Chapter 5. Cg

Mark Kilgard

Chapter 5: Cg

Mark J. Kilgard
NVIDIA Corporation
Austin, Texas

This chapter explains NVIDIA's Cg Programming Language for programmable graphics hardware. Cg provides broad shader portability across a range of graphics hardware functionality (supporting programmable GPUs spanning the DirectX 8 and DirectX 9 feature sets). Shaders written in Cg can be used with OpenGL or Direct3D; Cg is API-neutral and does not tie your shader to a particular 3D API or platform. For example, Direct3D programmers can re-compile Cg programs with Microsoft's HLSL language implementation. Cg supports all versions of Windows (including legacy NT 4.0 and Windows 95 versions), Linux, Apple's OS X for the Macintosh, and Sony's upcoming PlayStation 3.

Collected in this chapter are the following articles:

- *Cg in Two Pages*: As the title indicates, this article summaries Cg in just two pages, including one vertex and one fragment program example.

- *Cg: A system for programming graphics hardware in a C-like language*: This longer SIGGRAPH 2002 paper explains the design rationale for Cg.

- *A Follow-up Cg Runtime Tutorial for Readers of The Cg Tutorial*: This article presents a complete but simple ANSI C program that uses OpenGL, GLUT, and the Cg runtime to render a bump-mapped torus using Cg vertex and fragment shaders from Chapter 8 of *The Cg Tutorial*. It's easier than you think to integrate Cg into your application; this article explains how!

- *Comparison Tables for HLSL, OpenGL Shading Language, and Cg*: Are you looking for a side-by-side comparison of the various features of the several different hardware-accelerated shading languages available to you today?

Cg in Two Pages

Mark J. Kilgard
NVIDIA Corporation
Austin, Texas
January 16, 2003

1. Cg by Example

Cg is a language for programming GPUs. Cg programs look a lot like C programs. Here is a Cg vertex program:

```
void simpleTransform(float4 objectPosition : POSITION,
                     float4 color          : COLOR,
                     float4 decalCoord     : TEXCOORD0,
                     float4 lightMapCoord  : TEXCOORD1,
                 out float4 clipPosition   : POSITION,
                 out float4 oColor         : COLOR,
                 out float4 oDecalCoord    : TEXCOORD0,
                 out float4 oLightMapCoord : TEXCOORD1,
             uniform float brightness,
             uniform float4x4 modelViewProjection)
{
  clipPosition = mul(modelViewProjection, objectPosition);
  oColor = brightness * color;
  oDecalCoord = decalCoord;
  oLightMapCoord = lightMapCoord;
}
```

1.1 Vertex Program Explanation

The program transforms an object-space position for a vertex by a 4x4 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, two texture coordinate sets are passed through unperturbed.

Cg supports scalar data types such as `float` but also has first-class support for vector data types. `float4` represents a vector of four floats. `float4x4` represents a matrix. `mul` is a standard library routine that performs matrix by vector multiplication. Cg provides function overloading like C++; `mul` is an overloaded function so it can be used to multiply all 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 parameter with the `uniform` modifier indicates that its value is initialized by an external source that will not vary over a given batch of vertices. In this respect, the `uniform` modifier in Cg is different from the `uniform` modifier in RenderMan but used in similar contexts. In practice, the external source is some OpenGL or Direct3D state that your application takes care to load appropriately. For example, your application must supply the `modelViewProjection` matrix and the `brightness` scalar. The Cg runtime library provides an API for loading your application state into the appropriate API state required by the compiled program.

The `POSITION`, `COLOR`, `TEXCOORD0`, and `TEXCOORD1` identifiers following the `objectPosition`, `color`, `decalCoord`, and `lightMapCoord` parameters are input semantics. They indicate how their parameters are initialized by per-vertex varying data. In OpenGL, `glVertex` commands feed the `POSITION` input semantic; `glColor` commands feed the `COLOR` semantic; glMultiTexCoord commands feed the `TEXCOORD`*n* semantics.

The `out` modifier indicates that `clipPosition`, `oColor`, oDecalCoord, and `oLightMapCoord` parameters are output by the program. The semantics that follow these parameters are therefore output semantics. The respective semantics indicate the program outputs a transformed clip-space position and a scaled color. Also, two sets of texture coordinates are passed through. The resulting vertex is feed to primitive assembly to eventually generate a primitive for rasterization.

Compiling the program requires the program source code, the name of the entry function to compile (`simpleTransform`), and a profile name (`vs_1_1`).

The Cg compiler can then compile the above Cg program into the following DirectX 8 vertex shader:

```
vs.1.1
mov oT0, v7
mov oT1, v8
dp4 oPos.x, c1, v0
dp4 oPos.y, c2, v0
dp4 oPos.z, c3, v0
dp4 oPos.w, c4, v0
mul oD0, c0.x, v5
```

The profile indicates for what API execution environment the program should be compiled. This same program can be compiled for the DirectX 9 vertex shader profile (`vs_2_0`), the multi-vendor OpenGL vertex program extension (`arbvp1`), or NVIDIA-proprietary OpenGL extensions (`vp20` & `vp30`).

The process of compiling Cg programs can take place during the initialization of your application using Cg. The Cg runtime contains the Cg compiler as well as API-dependent routines that greatly simplify the process of configuring your compiled program for use with either OpenGL or Direct3D.

1.2 Fragment Program Explanation

In addition to writing programs to process vertices, you can write programs to process fragments. Here is a Cg fragment program:

```
float4 brightLightMapDecal(float4 color         : COLOR,
                           float4 decalCoord    : TEXCOORD0,
                           float4 lightMapCoord : TEXCOORD1,
                   uniform sampler2D decal,
                   uniform sampler2D lightMap) : COLOR
{
  float4 d = tex2Dproj(decal, decalCoord);
  float4 lm = tex2Dproj(lightMap, lightMapCoord);
  return 2.0 * color * d * lm;
}
```

The input parameters correspond to the interpolated color and two texture coordinate sets as designated by their input semantics.

The `sampler2D` type corresponds to a 2D texture unit. The Cg standard library routine `tex2Dproj` performs a projective 2D texture lookup. The two `tex2Dproj` calls sample a decal and light map texture and assign the result to the local variables, `d` and `lm`, respectively.

The program multiplies the two textures results, the interpolated color, and the constant 2.0 together and returns this RGBA color.

The program returns a `float4` and the semantic for the return value is `COLOR`, the final color of the fragment.

The Cg compiler generates the following code for `brightLightMapDecal` when compiled with the `arbfp1` multi-vendor OpenGL fragment profile:

```
!!ARBfp1.0
PARAM c0 = {2, 2, 2, 2}; TEMP R0; TEMP R1; TEMP R2;
TXP R0, fragment.texcoord[0], texture[0], 2D;
TXP R1, fragment.texcoord[1], texture[1], 2D;
MUL R2, c0.x, fragment.color.primary;
MUL R0, R2, R0;
MUL result.color, R0, R1;
END
```

This same program also compiles for the DirectX 8 and 9 profiles (`ps_1_3` & `ps_2_x`) and NVIDIA-proprietary OpenGL extensions (`fp20` & `fp30`).

2. Other Cg Functionality

2.1 Features from C

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

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

Like C, Cg does not mandate the precision and range of its data types. In practice, the profile chosen for compilation determines the concrete representation for each data type. `float`, `half`, and `double` are meant to represent continuous values, ideally in floating-point, but this can depend on the profile. `half` is intended for a 16-bit half-precision floating-point data type. (NVIDIA's CineFX architecture provides such a data type.) `int` is an integer data type, usually used for looping and indexing. `fixed` is an additional data type intended to represent a fixed-point continuous data type that may not be floating-point.

Cg provides `#include`, `#define`, `#ifdef`, etc. matching the C preprocessor. Cg supports C and C++ comments.

2.2 Additional Features Not in C

Cg provides built-in constructors (similar to C++ but not user-defined) for vector data types:

```
float4 vec1 = 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. Example:

```
float2 vec2 = vec1.yx;         // vec2 = (-2.0, 4.0)
float scalar = vec1.w;         // scalar = 3.0
float3 vec3 = scalar.xxx;      // vec3 = (3.0, 3.0, 3.0)
```

More complicated swizzling syntax is available for matrices. Vector and matrix elements can also be accessed with standard array indexing syntax as well.

Write masking restricts vector assignments to indicated components. Example:

