ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, “MATERIALS”) ARE BEING PROVIDED “AS IS.” NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent or patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. NVIDIA Corporation products are not authorized for use as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks
NVIDIA and the NVIDIA logo are trademarks of NVIDIA Corporation.
Microsoft, Windows, the Windows logo, and DirectX are registered trademarks of Microsoft Corporation.
OpenGL is a trademark of SGI.
Other company and product names may be trademarks of the respective companies with which they are associated.

Updates
Any changes, additions, or corrections will be posted at the NVIDIA Cg Web site:

http://developer.nvidia.com/Cg

Refer to this site often to keep up on the latest changes and additions to the Cg language.

Copyright
Copyright NVIDIA Corporation 2002
# Table of Contents

**Foreword** .......................................................................................................................................xiii

**Preface**............................................................................................................................................xv

  - Release Notes ................................................................................................................................................xvi
  - New with This Release ......................................................................................................................................xvi
  - Unimplemented Features and Other Compiler Limitations ..............................................................xvi
  - Online Updates ....................................................................................................................................... xvii

**Introduction to the Cg Language** ................................................................................................... 1

  - The Cg Language ...........................................................................................................................................2
  - Cg's Programming Model for GPUs .............................................................................................................2
  - Cg Language Profiles ...................................................................................................................................3
  - Declaring Programs in Cg ............................................................................................................................3
  - Program Inputs and Outputs .......................................................................................................................4
    - Two Kinds of Program Inputs ..................................................................................................................4
    - Varying Inputs to a Vertex Program .....................................................................................................4
    - Varying Outputs to and from Vertex Programs ...................................................................................6
    - Varying Outputs from Fragment Programs ...........................................................................................8
  - Uniform Inputs .......................................................................................................................................9
  - Working with Data ........................................................................................................................................10
    - Basic Data Types ...................................................................................................................................10
    - Type Conversions ..................................................................................................................................11
  - Structures ....................................................................................................................................................12
  - Arrays ..........................................................................................................................................................12
  - Statements and Operators ............................................................................................................................13
    - Control Flow ........................................................................................................................................13
    - Function Definitions and Function Overloading .................................................................................13
    - Arithmetic Operators from C ...............................................................................................................14
    - Multiplication Function .......................................................................................................................14
    - Vector Constructor ..............................................................................................................................15
<table>
<thead>
<tr>
<th>Cg Language Toolkit</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean and Comparison Operators ..........................................................................</td>
<td>15</td>
</tr>
<tr>
<td>Swizzle Operator ...................................................................................................</td>
<td>15</td>
</tr>
<tr>
<td>Write-Mask Operator .............................................................................................</td>
<td>16</td>
</tr>
<tr>
<td>Conditional Operator .............................................................................................</td>
<td>16</td>
</tr>
<tr>
<td>Texture Lookups in the NV30 Fragment Profile ....................................................</td>
<td>16</td>
</tr>
<tr>
<td>More Details ..........................................................................................................</td>
<td>17</td>
</tr>
</tbody>
</table>

**Cg Standard Library Functions** ........................................................................ 19

- **Structs Using Binding Semantics** ................................................................... 19
- Mathematical Functions ..................................................................................... 22
- Geometric Functions ......................................................................................... 25
- Texture Map Functions ..................................................................................... 26
- Derivative Functions ......................................................................................... 27

**Using the Cg Run-Time Library** ....................................................................... 29

- Introduction ...................................................................................................... 29
  - Capabilities Provided by the Run-Time API .................................................. 30
  - Key Concepts .................................................................................................. 30
    - Programs ...................................................................................................... 30
    - Parameters .................................................................................................. 31
    - Contexts ..................................................................................................... 32
  - Run-Time Setup ............................................................................................. 32
    - Adding a Program to a Context .................................................................. 33
    - General Parameter Management .................................................................. 34
  - Using the OpenGL Run-Time Library .............................................................. 35
    - Downloading Programs to the Hardware .................................................... 35
    - Preparing to Render Within an Application ............................................. 35
    - Managing Uniform Parameters ............................................................... 36
    - Managing Varying Parameters .................................................................... 38
    - Managing Textures and Texture Units ....................................................... 39
  - Using the Direct3D Run-Time Library ............................................................. 40
    - Managing Programs .................................................................................... 40
    - Managing Parameters ................................................................................ 41
    - Activating Programs .................................................................................. 42
    - Vertex Format ............................................................................................. 42
Texture Operations .............................................................................................................................. 44

A Brief Tutorial ..................................................................................................................................... 45

Loading the Workspace ......................................................................................................................... 45

Understanding simple.cg ....................................................................................................................... 46

Program Listing for simple.cg .............................................................................................................. 46

Definitions for Structures with Varying Data ....................................................................................... 48

