerate slow frame rates (e.g. 1 frame/sec), and for non-rendering uses of the GPU. Future hardware is likely to include better support for resource virtualization, at which point it would be easier for either the hardware driver or the Cg system to support it.

Of the systems contemporary with Cg, HLSL neither requires nor implements virtualization, and GLSL requires it only for

resources whose usage is not directly visible in the language (i.e. instructions and temporary registers).

4.4.2 Layered above an assembly language interface

High level languages are generally compiled to a machine/assembly language that runs directly on the hardware. The system designers must decide whether or not to expose this machine/assembly language as an additional interface for system users. If this interface is not exposed, the high level language serves as the only interface to the programmable hardware.

With a separate assembly language interface, the system is more modular. The compiler and associated run-time system may be distributed separately from the driver, or even shipped with the application itself. Users can choose between running the compiler as a command-line tool, or invoking it through an API at application run time. By providing access to the assembly code, the system allows users to tune their code by studying the compiler output, by manually editing the compiler output, or even by writing programs entirely in assembly language. All of these capabilities can be useful for maximizing performance, although they are less important if the compiler optimizes well.

In contrast, if the high-level language is the only interface to the hardware then the compiler must be integrated into the driver. This approach allows graphics architects to change the hardware instruction set in the future. Also, by forcing the user to compile via the driver, the system can guarantee that old applications will use compiler updates included in new drivers. However, the application developer loses the ability to guarantee that a particular pre-tested version of the compiler will be used. Since optimizing compilers are complex and frequently exhibit bugs at higher optimization levels, we considered this issue to be significant. Similarly, if the developer cannot control the compiler version, there is a risk that a program's use of non-virtualized resources could change and trigger a compilation failure where there was none before.

These and other factors led us to layer the Cg system above the low-level graphics API, with an assembly language serving as the interface between the two layers. RTSL and HLSL take this same approach, while GLSL takes the opposite approach of integrating the high-level language into the graphics API and driver.

4.4.3 Explicit program parameters

All input parameters to a Cg program must be explicitly declared using non-static global variables or by including the parameters on the entry function's parameter list. Similarly, the application is responsible for explicitly specifying the values for the parameters. Unlike GLSL, the core Cg specification does not include pre-defined global variables such as gl_ModelViewMatrix that are automatically filled from classical graphics API state. Such pre-defined variables are contrary to the philosophy of C and are not portable across 3D APIs with different state. We believe that even in shading programs all state used by vertex and fragment programs ought to be programmer-defined rather than mediated by fixed API-based definitions. However, pre-defined variables can be useful for retrofitting programmability into old applications, and for that reason some Cg profiles support them.

At the assembly language level, program inputs are passed in registers or, in some cases, named parameters. In either case, the parameter passing is untyped. For example, in the ARB_vertex_program assembly language each program parameter consists of four floating-point values. Because the Cg system is layered on top of the assembly-language level, developers may pass parameters to Cg programs in this manner if they wish.

However, Cg also provides a set of runtime API routines that allow parameters to be passed using their true names and types. GLSL uses a similar mechanism. In effect, this mechanism allows applications to pass parameters using Cg semantics rather than assembly-language semantics. Usually, this approach is easier and less error-prone than relying on the assembly-level parameter-passing mechanisms. These runtime routines make use of a header provided by the Cg compiler on its assembly language output that specifies the mapping between Cg parameters and registers (Figure 4). There are three versions of these runtime libraries – one for OpenGL, one for DirectX 8, and one for DirectX 9. Separate libraries were necessary to accommodate underlying API differences and to match the style of the respective APIs.

```
#profile arbvp1
#program simpleTransform
#semantic simpleTransform.brightness
#semantic simpleTransform.modelViewProjection
#var float4 objectPosition : $vin.POSITION : POSITION : 0 : 1
#var float4 color : $vin.COLOR : COLOR : 1 : 1
...
#var float brightness : : c[0] : 8 : 1
#var float4x4 modelViewProjection : : c[1], 4 : 9 : 1
```

Figure 4: *The Cg compiler prepends a header to its assembly code output to describe the mapping between program parameters and registers.*

5 Cg Language Summary

Although this paper is not intended to be a tutorial on the Cg language, we describe the language briefly. This description illustrates some of our design decisions and facilitates the discussions later in this paper.

5.1 Example program

Figure 5 shows a Cg program for a vertex processor. The program transforms an object-space position for a vertex by a four-by-four matrix containing the concatenation of the modeling, viewing, and projection transforms. The resulting vector is output as the clip-space position of the vertex. The per-vertex color is scaled by a floating-point parameter prior to output. Also, a texture coordinate set is passed through without modification.

```
void simpleTransform(float4 objectPosition :  POSITION,
                     float4 color          :  COLOR,
                     float4 decalCoord     :  TEXCOORD0,
                out  float4 clipPosition   :  POSITION,
                out  float4 oColor         :  COLOR,
                out  float4 oDecalCoord    :  TEXCOORD0,
            uniform  float  brightness,
            uniform  float4x4 modelViewProjection)

{
  clipPosition = mul(modelViewProjection, objectPosition);
  oColor = brightness * color;
  oDecalCoord = decalCoord;
}
```

Figure 5: *Example Cg Program for Vertex Processor*

Cg supports scalar data types such as float but also has first-class support for vector and matrix data types. The identifier float4 represents a vector of four floats, and float4x4 represents a matrix. The mul function is a standard library routine that performs matrix by vector multiplication. Cg provides function overloading like C++; mul is overloaded and may be used to multiply various combinations of vectors and matrices.