```
vec1.xw = vec3;  // vec1 = (3.0, -2.0, 5.0, 3.0)
```

Use either `.xyzw` or `.rgba` suffixes swizzling and write masking.

The Cg standard library includes a large set of built-in functions for mathematics (`abs`, `dot`, `log2`, `reflect`, `rsqrt`, etc.) and texture access (`texCUBE`, `tex3Dproj`, etc.). The standard library makes extensive use of function overloading (similar to C++) to support different vector lengths and data types. There is no need to use `#include` to obtain prototypes for standard library routines as in C; Cg standard library routines are automatically prototyped.

In addition to the `out` modifier for *call-by-result* parameter passing, the `inout` modifier treats a parameter as both a *call-by-value* input parameter and a *call-by-result* output parameter.

The `discard` keyword is similar to `return` but aborts the processing without returning a transformed fragment.

2.3 Features Not Supported

Cg has no support currently for pointers or bitwise operations (however, the necessary C operators and keywords are reserved for this purpose). Cg does not (currently) support unions and function variables.

Cg lacks C++ features for "programming in the large" such as classes, templates, operator overloading, exception handling, and namespaces.

The Cg standard library lacks routines for functionality such as string processing, file input/output, and memory allocation, which is beyond the specialized scope of Cg.

However, Cg reserves all C and C++ keywords so that features from these languages could be incorporated into future implementations of Cg as warranted.

3. Profile Dependencies

When you compile a C or C++ program, you expect it to compile without regard to how big (within reason) the program is or what the program does. With Cg, a syntactically and semantically correct program may still not compile due to limitations of the profile for which you are compiling the program.

For example, it is currently an error to access a texture when compiling with a vertex profile. Future vertex profiles may well allow texture accesses, but existing vertex profiles do not. Other errors are more inherent. For example, a fragment profile should not output a parameter with a `TEXCOORD0` semantic. Other errors may be due to exceeding a capacity limit of current GPUs such as the maximum number of instructions or the number of texture units available.

Understand that these profile dependent errors do not reflect limitations of the Cg language, but rather limitations of the current implementation of Cg or the underlying hardware limitations of your target GPU.

4. Compatibility and Portability

NVIDIA's Cg implementation and Microsoft's High Level Shader Language (HLSL) are very similar as they were co-developed. HLSL is integrated with DirectX 9 and the Windows operating system. Cg provides support for multiple APIs (OpenGL, Direct X 8, and Direct X 9) and multiple operating systems (Windows, Linux, and Mac OS X). Because Cg interfaces to multi-vendor APIs, Cg runs on GPUs from multiple vendors.

5. More Information

Read the *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics* (ISBN 0321194969) published by Addison-Wesley.

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.

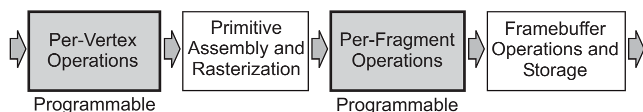

Figure 2: *Current graphics architectures* (DX9-class architectures) *include programmable floating-point vertex and fragment processors.*

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

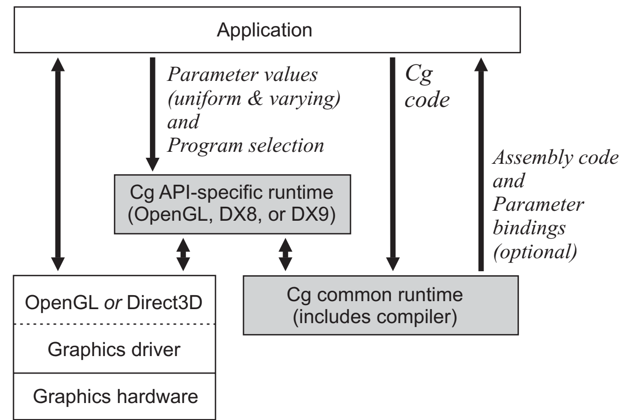

Figure 3: *Cg system architecture*

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.)

```
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.*

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.

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).

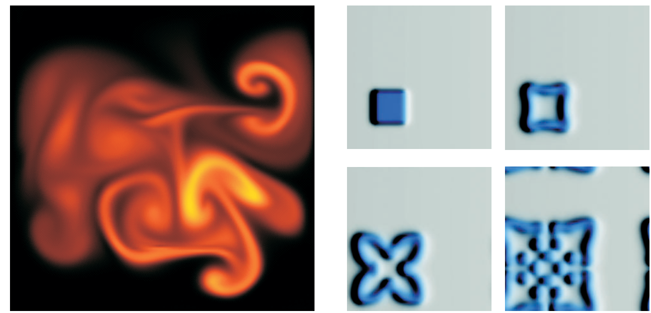

Figure 8: *Cg has been used to compute physical simulations on GPUs. Mark Harris at the University of North Carolina has implemented a Navier-Stokes fluid simulation (left) and a reaction-diffusion simulation (right).*

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*.

A Follow-up Cg Runtime Tutorial for Readers of *The Cg Tutorial*[*]

Mark J. Kilgard
NVIDIA Corporation
Austin, Texas

April 20, 2005

When Randy and I wrote *The Cg Tutorial*,[†] we wanted a book that would convey our intense enthusiasm for programmable graphics using Cg,[‡] short for C for Graphics. We focused our tutorial on the language itself: What is the Cg language and how do you write Cg programs for programmable graphics hardware?

We chose our language focus for a couple of different reasons.

First off, the language is where all the power and new concepts are. Once you interface Cg into your graphics application, it's the Cg language that really matters. For a conventional CPU programming language, explaining the Cg runtime is somewhat akin to explaining how to edit programs and how to run the compiler. Obviously, you've got to learn these tasks, but there's nothing profound about using an editor or compiler. Likewise, there's nothing deep about the Cg runtime either; it's a fairly straightforward programming interface.

Second, how you interface Cg to your application is a matter of personal design and depends on the nature of your application and your choice of application programming language, operating system, and 3D programming interface. While Randy and I are happy to explain Cg and show how to program your graphics hardware with it, you are the person best able to interface Cg into your application code.

Third, the language shares its design, syntax, and semantics with Microsoft's DirectX 9 High-Level Shader Language (HLSL). This means you can chose whether to use Microsoft's HLSL runtime (ideal for developers focused on DirectX for the Windows

[*] You have permission to redistribute or make digital or hard copy of this article for non-commercial or educational use.

[†] *The Cg Tutorial* by Randima (Randy) Fernando and Mark J. Kilgard is published by Addison-Wesley (ISBN 0321194969, 336 pages). The book is now available in Japanese translation (ISBN4-939007-55-3).

[‡] *Cg in Two Pages* (http://xxx.lanl.gov/ftp/cs/papers/0302/0302013.pdf) by Mark J. Kilgard is a short overview of the Cg language. *Cg: A System for Programming Graphics Hardware in a C-like Language* (http://www.cs.utexas.edu/users/billmark/papers/Cg) by Bill Mark, Steve Glanville, Kurt Akeley, and Mark J. Kilgard is a SIGGRAPH 2003 paper explaining Cg's design in 12 pages.

platform) or the Cg runtime—supplied by NVIDIA—for those of you who want to support a broad range of operating systems and 3D programming interfaces (such as Linux, Apple's OS X, and OpenGL). Because *The Cg Tutorial* focuses on the Cg language, all the concepts and syntax explained in the book apply whether you choose to use the Cg or HLSL implementation when it comes time to actually write your shader programs. Since there's been some confusion about this point, understand that *The Cg Tutorial* examples in the book compile with *either* language implementation. We hope *The Cg Tutorial* is an instructive book about both Cg *and* HLSL.

To avoid all the mundane details necessary to interface Cg programs to a real application, *The Cg Tutorial* includes an accompanying CD-ROM[*] with a software framework so you can examine and modify the various Cg programs in the book and see the rendering results without worrying about the mundane details of writing a full application, loading models and textures, and interfacing Cg to your application. Still, the book does provide a brief appendix describing the Cg runtime programming interface for both OpenGL and Direct3D.

Follow-up: A Complete Cg Demo

Still, there's not a *complete* basic example that shows how everything fits together. With that in mind, this article presents a complete graphics demo written in ANSI C that renders a procedurally-generated bump-mapped torus. The demo's two Cg programs are taken directly from the book's Chapter 8 (Bump Mapping). While the Cg programs are reprinted at the end of the article, please consult *The Cg Tutorial* for an explanation of the programs and the underlying bump mapping background and mathematics.

The demo renders with OpenGL and interfaces with the window system via the cross-platform OpenGL Utility Toolkit (GLUT).[†] To interface the application with the Cg programs, the demo calls the generic Cg and OpenGL-specific CgGL runtime routines.

OpenGL, GLUT, and the Cg and CgGL runtimes are supported on Windows, OS X, and Linux so the demo source code compiles and runs on all these operating systems. The demo automatically selects the most appropriate profile for your hardware. Cg supports multi-vendor OpenGL profiles (namely, **`arbvp1`** and **`arbfp1`**) so the demo works on GPUs from ATI, NVIDIA, or any other OpenGL implementation, such as Brian Paul's open source Mesa library, that exposes the multi-vendor **`ARB_vertex_program`** and **`ARB_fragment_program`** OpenGL extensions.

---

[*] You can download the latest version of the software accompanying *The Cg Tutorial* from http://developer.nvidia.com/object/cg_tutorial_software.html for either Windows or Linux. For best results, make sure you have the latest graphics drivers, latest Cg toolkit, and latest version of *The Cg Tutorial* examples installed.

[†] Documentation, source code, and pre-compiled GLUT libraries are available from http://www.opengl.org/developers/documentation/glut.html

I verified the demo works on DirectX 9-class hardware including ATI's Radeon 9700 and similar GPUs, NVIDIA's GeForce FX products, and the GeForce 6 Series. The demo even works on older NVIDIA DirectX 8-class hardware such as GeForce3 and GeForce4 Ti GPUs.

So this article's simple Cg-based demo handles multiple operating systems, two different GPU hardware generations (DirectX 8 & DirectX 9), and hardware from the two major GPU vendors (and presumably any other OpenGL implementation exposing OpenGL's standard, multi-vendor vertex and fragment program extensions) with absolutely no GPU-dependent or operating system-dependent code.

To further demonstrate the portability possible by writing shaders in Cg, you can also compile the discussed Cg programs with Microsoft's HLSL runtime with no changes to the Cg programs.