Passing Arguments ............................................................................................................................... 48

Basic Transformations .......................................................................................................................... 49

Prepare for Lighting ............................................................................................................................... 49

Calculating the Vertex Color .................................................................................................................. 50

Further Experimentation ....................................................................................................................... 50

NV30 Sample Shaders .......................................................................................................................... 51

Improved Skinning ............................................................................................................................... 52

Description .......................................................................................................................................... 52

Vertex Shader Cg Source Code ............................................................................................................. 53

Improved Water ..................................................................................................................................... 55

Description .......................................................................................................................................... 55

Vertex Shader Cg Source Code ............................................................................................................. 56

Melting Paint .......................................................................................................................................... 59

Description .......................................................................................................................................... 59

Vertex Shader Cg Source Code ............................................................................................................. 60

Pixel Shader Cg Source Code ............................................................................................................... 61

MultiPaint .............................................................................................................................................. 64

Description .......................................................................................................................................... 64

Vertex Shader Cg Source Code ............................................................................................................. 65

Pixel Shader Cg Source Code ............................................................................................................... 67

Ray-Traced Refraction ......................................................................................................................... 70

Description .......................................................................................................................................... 70

Vertex Shader Cg Source Code ............................................................................................................. 70

Pixel Shader Cg Source Code ............................................................................................................... 72

Skin ....................................................................................................................................................... 77

Description .......................................................................................................................................... 77

Pixel Shader Cg Source Code ............................................................................................................... 77
NV2X Sample Shaders

Anisotropic Lighting
Description
Vertex Shader Cg Source Code
Bump Dot3x2 Diffuse and Specular
Vertex Shader Cg Source Code
Pixel Shader Cg Source Code
Bump Reflection Mapping
Description
Vertex Shader Cg Source Code
Pixel Shader Cg Source Code
Fresnel
Description
Vertex Shader Cg Source Code
Grass
Description
Vertex Shader Cg Source Code
Refraction
Description
Vertex Shader Cg Source Code
Shadow Mapping
Vertex Shader Cg Source Code
Pixel Shader Cg Source Code
Shadow Volume Extrusion
Description
Vertex Shader Cg Source Code
Sine Wave Demo
Vertex Shader Cg Source Code
Matrix Palette Skinning
Description
Vertex Shader Cg Source Code

Appendix A. Cg Language Specification
Language Overview
New Vector Operators.......................................................................................................................... 131
Arithmetic Precision and Range ........................................................................................................... 133
Operator Precedence.............................................................................................................................. 133
Expanded Operators ............................................................................................................................. 133
Operators................................................................................................................................................. 134
  Boolean................................................................................................................................................ 134
  Comparisons....................................................................................................................................... 134
  Arithmetic........................................................................................................................................... 134
  Conditional Operator........................................................................................................................ 134
  Type Cast Operator........................................................................................................................... 135
  The C Comma Operator .................................................................................................................. 135
Reserved Words...................................................................................................................................... 135

Appendix B. Language Profiles ................................................................................................... 137
The DirectX 8 Vertex Shader Profile...................................................................................................... 137
  Binding Semantics.............................................................................................................................. 137
  Memory................................................................................................................................................ 138
  Statements and Operators..................................................................................................................... 138
  Variable Types ........................................................................................................................................ 139
  Using Conditionals................................................................................................................................. 139
  Using Arrays............................................................................................................................................ 139
  Constants................................................................................................................................................. 139
The DirectX 8 Pixel Shader Profile....................................................................................................... 140
  Data Types .............................................................................................................................................. 140
  Modifiers.................................................................................................................................................. 140
  Functions supported by dx8ps. ........................................................................................................... 142
  Texture Shader Operations................................................................................................................ 148
  Manual Assignment of Bindings............................................................................................................ 148
  Arithmetic Operations........................................................................................................................ 148
  Binding Semantics and Input/Output Varying Data ......................................................................... 149
  Uniform Parameters .......................................................................................................................... 149
  Other Language Constructs.................................................................................................................. 150
DirectX 8 Pixel Shader Examples ....................................................................................................... 150
  Example 1 ........................................................................................................................................... 150
Example 2 ........................................................................................................................................... 151

The \texttt{vp20} Vertex Program Profile ........................................................................................................ 152
Binding Semantics .................................................................................................................................. 152

The \texttt{vp30} Vertex Program Profile ........................................................................................................ 154
Binding Semantics .................................................................................................................................. 154

The \texttt{fp30} Fragment Program Profile .................................................................................................... 156
Binding Semantics .................................................................................................................................. 156

The ARB Vertex Program Profile ............................................................................................................ 158
Accessing OpenGL State ......................................................................................................................... 158
Binding Semantics .................................................................................................................................. 159
Compatibility with the \texttt{vp20} Vertex Program Profile .................................................................... 161
Loading Constants .................................................................................................................................. 161