Cg provides the same operators as C. Unlike C, however, Cg operators accept and return vectors as well as scalars. For example, the scalar, brightness, scales the vector, color, as you would expect.

In Cg, declaring a vertex program parameter with the uniform modifier indicates that its value will not vary over a batch of vertices. The application must provide the value of such parameters. For example, the application must supply the modelViewProjection matrix and the brightness scalar, typically by using the Cg runtime library's API.

The POSITION, COLOR, and TEXCOORD0 identifiers following the objectPosition, color, and decalCoord parameters specify how these parameters are bound to API resources. In OpenGL, glVertex commands feed POSITION; glColor commands feed COLOR; and glMultiTexCoord commands feed TEXCOORDn.

The out modifier indicates that clipPosition, oColor, and oDecalCoord parameters are output by the program. The identifier following the colon after each of these parameters specifies how the output is fed to the primitive assembly and rasterization stages of the graphics pipeline.

5.2 Other Cg functionality

Cg provides structures and arrays, including multi-dimensional arrays; all of C's arithmetic operators (+, *, /, etc.); a boolean type and boolean and relational operators (||, &&, !, etc.); increment/decrement (++/- -) operators; the conditional expression operator (?:); assignment expressions (+=, etc.); and even the C comma operator.

Cg supports programmer-defined functions (in addition to pre-defined standard library functions), but recursive functions are not allowed. Cg provides only a subset of C's control flow constructs: (do, while, for, if, break, and continue). Other constructs, such as goto and switch, are not supported in the current Cg implementation, but the necessary keywords are reserved.

Cg provides built-in constructors for vector data types (similar to C++ but not user-definable): e.g. float4 a = float4(4.0, -2.0, 5.0, 3.0);

Swizzling is a way of rearranging components of vector values and constructing shorter or longer vectors. For example:

```
float2 b = a.yx;          // b = (-2.0, 4.0)
```

Cg does not currently support pointers or bitwise operations. Cg lacks most C++ features for "programming in the large" such as full classes, templates, operator overloading, exception handling, and namespaces. Cg supports #include, #define, #ifdef, etc. matching the C preprocessor.

6 Design Issues

6.1 Support for hardware

By design, the C language is close to the level of the hardware – it exposes the important capabilities of CPU hardware in the language. For example, it exposes hardware data types (with extensions such as long long if necessary) and the existence of pointers. As a result, the C language provides performance transparency – programmers have straightforward control over machine-level operations, and thus the performance of their code.

When designing Cg, we followed this philosophy. The discussion below is organized around the characteristics of GPU hardware that led to differences between Cg and C.

6.1.1 Stream processor

The stream processing model used by the programmable processors in graphics architectures is significantly different from the purely sequential programming model used on CPUs. Much of the new functionality in Cg (as compared to C) supports this stream programming model. In particular, a GPU program is executed many times – once for each vertex or fragment. To efficiently accommodate this repeated execution, the hardware provides two kinds of inputs to the program. The first kind of input changes with each invocation of the program and is carried in the incoming stream of vertices or fragments. An example is the vertex position. The second kind of input may remain unchanged for many invocations of the program; its value persists until a new value is sent from the CPU as an update to the processor state. An example is the modelview matrix. At the hardware level, these two types of inputs typically reside in different register sets.

A GPU language compiler must know the category to which an input belongs before it can generate assembly code. Given the hardware-oriented philosophy of Cg, we decided that the distinction should be made in the Cg source code. We adapted RenderMan's terminology for the two kinds of inputs: a *varying* input is carried with the incoming stream of data, while a uniform input is updated by an explicit state change. Consistent with the general-purpose stream-processor orientation of Cg, this same terminology is used for any processor within the GPU (i.e. vertex or fragment), unlike the scheme used in GLSL, which uses different terminology (and keywords) for varying-per-vertex and varying-per-fragment variables.

Cg uses the uniform type qualifier differently than RenderMan. In RenderMan, it may be used in any variable declaration and specifies a general property of the variable, whereas in Cg it may only be applied to program inputs and it specifies initialization behavior for the variable. In the RenderMan interpretation, all Cg temporary variables would be considered to be varying, and even a uniform input variable becomes varying once it has been rewritten within the program. This difference reflects the difference in the processor models assumed by RenderMan and Cg: RenderMan is designed for a SIMD processor, where many invocations of the program are executing in lockstep and temporary results can be shared, while Cg is designed for a stream processor in which each invocation of the program may execute asynchronously from others, and no sharing of temporary results is possible.