This unmatched level of shader portability is why the Cg language radically changes how graphics applications get at programmable shading hardware today. With one high-level language, you can write high-performance, cross-platform, cross-vendor, and cross-3D API shaders. Just as you can interchange images and textures stored as JPEG, PNG, and Targa files across platforms, you can now achieve a similar level of interoperability with something as seemingly hardware-dependent as a hardware shading algorithm.

Demo Source Code Walkthrough

The demo, named `cg_bumpdemo`, consists of the following five source files:

1. `cg_bumpdemo.c`—ANSI C source code for the demo.
2. `brick_image.h`—Header file containing RGB8 image data for a mipmapped 128x128 normal map for a brick pattern.
3. `nmap_image.h`—Header file containing RGB8 image data for a normalization vector cube map with 32x32 faces.
4. `C8E6v_torus.cg`—Cg vertex program to generate a torus from a 2D mesh of vertices.
5. `C8E4f_specSurf.cg`—Cg fragment program for surface-local specular and diffuse bump mapping.

Later, we will go through `cg_bumpdemo.c` line-by-line.

To keep the demo self-contained and maintain the focus on how the Cg runtime loads, compiles, and configures the Cg programs and then renders with them, this demo uses static texture image data included in the two header files.

The data in these header files are used to construct OpenGL texture objects for a brick pattern normal map 2D texture and a "vector normalization" cube map. These texture objects are sampled by the fragment program.

The data in the two headers files consists of hundreds of comma-separated numbers (I'll save you the tedium of publishing all the numbers in this article…). Rather than static data compiled into an executable, a typical application would read normal map textures from on-disk image files or convert a height-field image file to a normal map. Likewise, a "normalization vector" cube map is typically procedurally generated rather than loaded from static data.

The two Cg files each contain a Cg entry function with the same name as the file. These functions are explained in Chapter 8 (Bump Mapping) of *The Cg Tutorial*. These files are read by the demo when the demo begins running. The demo uses the Cg runtime to read, compile, configure, and render with these Cg programs.

Rather than rehash the background, theory, and operation of these Cg programs, you should consult Chapter 8 of *The Cg Tutorial*. Pages 200 to 204 explain the construction of the brick pattern normal map. Pages 206 to 208 explain the construction and application of a normalization cube map. Pages 208 to 211 explains specular bump mapping, including the **`C8E4f_specSurf`** fragment program. Pages 211 to 218 explain texture-space bump mapping. Pages 218 to 224 explain the construction of the per-vertex coordinate system needed for texture-space bump mapping for the special case of an object (the torus) that is generated from parametric equations by the **`C8E6v_torus`** vertex program.

For your convenience and so you can map Cg parameter names used in the C source file to their usage in the respective Cg programs, the complete contents of **`C8E6v_torus.cg`** and **`C8E4f_specSurf.cg`** are presented in Appendix A and Appendix B at the end of this article (the Cg programs are short, so why not).

On to the C Code

Now, it's time to dissect **`cg_bumpdemo.c`** line-by-line as promised (we'll skip comments in the source code if the comments are redundant with the discussion below).

To help you identify which names are external to the program, the following words are listed in **`boldface`** within the C code: C keywords; C standard library routines and macros; OpenGL, GLU, and GLUT routines, types, and enumerants; and Cg and CgGL runtime routines, types, and enumerants.

Initial Declarations

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <GL/glut.h>
#include <Cg/cg.h>
#include <Cg/cgGL.h>
```

The first three includes are basic ANSI C standard library includes. We'll be using **sin**, **cos**, **printf**, **exit**, and **NULL**. We rely on the GLUT header file to include the necessary OpenGL and OpenGL Utility Library (GLU) headers.

The **<Cg/cg.h>** header contains generic routines for loading and compiling Cg programs but does not contain routines that call the 3D programming interface to configure the Cg programs for rendering. The generic Cg routines begin with a **cg** prefix; the generic Cg types begin with a **CG** prefix; and the generic Cg macros and enumerations begin with a **CG_** prefix.

The **<Cg/cgGL.h>** contains the OpenGL-specific routines for configuring Cg programs for rendering with OpenGL. The OpenGL-specific Cg routines begin with a **cgGL** prefix; the OpenGL-specific Cg types begin with a **CGGL** prefix; and the OpenGL-specific Cg macros and enumerations begin with a **CGGL_** prefix.

Technically, the **<Cg/cgGL.h>** header includes **<Cg/cg.h>** so we don't have to explicitly include **<Cg/cg.h>** but we include both to remind you that we'll be calling both generic Cg routines and OpenGL-specific Cg routines.

```
/* An OpenGL 1.2 define */
#define GL_CLAMP_TO_EDGE                    0x812F

/* A few OpenGL 1.3 defines */
#define GL_TEXTURE_CUBE_MAP                 0x8513
#define GL_TEXTURE_BINDING_CUBE_MAP         0x8514
#define GL_TEXTURE_CUBE_MAP_POSITIVE_X      0x8515
```

We will use these OpenGL enumerants later when initializing our "normalization vector" cube map. We list them here explicitly since we can't count on **<GL/gl.h>** (included by **<GL/glut.h>** above) to have enumerants added since OpenGL 1.1 because Microsoft still supplies the dated OpenGL 1.1 header file.

Next, we'll list all global variables we plan to use. We use the **my** prefix to indicate global variables that we define (to make it crystal clear what names we are defining rather than those names defined by header files). When we declare a variable of a type defined by the Cg runtime, we use the **myCg** prefix to remind you that the variable is for use with the Cg runtime.

Cg Runtime Variables

```
static CGcontext   myCgContext;
static CGprofile   myCgVertexProfile,
                   myCgFragmentProfile;
static CGprogram   myCgVertexProgram,
                   myCgFragmentProgram;
static CGparameter myCgVertexParam_lightPosition,
                   myCgVertexParam_eyePosition,
                   myCgVertexParam_modelViewProj,
                   myCgVertexParam_torusInfo,
                   myCgFragmentParam_ambient,
                   myCgFragmentParam_LMd,
                   myCgFragmentParam_LMs,
                   myCgFragmentParam_normalMap,
                   myCgFragmentParam_normalizeCube,
                   myCgFragmentParam_normalizeCube2;
```

These are the global Cg runtime variables the demo initializes uses. We need a single Cg compilation context named **myCgContext**. Think of your Cg compilation context as the "container" for all the Cg handles you manipulate. Typically your program requires just one Cg compilation context.

We need two Cg profile variables, one for our vertex program profile named **myCgVertexProfile** and another for our fragment program profile named **myCgFragmentProfile**. These profiles correspond to a set of programmable hardware capabilities for vertex or fragment processing and their associated execution environment. Profiles supported by newer GPUs are generally more functional than older profiles. The Cg runtime makes it easy to select the most appropriate profile for your hardware as we'll see when we initialize these profile variables.

Next we need two Cg program handles, one for our vertex program named **myCgVertexProgram** and another for our fragment program named **myCgFragmentProgram**. When we compile a Cg program successfully, we use these handles to refer to the corresponding compiled program.

We'll need handles to each of the uniform input parameters used by our vertex and fragment programs respectively. We use these handles to match the uniform input parameters in the Cg program text with the opaque OpenGL state used to maintain the corresponding Cg program state. Different profiles can maintain Cg program state with different OpenGL state so these Cg parameter handles abstract away the details of how a particular profile manages a particular Cg parameter.

The **myCgVertexParam_** prefixed parameter handles end with each of the four uniform input parameters to the **C8E6v_torus** vertex program in Appendix A. Likewise, the **myCgFragmentParam_** prefixed parameter handles end with each of the six uniform input parameters to the **C8E4v_specSurf** fragment program in Appendix B.

In a real program, you'll probably have more Cg program handles than just two. You may have hundreds depending on how complicated the shading is in your application. And each program handle requires a Cg parameter handle for each input parameter. This means you probably won't want to use global variables to store these handles. You'll probably want to encapsulate your Cg runtime handles within "shader objects" that may well combine vertex and fragment Cg programs and their parameters within the same object for convenience. Keep in mind that this demo is trying to be very simple.

Other Variables

```
static const char *myProgramName = "cg_bumpdemo",
                  *myVertexProgramFileName = "C8E6v_torus.cg",
                  *myVertexProgramName = "C8E6v_torus",
                  *myFragmentProgramFileName = "C8E4f_specSurf.cg",
                  *myFragmentProgramName = "C8E4f_specSurf";
```

We need various string constants to identify our program name (for error messages and the window name), the names of the file names containing the text of the vertex and fragment Cg programs to load, and the names of the entry functions for each of these files.

In Appendix A, you'll find the contents of the **`C8E6v_torus.cg`** file and, within the file's program text, you can find the entry function named **`C8E6v_torus`**. In Appendix B, you'll find the contents of the **`C8E4f_specSurf.cg`** file and, within the file's program text, you can find the entry function name **`C8E4f_specSurf`**.

```
static float myEyeAngle = 0,
             myAmbient[4] = { 0.3f, 0.3f, 0.3f, 0.3f }, /* Dull white */
             myLMd[4] = { 0.9f, 0.6f, 0.3f, 1.0f },     /* Gold */
             myLMs[4] = { 1.0f, 1.0f, 1.0f, 1.0f };     /* Bright white */
```

These are demo variables used to control the rendering of the scene. The viewer rotates around the fixed torus. The angle of rotation and a degree of elevation for the viewer is determined by **`myEyeAngle`**, specified in radians. The other three variables provide lighting and material parameters to the fragment program parameters. With these particular values, the bump-mapped torus has a "golden brick" look.

Texture Data

```
/* OpenGL texture object (TO) handles. */
enum {
  TO_NORMALIZE_VECTOR_CUBE_MAP = 1,
  TO_NORMAL_MAP = 2,
};
```

The **`TO_`** prefixed enumerants provide numbers for use as OpenGL texture object names.