\textbf{Appendix C. Cg Compiler Options} ............................................................................................... 163
Compiler Options ....................................................................................................................................... 163

\textbf{Appendix D. Color Plates} ............................................................................................................. 165
Color Illustrations ...................................................................................................................................... 165
Example of \texttt{simple.cg} Shader ........................................................................................................... 166
Example of Anisotropic Lighting ............................................................................................................. 166
Example of Dot3x2 Diffuse and Specular ............................................................................................... 167
Example of Bump Reflection Mapping .................................................................................................. 167
Example of Fresnel ................................................................................................................................ 168
Example of Grass ................................................................................................................................... 169
Example of Refraction ............................................................................................................................ 170
Example of Shadow Mapping ................................................................................................................ 170
Example of Shadow Volume Extrusion ................................................................................................. 171
Example of Sine Wave ........................................................................................................................... 172
Example of Matrix Palette Skinning ....................................................................................................... 172

\textbf{Appendix E. Converting to Binding Semantics} ........................................................................... 173

\textbf{Index} ........................................................................................................................................... 175
List of Figures

Figure 1. Cg’s Model of the GPU.................................................................2
Figure 2. The Cg_Simple Workspace....................................................45
Figure 3. The simple.cg Shader...............................................................46
Figure 4. Example of Improved Skinning...............................................52
Figure 5. Example of Improved Water....................................................55
Figure 6. Example of Melting Paint.........................................................59
Figure 7. Example of MultiPaint.............................................................64
Figure 8. Example of Ray-Traced Refraction...........................................70
Figure 9. Example of Skin......................................................................77
Figure 10. Example of Anisotropic Lighting............................................84
Figure 11. Example of Bump Dot3x2 Diffuse and Specular....................86
Figure 12. Example of Bump Reflection Mapping...................................90
Figure 13. Example of Fresnel.................................................................94
Figure 14. Example of Grass.................................................................96
Figure 15. Example of Refraction............................................................99
Figure 16. Example of Shadow Mapping.................................................102
Figure 17. Example of Shadow Volume Extrusion.................................105
Figure 18. Example of Sine Wave...........................................................108
Figure 19. Example of Matrix Palette Skinning.......................................111
Figure 20. The simple.cg Shader............................................................166
Figure 21. Example of Anisotropic Lighting Effect...................................166
Figure 22. Example of Bump Dot3x2 Diffuse and Specular....................167
Figure 23. Example of Bump Reflection Mapping...................................167
Figure 24. Example of Fresnel...............................................................168
Figure 25. Example of Grass.................................................................169
Figure 26. Example of Refraction............................................................170
Figure 27. Example of Shadow Mapping...............................................170
Figure 28. Example of Shadow Volume Extrusion ................................................................. 171
Figure 29. Example of Sine Wave.......................................................................................... 172
Figure 30. Example of Matrix Palette Skinning................................................................. 172
List of Tables

Table 1. Mathematical Functions ................................................................. 22
Table 2. Geometric Functions ..................................................................... 25
Table 3. Texture Map Functions ................................................................. 26
Table 4. Derivative Functions ..................................................................... 27
Table 5. dx8vs Input Binding Semantics ..................................................... 137
Table 6. dx8vs Output Binding Semantics .................................................. 138
Table 7. DirectX 8 Instruction Set Modifiers ............................................. 141
Table 8. dx8ps Profile Functions ................................................................. 142
Table 9. dx8ps Input Binding Semantics ..................................................... 149
Table 10. vp20 Input Binding Semantics ..................................................... 152
Table 11. vp20 Output Binding Semantics .................................................. 153
Table 12. vp30 Input Binding Semantics ..................................................... 154
Table 13. vp30 Output Binding Semantics .................................................. 155
Table 14. fp30 Input Binding Semantics ..................................................... 156
Table 15. arbvp1 Input Binding Semantics ................................................ 159
Table 16. arbvp1 Output Binding Semantics .............................................. 160
Table 17. Vertex Program Varying Inputs .................................................. 173
Table 18. Vertex Program Varying Outputs .............................................. 173
Table 19. Fragment Program Varying Inputs ............................................. 174
Table 20. Fragment Program Varying Outputs ......................................... 174
Table 21. Uniform Parameters ................................................................. 174
We are in the midst of a great transition in computer graphics, both in terms of graphics hardware and in terms of the visual quality and authoring process for games, interactive applications, and animation. Graphics hardware has evolved from “big iron” graphics workstations costing hundreds of thousands of dollars to single-chip graphics processing units (GPUs) whose performance and features have grown to match and now even to exceed traditional workstations. The processing power provided by a modern GPU in a single frame rivals the amount of computation that used to be expended for an offline-rendered animation frame. Indeed, at the launch of GeForce3 on the Apple Macintosh, a convincing version of Pixar’s Luxo, Jr. was demonstrated running interactively in real-time. At the 2001 SIGGRAPH conference, an interactive version of a more recent film, Square Studios’ Final Fantasy, was shown running in real-time, again on a GeForce3.