Computations that depend only on uniform parameters do not need to be redone for every vertex or fragment, and could be performed just once on the CPU with the result passed as a new uniform parameter. RTSL can perform this optimization, which may add or remove uniform parameters at the assembly language level. The current Cg compiler does not perform this optimization; if it did, applications would be required to pass uniform parameters through the Cg runtime system rather than passing them directly through the 3D API because the original inputs might no longer exist at the 3D API level. This optimization is an example of a global optimization that crosses system modules. We expect that the Cg system will support optimizations of this type in the future, but only when the application promises that it will pass all affected parameters using the Cg runtime API.

6.1.2 Data types

The data types supported by current graphics processors are different from those supported by standard CPUs, thus motivating corresponding adjustments in the Cg language.

Some graphics architectures support just one numeric data type, while others support multiple types. For example, the NVIDIA GeForce FX supports three different numeric data types in its fragment processor – 32-bit floating-point, 16-bit floating-point, and 12-bit fixed-point. In general, operations that use the lower-precision types are faster, so we wanted to provide some mechanism for using these data types. Several alternatives were possible. The first was to limit the language to a single float data type, and hope that the compiler could perform interval and/or

precision analysis to map some computations to the lower-precision types. This strategy is at odds with the philosophy of C, and has not proven to be successful in the past. The second alternative (used in GLSL) was to specify precision using hints, rather than first-class data types. This approach makes it impossible to overload functions based on the data types, a capability that we considered important for supporting high-performance library functions. The third alternative, used by Cg, is to include multiple numeric data types in the language. Cg includes float, half, and fixed data types.

Just as C provides some flexibility in the precision used for its different data types, the core Cg specification provides profiles with flexibility to specify the format used for each of the data types, within certain ranges. For example, in a profile that targets an architecture with just one floating-point type, half precision may be the same as float precision. For a few types (e.g. fixed and sampler), profiles are permitted to omit support when appropriate. In particular, the sampler types are used to represent textures, and thus are of no use in profiles that do not support texture lookups. However, to allow source code and data structures targeted at different profiles to be mixed in a single source file, the Cg specification requires that all profiles support definitions and declarations of all Cg data types, and to support corresponding assignment statements. The first two requirements are necessary because of a quirk of C syntax: correct parsing of C requires that the parser know whether an identifier was previously defined as a type or as a variable. The third requirement makes it easier to share data structures between different profiles.

In 3D rendering algorithms, three- and four-component vector and four-by-four matrix operations are common. As a result, most past and present graphics architectures directly support four-component vector arithmetic (see e.g. [Levinthal et al. 1987; Lindholm et al. 2001]). C's philosophy of exposing hardware data types suggests that these vector data types should be exposed, and there is precedent for doing so in both shading languages [Levinthal et al. 1987; Hanrahan and Lawson 1990] and in extensions to C [Motorola Corp. 1999]. Despite these precedents, we initially tried to avoid exposing these types by representing them indirectly with C's arrays-of-float syntax. This strategy failed because it did not provide a natural mechanism for programmers or the compiler to distinguish between the architecture's vectors (now float4 x), and an indirectly addressable array of scalars (now float x[4]). These two types must be stored differently and support different operations because current graphics architectures are restricted to 128-bit granularity for indirect addressing. Thus, Cg and GLSL include vector data types and operators, up to length four.

It would be possible to take the opposite approach to supporting short vector hardware, by omitting short vector data types from the language, and relying on the compiler to automatically combine scalar operations to form vectorized assembly code [Larsen and Amarasinghe 2000; Codeplay Corporation 2003]. This approach requires sophisticated compiler technology to achieve acceptable vectorization and obscures from the programmer the difference between code that will run fast and code that will not. At best, this fully automatic approach to vectorization can only hope to match the performance of languages such as Cg that allow both manual and automatic vectorization.

As a convenience for programmers, Cg also supports built-in matrix types and operations, up to size four by four. This decision was a concession to the primary use of Cg for rendering computations.

Current graphics processors do not support integer data types, but they do support boolean operations using condition codes and predicated instructions. Thus, we initially decided to omit support for the C int data type, but to add a bool data type for conditional operations. This change was partly inspired by the bool type in the latest C++ standard. We adjusted the data types expected by C's boolean operators and statements accordingly, so that most common C idioms work with no change. Because some graphics hardware supports highly-efficient vector operations on booleans, we extended C's boolean operations (&&, ||, ?:, etc.) to support bool vectors. For example, the expression bool2(true,false) ? float2(1,1) : float2(0,0) yields float2(1,0). Later, for better compatibility with C, we restored the int type to the Cg specification, but retained the bool type for operations that are naturally boolean and thus can be mapped to hardware condition-code registers.

6.1.3 Indirect addressing

CPUs support indirect addressing (i.e. pointer dereferencing) for reads or writes anywhere in memory. Current graphics processors have very limited indirect addressing capability – indirect addressing is available only when reading from the uniform registers, or sampling textures. Unfortunately, programs written in the C language use pointers frequently because C blurs the distinction between pointer types and array types.