```
static const GLubyte
myBrickNormalMapImage[3*(128*128+64*64+32*32+16*16+8*8+4*4+2*2+1*1)] = {
/* RGB8 image data for mipmapped 128x128 normal map for a brick pattern */
#include "brick_image.h"
};

static const GLubyte
myNormalizeVectorCubeMapImage[6*3*32*32] = {
/* RGB8 image data for normalization vector cube map with 32x32 faces */
#include "normcm_image.h"
};
```

These static, constant arrays include the header files containing the data for the normal map's brick pattern and the "normalization vector" cube map. Each texel is 3 unsigned bytes (one for red, green, and blue). While each byte of the texel format is unsigned, normal map components, as well as the vector result of normalizing an arbitrary direction vector, are logically signed values within the [-1,1] range. To accommodate signed values with OpenGL's conventional **`GL_RGB8`** unsigned texture format, the unsigned [0,1] range is expanded in the fragment program to a signed [-1,1] range. This is the reason for the **`expand`** helper function called by the **`C8E4f_specSurf`** fragment program (see Appendix B).

The normal map has mipmaps so there is data for the 128x128 level, and then, each of the successively downsampled mipmap levels. The "normalization vector" cube map has six 32x32 faces.

Error Reporting Helper Routine

```
static void checkForCgError(const char *situation)
{
  CGerror error;
  const char *string = cgGetLastErrorString(&error);

  if (error != CG_NO_ERROR) {
    printf("%s: %s: %s\n",
      myProgramName, situation, string);
    if (error == CG_COMPILER_ERROR) {
      printf("%s\n", cgGetLastListing(myCgContext));
    }
    exit(1);
  }
}
```

Cg runtime routines report errors by setting a global error value. Calling the **`cgGetLastErrorString`** routine both returns a human-readable string describing the last generated Cg error and writes an error code of type **`CGerror`**. **`CG_NO_ERROR`** (defined to be zero) means there was no error. As a side-effect, **`cgGetLastErrorString`** also resets the global error value to **`CG_NO_ERROR`**. The Cg runtime also includes the simpler function **`cgGetError`** that just returns and then resets the global error code if you just want the error code and don't need a human-readable string too.

The **checkForCgError** routine is used to ensure proper error checking throughout the demo. Rather than cheap out on error checking, the demo checks for errors after essentially every Cg runtime call by calling **checkForCgError**. If an error has occurred, the routine prints an error message including the **situation** string and translated Cg error value string, and then exits the demo.

When the error returned is **CG_COMPILER_ERROR** that means there are compiler error messages too. So **checkForCgError** then calls **cgGetLastListing** to get a listing of the compiler error messages and prints these out too. For example, if your Cg program had a syntax error, you'd see the compiler's error messages including the line numbers where the compiler identified problems.

While "just exiting" is fine for a demo, real applications will want to properly handle any errors generated. In general, you don't have to be so paranoid as to call **cgGetLastErrorString** after every Cg runtime routine. Check the runtime API documentation for each routine for the reasons it can fail; when in doubt, check for failures.

Demo Initialization

```
static void display(void);
static void keyboard(unsigned char c, int x, int y);

int main(int argc, char **argv)
{
  const GLubyte *image;
  unsigned int size, level, face;
```

The **main** entry-point for the demo needs a few local variables to be used when loading textures. We also need to forward declare the **display** and **keyboard** GLUT callback routines for redrawing the demo's rendering window and handling keyboard events.

OpenGL Utility Toolkit Initialization

```
  glutInitWindowSize(400, 400);
  glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
  glutInit(&argc, argv);

  glutCreateWindow(myProgramName);
  glutDisplayFunc(display);
  glutKeyboardFunc(keyboard);
```

Using GLUT, we request a double-buffered RGB color 400x400 window with a depth buffer. We allow GLUT to take a pass parsing the program's command line arguments. Then, we create a window and register the **display** and **keyboard** callbacks. We'll explain these callback routines after completely initializing GLUT, OpenGL, and Cg. That's it for initializing GLUT except for calling **glutMainLoop** to start event processing at the very end of **main**.

OpenGL Rendering State Initialization

```
glClearColor(0.1, 0.3, 0.6, 0.0);  /* Blue background */
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(
  60.0,   /* Field of view in degree */
  1.0,    /* Aspect ratio */
  0.1,    /* Z near */
  100.0); /* Z far */
glMatrixMode(GL_MODELVIEW);
glEnable(GL_DEPTH_TEST);
```

Next, we initialize basic OpenGL rendering state. For better aesthetics, we change the background color to a nice sky blue. We specify a perspective projection matrix and enable depth testing for hidden surface elimination.

OpenGL Texture Object Initialization

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 1); /* Tightly packed texture data. */
```

By default, OpenGL's assumes each image scanline is aligned to begin on 4 byte boundaries. However, RGB8 data (3 bytes per pixel) is usually tightly packed to a 1 byte alignment is appropriate. That's indeed the case for the RGB8 pixels in our static arrays used to initialize our textures. If you didn't know about this OpenGL pitfall before, you do now.[‡]

Normal Map 2D Texture Initialization

```
glBindTexture(GL_TEXTURE_2D, TO_NORMAL_MAP);
/* Load each mipmap level of range-compressed 128x128 brick normal
   map texture. */
for (size = 128, level = 0, image = myBrickNormalMapImage;
     size > 0;
     size /= 2, image += 3*size*size, level++) {
  glTexImage2D(GL_TEXTURE_2D, level,
    GL_RGB8, size, size, 0, GL_RGB, GL_UNSIGNED_BYTE, image);
}
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
  GL_LINEAR_MIPMAP_LINEAR);
```

We bind to the texture object for our brick pattern normal map 2D texture and load each of the 7 mipmap levels, starting with the 128x128 base level and working down to the 1x1 level. Each level is packed into the `myBrickNormalMapImage` array right after the

---

[‡] Being aware of pitfalls such as this one can save you a lot of time debugging. This and other OpenGL pitfalls are enumerated in my article "Avoiding 19 Common OpenGL Pitfalls" found here http://developer.nvidia.com/object/Avoiding_Common_ogl_Pitfalls.html An earlier HTML version of the article (with just 16 pitfalls) is found here http://www.opengl.org/developers/code/features/KilgardTechniques/oglpitfall/oglpitfall.html

previous level. So the 64x64 mipmap level immediately follows the 128x128 level, and so on. OpenGL's default minification filter is "nearest mipmap linear" (again, a weird default—it means nearest filtering within a mipmap level and then bilinear filtering between the adjacent mipmap levels) so we switch to higher-quality "linear mipmap linear" filtering.

Normalize Vector Cube Map Texture Initialization

```
glBindTexture(GL_TEXTURE_CUBE_MAP, TO_NORMALIZE_VECTOR_CUBE_MAP);
/* Load each 32x32 face (without mipmaps) of range-compressed "normalize
   vector" cube map. */
for (face = 0, image = myNormalizeVectorCubeMapImage;
     face < 6;
     face++, image += 3*32*32) {
  glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + face, 0,
    GL_RGB8, 32, 32, 0, GL_RGB, GL_UNSIGNED_BYTE, image);
}
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S,
  GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T,
  GL_CLAMP_TO_EDGE);
```

Next, we bind the texture object for the "normalization vector" cube map[1] intended to quickly normalize the 3D lighting vectors that are passed as texture coordinates. The cube map texture has six faces but there's no need for mipmaps. Each face is packed into the **myNormalizeVectorCubeMapImage** array right after the prior face with the faces ordered in the order of the sequential texture cube map face OpenGL enumerants.

Again, the default minification state is inappropriate (this time because we don't have mipmaps) so **GL_LINEAR** is specified instead. While the default **GL_REPEAT** wrap mode was fine for the brick pattern that we intend to tile over the surface of the torus, the **GL_CLAMP_TO_EDGE** wrap mode (introduced by OpenGL 1.2) keeps one edge of a cube map face from bleeding over to the other.

GLUT and OpenGL are now initialized so it is time to begin loading, compiling, and configuring the Cg programs.

Cg Runtime Initialization

```
myCgContext = cgCreateContext();
checkForCgError("creating context");
```

---

[1] Using a "normalization vector" cube map allows our demo to work on older DirectX 8-class GPUs that lacked the shading generality to normalize vectors mathematically. Ultimately as more capable GPUs become ubiquitous, use of normalization cube maps is sure to disappear in favor of normalizing a vector mathematically. See Exercise 5.

Before we can do anything with the Cg runtime, we need to allocate a Cg compilation context with **cgCreateContext**. Typically, your application just needs one Cg compilation context unless you have a multi-threaded application that requires using the Cg runtime concurrently in different threads. Think of the Cg context as the context and container for all your Cg programs that are creating, loading (compiling), and configured by the Cg runtime.