Cg introduces a clear distinction between these two types, both syntactically and semantically. In particular, an array assignment in Cg semantically performs a copy of the entire array. Of course, if the compiler can determine that a full copy is unnecessary, it may (and often does) omit the copy operation from the generated code. Cg currently forbids the use of pointer types and operators, although we expect that as graphics processors become more general, Cg will re-introduce support for pointer types using the C pointer syntax.

To accommodate the limitations of current architectures, Cg permits profiles to impose significant restrictions on the declaration and use of array types, particularly on the use of computed indices (i.e. indirect addressing). However, these restrictions take the form of profile-dependent prohibitions, rather than syntactic changes to the language. Thus, these prohibitions can be relaxed or removed in the future, allowing future Cg profiles to support general array operations without syntactic changes. In contrast, 3DLSL used special syntax and function calls (e.g. element) for the array operations supported by current architectures, although its descendent GLSL switched to C-like array notation.

The lack of hardware support for indirect addressing of a read/write memory makes it impossible to implement a runtime stack to hold temporary variables, so Cg currently forbids recursive or co-recursive function calls. With this restriction, all temporary storage can be allocated statically by the compiler.

Read/write parameters to a C function must be declared using pointer types. We needed a different mechanism in Cg, and considered two options. The first was to adopt the C++ call-by-reference syntax and semantics, as 3DLSL did. However, call-by-reference semantics are usually implemented using indirect addressing, to handle the case of parameter aliasing by the calling function. On current architectures it is possible for a compiler to support these semantics without the use of indirect addressing, but this technique precludes separate compilation of different functions (i.e. compile and link), and we were concerned that this technique might not be adequate on future architectures. Instead, we decided to support call-by-value-result semantics, which can be implemented without the use of indirect addressing. We support these semantics using a notation that is new to C/C++ (in and out parameter modifiers, taken from Ada), thus leaving the C++ & notation available to support call-by-reference semantics in the future. GLSL takes this same approach.

6.1.4 Interaction with the rest of the graphics pipeline

In current graphics architectures, some of the input and output registers for the programmable processors are used to control the non-programmable parts of the graphics pipeline, rather than to pass general-purpose data. For example, the vertex processor must

store a position vector in a particular output register, so that it may be used by the rasterizer. Likewise, if the fragment processor modifies the depth value, it must write the new value to a particular output register that is read by the framebuffer depth-test unit. We could have chosen to pre-define global variables for these inputs and outputs, but instead we treat them as much as possible like other varying inputs and outputs. However, these inputs and outputs are only available by using the language's syntax for binding a parameter to a register, which is optional in other cases. To ensure program portability, the Cg specification mandates that certain register identifiers (e.g. POSITION) be supported as an output by all vertex profiles, and that certain other identifiers be supported by all fragment profiles.

6.1.5 Shading-specific hardware functionality

The latest generation of graphics hardware includes a variety of capabilities specialized for shading. For example, although texture sampling instructions can be thought of as memory-read instructions, their addressing modes and filtering are highly specialized for shading. The GeForce FX fragment processor also includes built-in discrete-differencing instructions [NVIDIA Corp. 2003b], which are useful for shader anti-aliasing.

We chose to expose these capabilities via Cg's standard library functions, rather than through the language itself. This approach maintains the general-purpose nature of the language, while supporting functionality that is important for shading. Thus, many of Cg's standard library functions are provided for more than just convenience – they are mechanisms for accessing particular hardware capabilities that would otherwise be unavailable.

In other cases, such as the lit function, library functions represent common shading idioms that may be implemented directly in the language, but can be more easily optimized by the compiler and hardware if they are explicitly identified.

Although we do not discuss the details of the Cg standard library in this paper, significant care went into its design. It supports a variety of mathematical, geometric, and specialized functions. When possible, the definitions were chosen to be the same as those used by the corresponding C standard library and/or RenderMan functions.

6.2 User-defined interfaces between modules

The RenderMan shading language and RTSL include support for separate surface and light shaders, and the classical fixed-function OpenGL pipeline does too, in a limited manner. However, these shaders don't actually execute independently; computing the color of any surface point requires binding the light shaders to the surface shader either explicitly or implicitly. In RenderMan and fixed-function OpenGL, the binding is performed implicitly by changing the current surface or light shaders. In RTSL, the application must explicitly bind the shaders at compile time.

Considered more fundamentally, this surface/light modularity consists of built-in surface and light object types that communicate across a built-in interface between the two types of objects. In this conceptual framework, a complete program is constructed from one surface object that invokes zero or more light objects via the built-in interface. There are several subtypes of light objects corresponding to directional, positional, etc. lights. Light objects of different subtypes contain different data (e.g. positional lights have a "light position" but directional lights do not).

It would have run contrary to the C-like philosophy of Cg to include specialized surface/light functionality in the language. However, the ability to write separate surface and light shaders has proven to be valuable, and we wanted to support it with more general language constructs.

The general-purpose solution we chose is adopted from Java and C#.[1] The programmer may define an interface, which specifies one or more function prototypes.[2] For example, an interface may define the prototypes for functions used to communicate between a surface shader and a light shader. An interface may be treated as a generic object type so that one routine (e.g. the surface shader) may call a method from another object (e.g. an object representing a light) using the function prototypes defined in the interface. The programmer implements the interface by defining a struct (i.e. class) that contains definitions for the interface's functions (i.e. methods). This language feature may be used to create programmer-defined categories of interoperable modules; Figure 6 shows how it may be used to implement separate surface and light shaders, although it is useful for other purposes too. GLSL and HLSL do not currently include any mechanism – either specialized or general-purpose – that provides equivalent functionality.

All current Cg language profiles require that the binding of interfaces to actual functions be resolvable at Cg compile time. This binding may be specified either in the Cg language (as would be done in Java), or via Cg runtime calls prior to compilation. Future profiles could relax the compile-time binding requirement, if the corresponding graphics instruction sets include an indirect jump instruction.

6.3 Other language design decisions

6.3.1 Function overloading by types and by profile

Our decision to support a wide variety of data types led us to conclude that we should support function overloading by data type. In particular, most of Cg's standard library functions have at least twelve variants for different data types, so following C's approach of specifying parameter types in function name suffixes would have been unwieldly.

Cg's function overloading mechanism is similar to that of C++, although Cg's matching rules are less complex. For simple cases, Cg's matching rules behave intuitively. However, since matching is performed in multiple dimensions (base type, vector length, etc.) and implicit type promotion is allowed, it is still possible to construct complex cases for which it is necessary to understand the matching rules to determine which overloaded function will be chosen.

Cg also permits functions to be overloaded by profile. Thus, it is possible to write multiple versions of a function that are optimized for different architectures, and the compiler will automatically chose the version for the current profile. For example, one version of a function might use standard arithmetic operations, while a second version uses a table lookup from a texture (Figure 7). This capability is useful for writing portable programs that include optimizations for particular architectures. Some wildcarding of profiles is supported – for example, it is possible to specify just vertex and fragment versions of a function, rather than specifying a version for every possible vertex and fragment profile. The overloading rules cause more-specific profile matches to be preferred over less-specific matches, so program portability can be ensured by defining one lowest-common-denominator version of the function.

[1] Unlike the other Cg features described in this paper, this capability is not yet supported in a public release (as of April 2003). It is currently being implemented and will be supported in a future Cg release.

[2] C++ provides a similar capability via pure virtual base classes. We chose Java's approach because we consider it to be cleaner and easier to understand.

```
// Declare interface to lights
interface Light {
  float3 direction(float3 from);
  float4 illuminate(float3 p, out float3 lv);
};

// Declare object type (light shader) for point lights
struct PointLight : Light {
  float3 pos, color;
  float3 direction(float3 p) { return pos - p; }
  float3 illuminate(float3 p, out float3 lv) {
    lv = normalize(direction(p));
    return color;
  }
};

// Declare object type (light shader) for directional lights
struct DirectionalLight : Light {
  float3 dir, color;
  float3 direction(float3 p) { return dir; }
  float3 illuminate(float3 p, out float3 lv) {
    lv = normalize(dir);
    return color;
  }
};

// Main program (surface shader)
float4 main(appin IN, out float4 COUT,
            uniform Light lights[]) {
  ...
  for (int i=0; i < lights.Length; i++) { // for each light
    Cl = lights[i].illuminate(IN.pos, L); // get dir/color
    color += Cl * Plastic(texcolor, L, Nn, In, 30); // apply
  }
  COUT = color;
}
```

Figure 6: *Cg's* interface *functionality may be used to implement separate surface and light shaders. The application must bind the light objects to the main program prior to compilation. In this example, the application would perform the binding by making Cg runtime API calls to specify the size and contents of the* lights *array, which is a parameter to* main.

6.3.2 Constants are typeless

In C, if x is declared as float, then the expression 2.0*x is evaluated at double precision. Often, this type promotion is not what the user intended, and it may cause an unintended performance penalty. In our experience, it is usually more natural to think of floating-point constants as being typeless.

This consideration led us to change the type promotion rules for constants. In Cg, a constant is either integer or floating-point, and otherwise has no influence on type promotion of operators. Thus, if y is declared as half, the expression 2.0*y is evaluated at half precision. Users may still explicitly assign types to constants with a suffix character (e.g. 2.0f), in which case the type promotion rules are identical to those in C. Internally, the new constant promotion rules are implemented by assigning a different type (cfloat or cint) to constants that do not have an explicit type suffix. These types always take lowest precedence in the operator type-promotion rules.

These new rules are particularly useful for developing a shader using float variables, then later tuning the performance by selectively changing float variables to half or fixed. This process does not require changes to the constants used by the program.