Cg Vertex Profile Selection

```
myCgVertexProfile = cgGLGetLatestProfile(CG_GL_VERTEX);
cgGLSetOptimalOptions(myCgVertexProfile);
checkForCgError("selecting vertex profile");
```

We need a profile with which to compile our vertex program. We could hard-code a particular profile (for example, the multi-vendor **CG_PROFILE_ARBVP1** profile), but we are better off asking the CgGL runtime to determine the best vertex profile for our current OpenGL context by calling the **cgGLGetLatestProfile** routine. (Keep in mind there's a current OpenGL rendering context that GLUT created for us when we called **glutCreateWindow**.) **cgGLGetLatestProfile** calls OpenGL queries to examine the current OpenGL rendering context. Based on the OpenGL **GL_EXTENSIONS** string, this routine can decide what profiles are supported and then which hardware-supported profile offers the most functionality and performance. The **CG_GL_VERTEX** parameter says to return the most appropriate vertex profile, but we can also pass **CG_GL_FRAGMENT**, as we will do later, to determine the most appropriate fragment profile.

Cg supports a number of vertex profiles. These are the vertex profiles currently supported by Cg 1.4 for OpenGL: **CG_PROFILE_VP40** corresponds to the **vp40** vertex program profile for the **NV_vertex_program3** OpenGL extension (providing full access to the vertex processing features of NVIDIA's GeForce 6 Series GPUs such as vertex textures). **CG_PROFILE_VP30** corresponds to the **vp30** vertex program profile for the **NV_vertex_program2** OpenGL extension (providing full access to the vertex processing features of NVIDIA's GeForce FX GPUs such as per-vertex dynamic branching). **CG_PROFILE_ARBVP1** corresponds to the **arbvp1** vertex program profile for the **ARB_vertex_program** OpenGL extension (a multi-vendor OpenGL standard, supported by both NVIDIA and ATI). **CG_PROFILE_VP20** corresponds to the **vp20** vertex program profile for the **NV_vertex_program** and **NV_vertex_program1_1** OpenGL extensions (for NVIDIA's GeForce3, GeForce4 Ti, and later GPUs).

While several GPUs can support the same profile, there may be GPU-specific techniques the Cg compiler can use to make the most of the available functionality and generate better code for your given GPU. By calling **cgGLSetOptimalOptions** with the profile we've selected, we ask the compiler to optimize for the specific hardware underlying our OpenGL rendering context.

For example, some vertex profiles such as **CG_PROFILE_VP40** support texture fetches but typically support fewer texture image units than the hardware's corresponding fragment-

level texture functionality. **cgGLSetOptimalOptions** informs the compiler what the hardware's actual vertex texture image unit limit is.

Vertex Program Creation and Loading

```
myCgVertexProgram =
  cgCreateProgramFromFile(
    myCgContext,              /* Cg runtime context */
    CG_SOURCE,                /* Program in human-readable form */
    myVertexProgramFileName,  /* Name of file containing program */
    myCgVertexProfile,        /* Profile to try */
    myVertexProgramName,      /* Entry function name */
    NULL);                    /* No extra compiler options */
checkForCgError("creating vertex program from file");
cgGLLoadProgram(myCgVertexProgram);
checkForCgError("loading vertex program");
```

Now we try to create and load the Cg vertex program. We use the optimal vertex profile for our OpenGL rendering context to compile the vertex program contained in the file named by **myVertexProgramFileName**. As it turns out, the **C8E6v_torus** vertex program is simple enough that every Cg vertex profile mentioned in the last section is functional enough to compile the **C8E6v_torus** program.

The **cgCreateProgramFromFile** call reads the file, parses the contents, and searches for the entry function specified by **myVertexProgramName** and, if found, creates a vertex program for the profile specified by **myCgVertexProfile**. The **cgCreateProgramFromFile** routine is a generic Cg runtime routine so it just creates the program without actually translating the program into a form that can be passed to the 3D rendering programming interface.

You don't actually need a current OpenGL rendering context to call **cgCreateProgramFromFile**, but you do need a current OpenGL rendering context that supports the profile of the program for **cgGLLoadProgram** to succeed.

It is the OpenGL-specific **cgGLLoadProgram** routine that translates the program into a profile-dependent form. For example, in the case of the multi-vendor **arbvp1** profile, this includes calling the **ARB_vertex_program** extension routine **glProgramStringARB** to create an OpenGL program object.

We expect **cgGLLoadProgram** to "just work" because we've already selected a profile suited for our GPU and **cgCreateProgramFromFile** successfully compiled the Cg program into a form suitable for that profile.

Vertex Program Parameter Handles

```
myCgVertexParam_lightPosition =
  cgGetNamedParameter(myCgVertexProgram, "lightPosition");
checkForCgError("could not get lightPosition parameter");

myCgVertexParam_eyePosition =
  cgGetNamedParameter(myCgVertexProgram, "eyePosition");
checkForCgError("could not get eyePosition parameter");

myCgVertexParam_modelViewProj =
  cgGetNamedParameter(myCgVertexProgram, "modelViewProj");
checkForCgError("could not get modelViewProj parameter");

myCgVertexParam_torusInfo =
  cgGetNamedParameter(myCgVertexProgram, "torusInfo");
checkForCgError("could not get torusInfo parameter");
```

Now that the vertex program is created and successfully loaded, we initialize all the Cg parameter handles. Later during rendering in the **`display`** callback, we will use these parameter handles to update whatever OpenGL state the compiled program associates with each parameter.

In this demo, we know *a priori* what the input parameter names are to keep things simple. If we had no special knowledge of the parameter names, we could use Cg runtime routines to iterate over all the parameter names for a given program (see the **`cgGetFirstParameter`**, **`cgGetNextParameter`**, and related routines—use these for Exercise 11 at the end of this article).

Cg Fragment Profile Selection

```
myCgFragmentProfile = cgGLGetLatestProfile(CG_GL_FRAGMENT);
cgGLSetOptimalOptions(myCgFragmentProfile);
checkForCgError("selecting fragment profile");
```

We select our fragment profile in the same manner we used to select our vertex profile. The only difference is we pass the **`CG_GL_FRAGMENT`** parameter when calling **`cgGLGetLatestProfile`**.

Cg supports a number of fragment profiles. These are the fragment profiles currently supported by Cg 1.4 for OpenGL: **`CG_PROFILE_FP40`** corresponds to the **`fp40`** vertex program profile for the **`NV_fragment_program2`** OpenGL extension (providing full access to the fragment processing features of NVIDIA's GeForce 6 Series GPUs such as per-fragment dynamic branching). **`CG_PROFILE_FP30`** corresponds to the **`fp30`** vertex program profile for the **`NV_fragment_program`** OpenGL extension (providing full access to the fragment processing features of NVIDIA's GeForce FX GPUs). **`CG_PROFILE_ARBFP1`** corresponds to the **`arbfp1`** fragment program profile for the **`ARB_fragment_program`** OpenGL extension (a multi-vendor OpenGL standard, supported by both NVIDIA and ATI). **`CG_PROFILE_FP20`** corresponds to the **`fp20`** vertex program profile for the **`NV_texture_shader`**, **`NV_texture_shader2`**,

NV_register_combiners, and **NV_register_combiners2** OpenGL extensions (for NVIDIA's GeForce3, GeForce4 Ti, and later GPUs).

As in the vertex profile case, **cgGLSetOptimalOptions** informs the compiler about specific hardware limits relevant to fragment profiles. For example, when your OpenGL implementation supports the **ATI_draw_buffers** extension, the **cgGLSetOptimalOptions** informs the compiler of this fact so the compiler can know how many color buffers are actually available when compiling for fragment profiles that support output multiple color buffers. Other limits such as the **ARB_fragment_program** limit on texture indirections are likewise queried so the compiler is aware of this limit. The maximum number of texture indirections the GPU can support may require the compiler to re-schedule the generated instructions around this limit. Other profile limits include the number of texture image units available, the maximum number of temporaries and constants allowed, and the static instruction limit.

Fragment Program Creation and Loading

```
myCgFragmentProgram =
  cgCreateProgramFromFile(
    myCgContext,                /* Cg runtime context */
    CG_SOURCE,                  /* Program in human-readable form */
    myFragmentProgramFileName,  /* Name of file containing program */
    myCgFragmentProfile,        /* Profile to try */
    myFragmentProgramName,      /* Entry function name */
    NULL);                      /* No extra compiler options */
checkForCgError("creating fragment program from file");
cgGLLoadProgram(myCgFragmentProgram);
checkForCgError("loading fragment program");
```

We create and load the fragment program in much the same manner as the vertex program.

Fragment Program Parameter Handles

```
myCgFragmentParam_ambient =
  cgGetNamedParameter(myCgFragmentProgram, "ambient");
checkForCgError("getting ambient parameter");

myCgFragmentParam_LMd =
  cgGetNamedParameter(myCgFragmentProgram, "LMd");
checkForCgError("getting LMd parameter");

myCgFragmentParam_LMs =
  cgGetNamedParameter(myCgFragmentProgram, "LMs");
checkForCgError("getting LMs parameter");

myCgFragmentParam_normalMap =
  cgGetNamedParameter(myCgFragmentProgram, "normalMap");
checkForCgError("getting normalMap parameter");

myCgFragmentParam_normalizeCube =
  cgGetNamedParameter(myCgFragmentProgram, "normalizeCube");
checkForCgError("getting normalizeCube parameter");
```

```
myCgFragmentParam_normalizeCube2 =
  cgGetNamedParameter(myCgFragmentProgram, "normalizeCube2");
checkForCgError("getting normalizeCube2 parameter");
```

We initialize input parameter handles in the same manner as done for vertex parameter handles.

Setting OpenGL Texture Objects for Sampler Parameters

```
cgGLSetTextureParameter(myCgFragmentParam_normalMap,
  TO_NORMAL_MAP);
checkForCgError("setting normal map 2D texture");

cgGLSetTextureParameter(myCgFragmentParam_normalizeCube,
  TO_NORMALIZE_VECTOR_CUBE_MAP);
checkForCgError("setting 1st normalize vector cube map");

cgGLSetTextureParameter(myCgFragmentParam_normalizeCube2,
  TO_NORMALIZE_VECTOR_CUBE_MAP);
checkForCgError("setting 2nd normalize vector cube map");
```

Parameter handles for sampler parameters need to be associated with OpenGL texture objects. The first **cgGLSetTextureParameter** call associates the **TO_NORMAL_MAP** texture object with the **myCgFragmentParam_normalMap** parameter handle.

Notice how the **TO_NORMALIZE_VECTOR_CUBE_MAP** texture object is associated with the *two* distinct sampler parameters, **normalizeCube** and **normalizeCube2**. The reason this is done is to support older DirectX 8-class hardware such as the GeForce3 and GeForce4 Ti. These older DirectX 8-class GPUs must sample the texture associated with a given texture unit and that unit's corresponding texture coordinate set (and *only* that texture coordinate set). In order to support DirectX 8-class profiles (namely, **fp20**), the **C8E4f_specSurf** fragment program is written in such a way that the texture units associated with the two 3D vectors to be normalized (**lightDirection** and **halfAngle**) are each bound to the same "normalization vector" cube map. If there was no desire to support older DirectX 8-class hardware, fragment programs targeting the more general DirectX 9-class profiles (namely, **arbfp1** and **fp30**) could simply sample a single "normalization vector" texture unit.

Alternatively, the Cg fragment program could normalize the 3D lighting vectors with the **normalize** Cg standard library routine (see Exercise 5 at the end of this article), but for a lot of current hardware, a "normalization vector" cube map is faster and the extra precision for a mathematical normalize function is not crucial for lighting.

Start Event Processing

```
  glutMainLoop();
  return 0;  /* Avoid a compiler warning. */
}
```

GLUT, OpenGL, and Cg are all initialized now so we can start GLUT event processing. This routine never returns. When a redisplay of the GLUT window created earlier is needed, the **display** callback is called. When a key press occurs in the window, the **keyboard** callback is called.

Displaying the Window

Earlier in the code, we forward declared the **display** callback. Now it's time to discuss what the **display** routine does and how exactly we render our bump-mapped torus using the textures and Cg vertex and fragment programs we've loaded.

Rendering a 2D Mesh to Generate a Torus

In the course of updating the window, the **display** callback invokes the **drawFlatPatch** subroutine. This subroutine renders a flat 2D mesh with immediate-mode OpenGL commands.

```
/* Draw a flat 2D patch that can be "rolled & bent" into a 3D torus by
   a vertex program. */
void
drawFlatPatch(float rows, float columns)
{
  const float m = 1.0f/columns;
  const float n = 1.0f/rows;
  int i, j;

  for (i=0; i<columns; i++) {
    glBegin(GL_QUAD_STRIP);
    for (j=0; j<=rows; j++) {
      glVertex2f(i*m, j*n);
      glVertex2f((i+1)*m, j*n);
    }
    glVertex2f(i*m, 0);
    glVertex2f((i+1)*m, 0);
    glEnd();
  }
}
```

The mesh consists of a number of adjacent quad strips. The **C8E6v_torus** vertex program will take these 2D vertex coordinates and use them as parametric coordinates for evaluating the position of vertices on a torus.

Nowadays it's much faster to use OpenGL vertex arrays, particularly with vertex buffer objects, to render geometry, but for this simple demo, immediate mode rendering is easier.

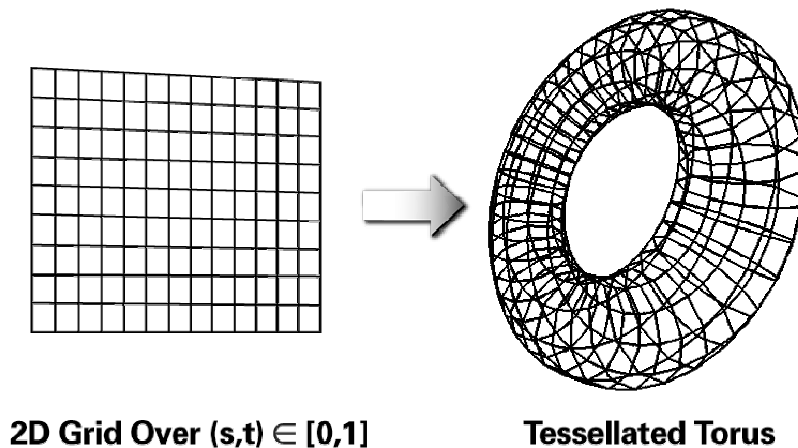

Figure 8-7 from *The Cg Tutorial* is replicated to illustrate how a 2D mesh could be procedurally “rolled and bent” into a torus by a vertex program.

The Display Callback

```
static void display(void)
{
  const float outerRadius = 6, innerRadius = 2;
  const int sides = 20, rings = 40;
  const float eyeRadius = 18.0;
  const float eyeElevationRange = 8.0;
  float eyePosition[3];

  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

The **display** callback has a number of constants that control the torus size and tessellation and how the torus is viewed.

```
  eyePosition[0] = eyeRadius * sin(myEyeAngle);
  eyePosition[1] = eyeElevationRange * sin(myEyeAngle);
  eyePosition[2] = eyeRadius * cos(myEyeAngle);

  glLoadIdentity();
  gluLookAt(
    eyePosition[0], eyePosition[1], eyePosition[2],
    0.0 ,0.0,  0.0,   /* XYZ view center */
    0.0, 1.0,  0.0);  /* Up is in positive Y direction */
```

The viewing transform is re-specified each frame. The eye position is a function of **myEyeAngle**. By animating this variable, the viewer rotates around the torus with a sinusoidally varying elevation. Because specular bump mapping is view-dependent, the specular lighting varies over the torus as the viewer rotates around.

Binding, Configuring, and Enabling the Vertex Program

```
cgGLBindProgram(myCgVertexProgram);
checkForCgError("binding vertex program");

cgGLSetStateMatrixParameter(myCgVertexParam_modelViewProj,
                            CG_GL_MODELVIEW_PROJECTION_MATRIX,
                            CG_GL_MATRIX_IDENTITY);
checkForCgError("setting modelview-projection matrix");
cgGLSetParameter3f(myCgVertexParam_lightPosition, -8, 0, 15);
checkForCgError("setting light position");
cgGLSetParameter3fv(myCgVertexParam_eyePosition, eyePosition);
checkForCgError("setting eye position");
cgGLSetParameter2f(myCgVertexParam_torusInfo, outerRadius, innerRadius);
checkForCgError("setting torus information");

cgGLEnableProfile(myCgVertexProfile);
checkForCgError("enabling vertex profile");
```

Prior to rendering the 2D mesh, we must bind to the vertex program, set the various input parameters used by the program with the parameter handles, and then enable the particular profile. Underneath the covers of these OpenGL-specific Cg routines, the necessary OpenGL commands are invoked to configure the vertex program with its intended parameter values.

Rather than specifying the parameter value explicitly as with the **cgGLSetParameter** routines, the **cgGLSetStateMatrixParameter** call binds the current composition of the modelview and projection matrices (specified earlier by **gluLookAt** and **gluPerspective** commands respectively) to the **modelViewProj** parameter.

One of the really nice things about the CgGL runtime is it saves you from having to know the details of what OpenGL routines are called to configure use of your Cg vertex and fragment programs. Indeed, the required OpenGL commands can very considerably between different profiles.

Binding, Configuring, and Enabling the Fragment Program

```
  cgGLBindProgram(myCgFragmentProgram);
  checkForCgError("binding fragment program");

  cgGLSetParameter4fv(myCgFragmentParam_ambient, myAmbient);
  checkForCgError("setting ambient");
  cgGLSetParameter4fv(myCgFragmentParam_LMd, myLMd);
  checkForCgError("setting diffuse material");
  cgGLSetParameter4fv(myCgFragmentParam_LMs, myLMs);
  checkForCgError("setting specular material");

  cgGLEnableTextureParameter(myCgFragmentParam_normalMap);
  checkForCgError("enable texture normal map");
  cgGLEnableTextureParameter(myCgFragmentParam_normalizeCube);
  checkForCgError("enable 1st normalize vector cube map");
  cgGLEnableTextureParameter(myCgFragmentParam_normalizeCube2);
  checkForCgError("enable 2nd normalize vector cube map");

  cgGLEnableProfile(myCgFragmentProfile);
  checkForCgError("enabling fragment profile");
```

The fragment program is bound, configured, and enabled in much the same manner with the additional task of enabling texture parameters with **`cgGLEnableTextureParameter`** to ensure the indicated texture objects are bound to the proper texture units.

Without you having to know the details, **`cgGLEnableTextureParameter`** calls **`glActiveTexture`** and **`glBindTexture`** to bind the correct texture object (specified earlier with **`cgGLSetTextureParameter`**) into the compiled fragment program's appropriate texture unit in the manner required for the given profile.

Render the 2D Mesh