6.3.3 No type checking for textures

The Cg system leaves the responsibility for most texture management (e.g. loading textures, specifying texture formats, etc.) with the underlying 3D API. Thus, the Cg system has very little information about texture types – e.g. is a particular texture an RGB (float3) texture, or an RGBA (float4) texture? Since compile-time type checking is not possible in this situation, the user is responsible for insuring that Cg texture lookups are used in manner that is consistent with the way the application loads and binds the corresponding textures at run time. Stronger type checking would be possible by integrating the Cg system more tightly with the 3D API.

```
uniform samplerCUBE norm_cubemap;

// For ps_1_1 profile, use cubemap to normalize
ps_1_1 float3 mynormalize(float3 v) {
  return texCUBE(norm_cubemap, v.xyz).xyz;
}

// For ps_2_0 profile, use stdlib routine to normalize
ps_2_0 float3 mynormalize(float3 v) {
  return normalize(v);
}
```

Figure 7: *Function overloading by hardware profile facilitates the use of optimized versions of a function for particular hardware platforms.*

6.4 Runtime API

As described earlier, the Cg runtime API is composed of two parts. The first part is independent of the 3D API and provides a procedural interface to the compiler and its output. The second part is layered on top of the 3D API and is used to load and bind Cg programs, to pass uniform and varying parameters to them, and to perform miscellaneous housekeeping tasks. These interfaces are crucial for system usability since they provide the primary interface between the application and the Cg system. In this section, we discuss a few of the more interesting questions that arose in the design of the runtime API.

6.4.1 Compound types are exploded to cross API

Cg programs may declare uniform parameters with compound types such as structures and arrays. Typically, the application passes the values of these parameters to the Cg program by using the Cg runtime API. Unfortunately, most operating systems do not specify and/or require a standard binary format for compound data types. For example, a data structure defined in a FORTRAN program does not have the same memory layout as the equivalent data structure defined in a C program. Thus, it is difficult to define a natural binary format for passing compound data structures across an API. This problem has plagued API designers for a long time; OpenGL finessed one aspect of it by specifying 2D matrices in terms of 1D arrays.

There are several possible approaches to this issue. The first is to choose a particular binary format, presumably the one used by the dominant C/C++ compiler on the operating system. This approach makes it difficult to use the API from other languages, and invites cross-platform portability issues (e.g. between 32-bit and 64-bit machines). The second is to use Microsoft's .NET common type system [Microsoft Corp. 2003], which directly addresses this problem, but would have restricted the use of the Cg APIs to the .NET platform. We chose a third approach, which is to explode compound data structures into their constituent parts to pass them across the API. For example, a struct consisting of a float3 and a float must be passed using one API call for the float3, and a second API call for the float. Although this approach imposes some overhead,

it is not generally a performance bottleneck when it is used for passing uniform values.

6.4.2 Cg system can shadow parameter values

The Cg runtime can manage many Cg programs (both vertex and fragment) at once, each with its own uniform parameters. However, GPU hardware can only hold a limited number of programs and parameters at a time. Thus, the values of the active program's uniform parameters may be lost when a new program is loaded into the hardware. The Cg runtime can be configured to shadow a program's parameters, so that the parameter values persist when the active program is changed. Note that some, but not all, OpenGL extensions already implement this type of shadowing in the driver.

7 CgFX

The Cg language and runtime do not provide facilities for managing the non-programmable parts of the graphics pipeline, such as the framebuffer tests. Since many graphics applications find it useful to group the values for this non-programmable state with the corresponding GPU programs, this capability is supported with a set of language and API extensions to Cg, which we refer to as CgFX. We do not discuss CgFX in detail in this paper, but we will briefly summarize its additional capabilities to avoid confusion with the base Cg language. CgFX can represent and manage:

- Functions that execute on the CPU, to perform setup operations such as computing the inverse-transpose of the modelview matrix
- Multi-pass rendering effects
- Configurable graphics state such as texture filtering modes and framebuffer blend modes
- Assembly-language GPU programs
- Multiple implementations of a single shading effect

8 System Experiences

NVIDIA released a beta version of the Cg system in June 2002, and the 1.0 version of the system in December 2002. Windows and Linux versions of the system and its documentation are available for download [NVIDIA Corp. 2003a]. The system is already widely used.

The modularity of the system has proven to be valuable. From online forums and other feedback, it is clear that some developers use the full system, some use just the off-line compiler, and some use Cg for vertex programs but assembly language for fragment programs. We know that some users examine the assembly language output from the compiler because they complain when the compiler misses optimization opportunities. In some cases, these users have hand-tuned the compiler's assembly-code output to improve performance, typically after they have reached the point where their program produces the desired visual results.

To the best of our knowledge, our decision to omit automatic virtualization from the system has not been a serious obstacle for any developer using DX9-class hardware for an application that requires real-time frame rates. In contrast, we have heard numerous complaints about the resource limits in DX8 fragment hardware, but we still believe that we would not have been able to virtualize DX8 hardware well enough to satisfy developers.

Researchers are already using Cg to implement non-rendering algorithms on GPUs. Examples include fluid dynamics simulations and reaction-diffusion simulations (Figure 8).