```
  drawFlatPatch(sides, rings);
```

With the vertex and fragment program each configured properly, now render the flat 2D mesh that will be formed into a torus and illuminated with specular and diffuse bump mapping.

Disable the Profiles and Swap

```
  cgGLDisableProfile(myCgVertexProfile);
  checkForCgError("disabling vertex profile");

  cgGLDisableProfile(myCgFragmentProfile);
  checkForCgError("disabling fragment profile");

  glutSwapBuffers();
}
```

While not strictly necessary for this demo because just one object is rendered per frame, after rendering the 2D mesh, the profiles associated with the vertex program and

fragment program are each disabled. This way you could perform conventional OpenGL rendering. After using the OpenGL-specific Cg runtime, be careful not to assume how OpenGL state such as what texture objects are bound to what texture units.

Keyboard Processing

Along with the **`display`** callback, we also forward declared and registered the **`keyboard`** callback. Now it's time to see how the demo responses to simple keyboard input.

Animating the Eye Position

```
static void advanceAnimation(void)
{
  myEyeAngle += 0.05f;
  if (myEyeAngle > 2*3.14159)
    myEyeAngle -= 2*3.14159;
  glutPostRedisplay();
}
```

In order to animate the changing eye position so the view varies, the **`advanceAnimation`** callback is registered as the GLUT idle function. The routine advances **`myEyeAngle`** and posts a request for GLUT to redraw the window with **`glutPostRedisplay`**. GLUT calls the idle function repeatedly when there are no other events to process.

The Keyboard Callback

```
static void keyboard(unsigned char c, int x, int y)
{
  static int animating = 0;

  switch (c) {
  case ' ':
    animating = !animating; /* Toggle */
    glutIdleFunc(animating ? advanceAnimation : NULL);
    break;
```

The space bar toggles animation of the scene by registering and de-registering the **`advanceAnimation`** routine as the idle function.

```
  case 27:  /* Esc key */
    cgDestroyProgram(myCgVertexProgram);
    cgDestroyProgram(myCgFragmentProgram);
    cgDestroyContext(myCgContext);
    exit(0);
    break;
  }
}
```

The Esc key exits the demo. While it is not necessary to do so, the calls to **`cgDestroyProgram`** and **`cgDestroyContext`** deallocate the Cg runtime objects, along with their associated OpenGL state.

The Demo in Action

The images below show the rendered bump-mapped torus initially (left) and while animating (right).

Conclusions

This tutorial presents a complete Cg bump mapping demo written in ANSI C and rendering with OpenGL, relying on two of the actual Cg vertex and fragment programs detailed in *The Cg Tutorial*. I hope this tutorial "fills in the gaps" for those intrepid *Cg Tutorial* readers now inspired to integrate Cg technology into their graphics application. The **`cg_bumpdemo`** demo works on ATI and NVIDIA GPUs (and GPUs from any other vendor that support the standard, multi-vendor vertex and fragment program extensions). The demo is cross-platform as well, supporting Windows, OS X, and Linux systems.

The time you invest integrating the Cg runtime to your graphics application is time well spent because of the productivity and cross-platform support you unleash by writing shaders in Cg rather than resorting to low-level 3D rendering commands or a high-level shading language tied to a particular 3D API. With Cg, you can write shaders that work with two implementations of the same basic language (Cg & HLSL), two 3D rendering programming interfaces (OpenGL & Direct3D), three operating systems (Windows, OS X, and Linux), and the two major GPU vendors (ATI & NVIDIA—and any other vendors supporting DirectX 9-level graphics functionality).

Finally, Cg has evolved considerably since Randy and I wrote *The Cg Tutorial*. Cg 1.2 introduced a "sub-shader" facility allowing you to write shaders in Cg in a more modular fashion. And be sure to explore Cg 1.4's updated implementation of the CgFX meta-shader format (compatible with Microsoft's DirectX 9 FX format) to encapsulate non-programmable state, semantics, hardware-dependent rendering techniques, and support for multiple passes.

Exercises

Just as *The Cg Tutorial* provides exercises at the end of each chapter, here are some exercises to help you expand on what you've learned.

Improving the Shading

1. Support two lights. You'll need a second light position uniform parameter and your updated vertex program must output a second tangent-space light position. Example 5-4 in *The Cg Tutorial* will give you some ideas for supporting multiple lights. However, Example 5-4 is for two *per-vertex* lights; for this exercise, you want two *per-fragment* lights combined with bump mapping. *Hint:* If you add multiple lights, you might want to adjust down the values of `ambient`, `LMd`, and `LMs` to avoid an "over bright" scene.
2. Support a *positional* light (the current light is directional). Add controls so you can interactively position the light in the "hole" of the torus. Section 5.5 of *The Cg Tutorial* briefly explains the distinction between directional and positional lights.
3. Add geometric self-shadowing to the fragment program.
    a. Clamp the specular to zero if the *z* component of the tangent-space light direction is non-positive to better simulate self-shadowing (this is a situation where the light is "below" the horizon of the torus surface). See section 8.5.3 of *The Cg Tutorial* for details about geometric self-shadowing.
    b. Further tweak the geometric self-shadowing. Instead of clamping, modulate with `saturate(8*lightDirection.z)` so specular highlights don't "wink off" when self-shadowing occurs but rather drop off. When the scene animates, which approach looks better?
4. Change the specular exponent computation to use the `pow` standard library function instead of successive multiplication (you'll find `pow` is only available on more recent DirectX 9-class profiles such as `arbfp1` and `fp30`, not `fp20`). Provide the specular exponent as a uniform parameter to the fragment program.
5. Instead of using normalization cube maps, use the `normalize` standard library routine? Does the lighting change much? Does the performance change?
6. Rather than compute the tangent-space half-angle vector at each vertex and interpolate the half-angle for each fragment, compute the view vector at each vertex; then compute the half-angle at each fragment (by normalizing the sum of the interpolated normalized light vector and the interpolated normalized view vector). Does the lighting change much? Does the performance change?
7. **Advanced:** Read section 8.4 of *The Cg Tutorial* and implement bump mapping on an arbitrary textured polygonal mesh. Implement this approach to bump map an arbitrary textured model.

8. **Advanced:** Read section 9.4 of *The Cg Tutorial* and combine bump mapping with shadow mapping.

Improving the Cg Runtime Usage

9. Provide command line options to specify what file names contain the vertex and fragment programs.
10. Provide better diagnostic messages when errors occur.
11. Use the Cg runtime to query the uniform parameter names and then prompt the user for values for the various parameters (rather than having the parameter names and values hard coded in the program itself).
12. Rather than using global variables for each vertex and fragment program object, support loading a set of vertex and fragment programs and allow the user to select the current vertex and current fragment program from an interactive menu.

Appendix A: C8E6v_torus.cg Vertex Program

```
void C8E6v_torus(float2 parametric : POSITION,

             out float4 position       : POSITION,
             out float2 oTexCoord      : TEXCOORD0,
             out float3 lightDirection : TEXCOORD1,
             out float3 halfAngle      : TEXCOORD2,

         uniform float3 lightPosition,  // Object-space
         uniform float3 eyePosition,    // Object-space
         uniform float4x4 modelViewProj,
         uniform float2 torusInfo)
{
  const float pi2 = 6.28318530;  // 2 times Pi
  // Stetch texture coordinates counterclockwise
  // over torus to repeat normal map in 6 by 2 pattern
  float M = torusInfo[0];
  float N = torusInfo[1];
  oTexCoord = parametric * float2(-6, 2);
  // Compute torus position from its parameteric equation
  float cosS, sinS;
  sincos(pi2 * parametric.x, sinS, cosS);
  float cosT, sinT;
  sincos(pi2 * parametric.y, sinT, cosT);
  float3 torusPosition = float3((M + N * cosT) * cosS,
                                (M + N * cosT) * sinS,
                                N * sinT);
  position = mul(modelViewProj, float4(torusPosition, 1));
  // Compute per-vertex rotation matrix
  float3 dPds = float3(-sinS*(M+N*cosT), cosS*(M+N*cosT), 0);
  float3 norm_dPds = normalize(dPds);
  float3 normal = float3(cosS * cosT, sinS * cosT, sinT);
  float3 dPdt = cross(normal, norm_dPds);
  float3x3 rotation = float3x3(norm_dPds,
                               dPdt,
                               normal);
  // Rotate object-space vectors to texture space
  float3 eyeDirection = eyePosition - torusPosition;
  lightDirection = lightPosition - torusPosition;
  lightDirection = mul(rotation, lightDirection);
  eyeDirection = mul(rotation, eyeDirection);
  halfAngle = normalize(normalize(lightDirection) +
                        normalize(eyeDirection));
}
```

Appendix B: C8E4f_specSurf.cg Fragment Program

```
float3 expand(float3 v) { return (v-0.5)*2; }

void C8E4f_specSurf(float2 normalMapTexCoord : TEXCOORD0,
                    float3 lightDirection    : TEXCOORD1,
                    float3 halfAngle         : TEXCOORD2,

               out float4 color : COLOR,

           uniform float  ambient,
           uniform float4 LMd, // Light-material diffuse
           uniform float4 LMs, // Light-material specular
           uniform sampler2D   normalMap,
           uniform samplerCUBE normalizeCube,
           uniform samplerCUBE normalizeCube2)
{
  // Fetch and expand range-compressed normal
  float3 normalTex = tex2D(normalMap, normalMapTexCoord).xyz;
  float3 normal = expand(normalTex);
  // Fetch and expand normalized light vector
  float3 normLightDirTex = texCUBE(normalizeCube,
                                   lightDirection).xyz;
  float3 normLightDir = expand(normLightDirTex);
  // Fetch and expand normalized half-angle vector
  float3 normHalfAngleTex = texCUBE(normalizeCube2,
                                    halfAngle).xyz;
  float3 normHalfAngle = expand(normHalfAngleTex);