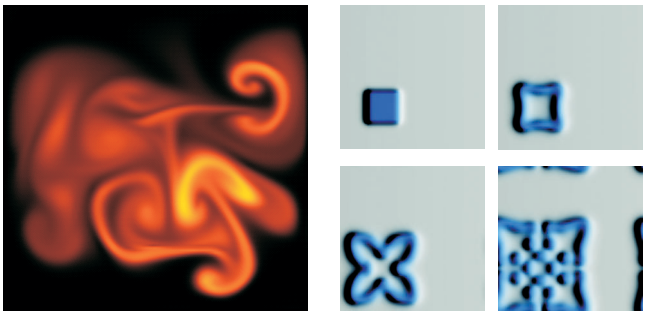

Figure 8: *Cg has been used to compute physical simulations on GPUs. Mark Harris at the University of North Carolina has implemented a Navier-Stokes fluid simulation (left) and a reaction-diffusion simulation (right).*

9 Conclusion

The Cg language is a C-like language for programming GPUs. It extends and restricts C in certain areas to support the stream-processing model used by programmable GPUs, and to support new data types and operations used by GPUs.

Current graphics architectures lack certain features that are standard on CPUs. Cg reflects the limitations of these architectures by restricting the use of standard C functionality, rather than by introducing new syntax or control constructs. As a result, we believe that Cg will grow to support future graphics architectures, by relaxing the current language restrictions and restoring C capabilities such as pointers that it currently omits.

If one considers all of the possible approaches to designing a programming language for GPUs, it is remarkable that the recent efforts originating at three different companies have produced such similar designs. In part, this similarity stems from extensive cross-pollination of ideas among the different efforts. However, we believe that a more significant factor is the de-facto agreement by the different system architects on the best set of choices for a contemporary GPU programming language. Where differences remain between between the contemporary systems, they often stem from an obvious difference in design goals, such as support for different 3D APIs.

We hope that this paper's discussion of the tradeoffs that we faced in the design of Cg will help users to better understand Cg and the other contemporary GPU programming systems, as well as the graphics architectures that they support. We also hope that this distillation of our experiences will be useful for future system architects and language designers, who will undoubtedly have to address many of the same issues that we faced.

10 Acknowledgments

Craig Peeper and Loren McQuade at Microsoft worked closely with us on the design of the Cg/HLSL language. If we had limited this paper to a discussion of the language itself, they probably would have been co-authors.

At NVIDIA, Cass Everitt helped to set the initial design direction for Cg. Craig Kolb designed most of the user-defined interface functionality described in Section 6.2, and Chris Wynn designed the standard library.

We designed and implemented Cg on a very tight schedule, which was only possible because of the highly talented team of people working on the project. Geoff Berry, Michael Bunnell, Chris Dodd, Cass Everitt, Wes Hunt, Craig Kolb, Jayant Kolhe, Rev Lebaredian, Nathan Paymer, Matt Pharr, Doug Rogers, and Chris Wynn developed the Cg compiler, standard library, and runtime technology. These individuals contributed to the language

design, the runtime API design, and the implementation of the system. Nick Triantos directed the project. Many other people inside and outside of NVIDIA contributed to the project; the Cg tutorial [Fernando and Kilgard 2003] includes a more complete list of acknowledgments. The Cg compiler backend for DX8-class hardware includes technology licensed from Stanford University [Mark and Proudfoot 2001].

Finally, we thank the anonymous reviewers of this paper for their thoughtful and constructive suggestions.

References

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

AKELEY, K. 1993. RealityEngine graphics. In *SIGGRAPH 93*, 109–116.

BOLTZ, J., FARMER, I., GRINSPUN, E., AND SCHRODER, P. 2003. The GPU as numerical simulation engine. In *SIGGRAPH 2003*.

BUCK, I., AND HANRAHAN, P. 2003. Data parallel computation on graphics hardware. unpublished report, Jan.

CHAN, E., NG, R., SEN, P., PROUDFOOT, K., AND HANRAHAN, P. 2002. Efficient partitioning of fragment shaders for multipass rendering on programmable graphics hardware. In *SIGGRAPH/Eurographics workshop on graphics hardware*.

CODEPLAY CORPORATION. 2003. *VectorC documentation*, Jan. Available at http://www.codeplay.com/support/documentation.html.

COOK, R. L. 1984. Shade trees. In *SIGGRAPH 84*, 223–231.

DALLY, W. J., AND POULTON, J. W. 1998. *Digital Systems Engineering*. Cambridge University Press.

FERNANDO, R., AND KILGARD, M. J. 2003. *The Cg Tutorial: The definitive guide to programmable real-time graphics*. Addison-Wesley.

HANRAHAN, P., AND LAWSON, J. 1990. A language for shading and lighting calculations. In *SIGGRAPH 90*, 289–298.

HERWITZ, P. S., AND POMERENE, J. H. 1960. The Harvest system. In *Proc. of the AIEE-ACM-IRE 1960 Western Joint Computer Conf.*, 23–32.