  // Compute diffuse and specular lighting dot products
  float diffuse = saturate(dot(normal, normLightDir));
  float specular = saturate(dot(normal, normHalfAngle));
  // Successive multiplies to raise specular to 8th power
  float specular2 = specular*specular;
  float specular4 = specular2*specular2;
  float specular8 = specular4*specular4;

  color = LMd*(ambient+diffuse) + LMs*specular8;
}
```

Comparison Tables for HLSL, OpenGL Shading Language, and Cg

April 2005

	DirectX 9 HLSL	OpenGL Shading Language	Cg Toolkit
Availability			
Available today	Yes	Yes	Yes
Installation/upgrade requirements	Part of DirectX 9; comes with XP or a free Windows updated	May need driver update from hardware vendor for ARB extensions or OpenGL 2.0	User level libraries so no driver upgrade typically required
Time of first release	March 2003, DirectX 9 ship	June 2003, ARB standards approved; implementation s in late 2003	December 2002, 1.0 release
Current version	DirectX 9.0c	1.10	Cg 1.4
Standard maker	Microsoft	OpenGL Architectural Review Board	NVIDIA
Implementer	Microsoft	Each OpenGL driver vendor	NVIDIA
3D Graphics API Support			
OpenGL	No	Yes	Yes
Direct3D	Yes	No	Yes
One shader can compile for either API	No	No	Yes

	DirectX 9 HLSL	OpenGL Shading Language	Cg Toolkit
OpenGL Specifics			
OpenGL 1.*x* support	n/a	Needs ARB GLSL extensions	Needs ARB or NV low-level assembly extensions
Multi-vendor ARB_vertex_program support	n/a	No	Yes
Multi-vendor ARB_fragment_program support	n/a	No	Yes
NVIDIA OpenGL extension support	n/a	No	Yes, profiles for fp20, vp20, fp30, vp30, fp40, and vp40
Relationship to OpenGL standard	n/a	Part of core OpenGL 2.0 standard	Layered upon ARB-approved assembly extensions
Access to OpenGL state settings	n/a	Yes	Yes
Open Source OpenGL rendering support (via Mesa)	n/a	No Mesa support yet	Yes, no changes required
Language tied to OpenGL	n/a	Yes	No, API-independent
Direct3D Specifics			
DirectX 8 support	Requires DirectX 9 upgrade but supports DirectX 8-class hardware profiles	n/a	Yes
DirectX 9 support	Yes	n/a	Yes
GPU Hardware Support			
NVIDIA DirectX 9-class GPUs	Yes	Yes	Yes
ATI DirectX 9-class GPUs	Yes	Yes	Yes
3Dlabs DirectX 9-class GPUs	Yes	Yes	Yes
DirectX 8-class GPUs	Yes, with ps1.x and vs1.1 profiles	No	Yes, fp20 and vp20 profiles

	DirectX 9 HLSL	OpenGL Shading Language	Cg Toolkit
Graphics Hardware Feature Support			
Vertex textures	Yes, if hardware supports vs3.0 profile	Yes, if hardware supports	Yes, if hardware supports vp40 profile
Dynamic vertex branching	Yes, if hardware supports vs2.0 profile	Yes, if hardware supports	Yes, if hardware supports vp30 or vp40 profiles
Dynamic fragment branching	Yes, if hardware supports ps3.0 profile	Yes, if hardware supports	Yes, if hardware supports fp40 profile
Fragment depth output	Yes	Yes	Yes
Multiple render targets	Yes, if hardware supports	Yes, if hardware supports	Yes, if hardware supports fp40 profile
1D, 2D, 3D, and cube map texture support	Yes	Yes	Yes
Shadow mapping	Yes	Yes, but needs special shadow texture fetch routines to be well-defined	Yes
Point size output	Yes	Yes	Yes
Clipping support	Output clip coordinate(s)	Output clip position	Output clip coordinate(s)
Texture rectangles	No	Yes, when ARB_texture_rectangle is supported	Yes, when ARB_texture_rectangle is supported
Access to fragment derivatives	Yes, when supported by the fragment profile	Yes	Yes, when supported by the fragment profile
Front and back vertex color outputs for two-sided lighting	No	Yes	Yes, for OpenGL
Front facing fragment shader input	Yes	Yes	Yes, for fp40 profile
Window position fragment shader input	Yes, for ps2 and ps3 profiles	Yes	Yes, for all but fp20 profile

	DirectX 9 HLSL	OpenGL Shading Language	Cg Toolkit
Language Details			
C-like languages	Yes	Yes	Yes
Language compatibility with Microsoft's DirectX 9 HLSL	Yes	No	Yes
Vertex and fragment shaders written as separate programs	Yes	Yes	Yes
C operators with C precedence	Yes	Yes	Yes
C control flow (if, else, for, do while)	Yes	Yes	Yes
Vector data types	Yes	Yes	Yes
Fixed-point data type	No	No. has fixed reserved word	Yes, fixed
Matrix arrangement	Column major by default	Column major	Row major by default
Non-square matrices	Yes	No	Yes
Data type constructors	Yes	Yes	Yes
Structures and arrays	Yes	Yes	Yes
Function overloading	Yes	Yes	Yes
Function parameter qualifiers: in, out, and inout	Yes	Yes	Yes
Type qualifiers: uniform, const	Yes	Yes	Yes
Shader outputs readable	Yes	No	Yes
Texture samplers	Yes	Yes	Yes
Keyword discard to discard a fragment	Yes	Yes	Yes
C and C++ style comments	Yes	Yes	Yes
C preprocessor support	Yes	Yes	Yes
Vector write masking	Yes	No	Yes
Vector component and matrix swizzling	Yes	No	Yes
Vector ?: operator	Yes	No, Boolean only	Yes
Vector comparison operators	Yes	No, must use lessThan, etc. standard library routines	Yes
Semantic qualifiers	Yes	No	Yes
Array dimensionality	Multi-dimensional	1D only	Multi-dimensional
Un-sized arrays (dynamic sizing)	No	No	Yes

	DirectX 9 HLSL	OpenGL Shading Language	Cg Toolkit
Shader Linking Support			
Sub-shader support via interface mechanism	No	No, but keyword reserved	Yes, in Cg 2.0
Separate compilation and linking	D3DX utility library: Fragment Linker	Yes	No, but interfaces provide a structured form of program reconfiguration
Cross domain linking by varying name	No	Yes	No
Standard Library			
Standard function library	Yes	Yes	Yes
Standard function library compatibility with Microsoft's DirectX 9 HLSL	Yes	No	Yes
Operating System Support			
Windows support (98/2000/XP)	Yes	Yes	Yes
Legacy Windows 95 support	No	Yes	Yes
Legacy Windows NT 4.0 support	No	Yes	Yes
Linux	No	Yes	Yes
Mac OS X	No	Yes	Yes (since 1.2)

	DirectX 9 HLSL	OpenGL Shading Language	Cg Toolkit
Shader Meta File Format Support			
Standard shader meta file format	Yes (FX)	No	Yes (CgFX)
Shader meta file format compatible with DirectX 9 Effects format	Yes	n/a	Yes
Specification of multiple techniques for an effect	Yes	n/a	Yes
Virtual machine for CPU shader execution	Yes	No	Yes
Annotations	Yes	n/a	Yes
State assignments	Yes	n/a	Yes
Run-time API	Yes	n/a	Yes
Direct3D loader	Yes	n/a	Yes
OpenGL loader	n/a	n/a	Yes
Free shader meta file viewer available	FX Composer	n/a	FX Composer
Debugging			
Interactive shader debugger	Visual Studio .NET Shader Debugger	No	No
Documentation			
Specification or definitive documentation	MSDN Direct3D 9 documentation	*The OpenGL Shading Language* specification	*The Cg Language Specification*
Tutorial book	Various books	*OpenGL Shading Language* (Rost)	*The Cg Tutorial* (Fernando & Kilgard)
User's manual	MSDN Direct3D 9 documentation	No	*Cg User's Manual*
Japanese documentation available	Yes	No	Yes

	DirectX 9 HLSL	OpenGL Shading Language	Cg Toolkit
Example Code			
Sources of example code	Microsoft DirectX 9 SDK examples	ATI, 3Dlabs, NVIDIA SDK examples	Cg Tutorial examples, NVIDIA SDK examples
Miscellaneous			
Standalone command line compiler	Yes (fxc)	No	Yes (cgc)
Generates human-readable intermediate assembly code	Yes	No	Yes
Supports ILM's OpenEXR half-precision data type	Yes	Reserved word for half; NVIDIA supports it	Yes
Open source language parser available	No	Yes	Yes
Compiler tied into graphics driver	No	Yes, compiled result depends on end-user hardware and driver version	No
Multi-vendor extension mechanism	No	Yes	No
No-fee redistributable library	Yes, Windows only	n/a	Yes, all platforms
What its name stands for	High Level Shader Language	OpenGL Shading Language	C for Graphics

n/a = Not available *or* not applicable.

Based on available knowledge circa April 2005.