JAQUAYS, P., AND HOOK, B. 1999. *Quake 3: Arena Shader Manual, Revision 10*, Sept.

JOY, B., STEELE, G., GOSLING, J., AND BRACHA, G. 2000. *Java(TM) Language Specification*, 2nd ed. Addison-Wesley.

KAPASI, U. J., DALLY, W. J., RIXNER, S., OWENS, J. D., AND KHAILANY, B. 2002. The Imagine stream processor. In *Proc. of IEEE Conf. on Computer Design*, 295–302.

KERNIGHAN, B. W., AND RITCHIE, D. M. 1988. *The C Programming Language*. Prentice Hall.

KESSENICH, J., BALDWIN, D., AND ROST, R. 2003. *The OpenGL Shading Language, version 1.05*, Feb.

LALONDE, P., AND SCHENK, E. 2002. Shader-driven compilation of rendering assets. In *SIGGRAPH 2002*, 713–720.

LARSEN, S., AND AMARASINGHE, S. 2000. Exploiting superworld level parallelism with multimedia instruction sets. In *Proc. of ACM SIGPLAN PLDI 2000*, 145–156.

LEECH, J. 1998. OpenGL extensions and restrictions for PixelFlow. Technical Report UNC-CH TR98-019, Univ. of North Carolina at Chapel Hill, Dept. of Computer Science, Apr.

LEVINTHAL, A., HANRAHAN, P., PAQUETTE, M., AND LAWSON, J. 1987. Parallel computers for graphics applications. In *Proc. of 2nd Intl. Conf. on architectural support for programming languages and operating systems (ASPLOS '87)*, 193–198.

LINDHOLM, E., KILGARD, M. J., AND MORETON, H. 2001. A user-programmable vertex engine. In *SIGGRAPH 2001*.

MARK, W. R., AND PROUDFOOT, K. 2001. Compiling to a VLIW fragment pipeline. In *SIGGRAPH/Eurographics workshop on graphics hardware*.

MATTSON, P. 2001. *A Programming System for the Imagine Media Processor*. PhD thesis, Stanford University.

MCCOOL, M. D., QIN, Z., AND POPA, T. S. 2002. Shader metaprogramming. In *Eurographics/SIGGRAPH workshop on graphics hardware*, 57–68.

MICROSOFT CORP. 2002. *DirectX 9.0 graphics*, Dec. Available at http://msdn.microsoft.com/directx.

MICROSOFT CORP. 2002. High-level shader language. In *DirectX 9.0 graphics*. Dec. Available at http://msdn.microsoft.com/directx.

MICROSOFT CORP. 2003. Common type system. In *.NET framework developer's guide*. Jan. Available at http://msdn.microsoft.com/.

MITCHELL, J. L. 2002. *RADEON 9700 Shading (ATI Technologies white paper)*, July.

MOLNAR, S., EYLES, J., AND POULTON, J. 1992. PixelFlow: high-speed rendering using image composition. In *SIGGRAPH 92*, 231–240.

MOTOROLA CORP. 1999. *AltiVec Technology Programming Interface Manual*, June.

NVIDIA CORP. 2003. *Cg Toolkit, Release 1.1*. Software and documentation available at http://developer.nvidia.com/Cg.

NVIDIA CORP. 2003. NV_fragment_program. In *NVIDIA OpenGL Extension Specifications*. Jan.

NVIDIA CORP. 2003. NV_vertex_program2. In *NVIDIA OpenGL Extension Specifications*. Jan.

OLANO, M., AND LASTRA, A. 1998. A shading language on graphics hardware: The PixelFlow shading system. In *SIGGRAPH 98*, 159–168.

PEERCY, M., OLANO, M., AIREY, J., AND UNGAR, J. 2000. Interactive multi-pass programmable shading. In *SIGGRAPH 2000*, 425–432.

PERLIN, K. 1985. An image synthesizer. In *SIGGRAPH 85*, 287–296.

PROUDFOOT, K., MARK, W. R., TZVETKOV, S., AND HANRAHAN, P. 2001. A real-time procedural shading system for programmable graphics hardware. In *SIGGRAPH 2001*.

RITCHIE, D. M. 1993. The development of the C language. In *Second ACM SIGPLAN Conference on History of Programming Languages*, 201–208.

ROHLF, J., AND HELMAN, J. 1994. IRIS performer: A high performance multiprocessing toolkit for real–time 3D graphics. In *SIGGRAPH 94*, 381–394.

SEGAL, M., AND AKELEY, K. 2002. *The OpenGL Graphics System: A Specification (Version 1.4)*. OpenGL Architecture Review Board. Editor: Jon Leech.

STEPHENS, R. 1997. A survey of stream processing. *Acta Informatica 34*, 7, 491–541.

STROUSTRUP, B. 2000. *The C++ Programming Language*, 3rd ed. Addison-Wesley.

THIES, W., KARCZMAREK, M., AND AMARASINGHE, S. 2002. StreamIt: a language for streaming applications. In *Proc. Intl. Conf. on Compiler Construction*.