Although these feats of computation are astounding, there is much more to come. Today’s GPUs evolve very quickly. Typically, a product generation is only six months long, and with each new product generation comes a two-fold increase in performance. Graphics processor performance increases at approximately three times the rate of microprocessors—Moore’s Law cubed! In addition to the performance increases, each year brings new hardware features, supported by new application programming interfaces (APIs). This dizzying pace is difficult for developers to adapt to, but adapt they must.

Developers and users are demanding better rendering quality and more realistic imagery and experiences. Users don’t care about the details; they simply want games and other interactive applications to look more like movies, special effects, and animation. Developers want more power (always more), along with more flexibility in controlling the massively capable GPUs of today and tomorrow. APIs do not, and cannot, keep up with the rapid pace of innovation in GPUs. As APIs and underlying technologies change, programmers, artists, and software publishers struggle to adapt to the change and the churn of the hardware/software platform.

What’s needed is to raise the level of abstraction for interaction with GPUs. Continued updates and improvements to the hardware and APIs are too painful if developers are too “close to the metal.” This problem was exacerbated by the advent of programmability in GPUs. Older GPUs had a small number of controllable or configurable rendering paths, but the most recent technology is highly programmable, and becoming ever more so. We can now write short vertex and fragment programs to be executed by the GPU. This requires great skill, and is only possible with short programs.

When GPU hardware grows to allow programs of hundreds, thousands, or even more instructions, assembly coding will no longer be practical. Rather than programming each rendering state, each bit, byte, and word of data and control through a low-level
assembly language, we want to express our ideas in a more straightforward form, using a high-level language.

Thus Cg, C for Graphics, becomes necessary and inevitable. Just as C was derived to expose the specific capabilities of processors while allowing higher-level abstraction, Cg allows the same abstraction for GPUs. Cg changes the way programmers can program: focusing on the ideas, the concepts, and the effects they wish to create—not on the details of the hardware implementation. Cg also decouples programs from specific hardware because the language is functional, not hardware implementation-specific. Also, since Cg can be compiled at run time on any platform, operating system, and for any graphics hardware, Cg programs are truly portable. Finally, and perhaps best of all, Cg programs are future-proof and can adapt to run well on future products. The compiler can optimize directly for a new target GPU that perhaps did not even exist when the original Cg program was written.

This book is intended as an introduction to Cg, as well as a practical handbook to get programmers started developing in Cg. It includes a language description, a reference for the standard and run-time libraries, and is full of helpful examples. The goal for this book is to be both an introduction and a tool for the new user, as well as a reference and resource for developers as they become more proficient.

Welcome to the world of Cg!

David Kirk
Chief Scientist
NVIDIA Corporation
The goal of this book is to introduce to you Cg, a new high-level language for graphics programming. To that end, we have organized this document into the following sections:

- **Introduction to the Cg Language**  
  A quick introduction to the current release of Cg, with everything you need to know to start working it.

- **Cg Standard Library Functions**  
  A list of the Standard Library functions, which can help to reduce your program development time.

- **Using the Cg Run-Time Library**  
  An introduction to the Cg run-time APIs, which allow you to easily compile Cg programs and pass data to them from within an application.

- **A Brief Cg Tutorial**  
  A description of a simple Cg program and Microsoft Visual Studio workspace (both provided on the accompanying CD) that you can use to start experimenting with Cg.

- **Sample Shaders**  
  A list of sample Cg shaders, complete with source code.

- **Appendix A: The Cg Language Specification**  
  The formal Cg language specification.

- **Appendix B: Language Profiles**  
  Describes features and restrictions of the currently supported language profiles: DirectX 8 vertex; DirectX 8 pixel; OpenGL ARB vertex; NV2X OpenGL vertex; NV30 OpenGL vertex; and NV30 OpenGL fragment.

- **Appendix C: Cg Compiler Options**  
  A list of the various command-line options that the Cg compiler accepts.

- **Appendix D: Color Plates**  
  Color versions of the sample shader images.

- **Appendix E: Converting to Binding Semantics**  
  Gives correspondence of binding semantics names to `#pragma bind` registers.

- **Cg Developer’s CD**  
  The CD provided with this book contains the entire Cg release, which allows you get started immediately. The `readme.txt` file on the CD describes the contents of the release in detail.

You can begin working with Cg immediately by reading the Introduction to the Cg Language section beginning on page 1 and then going through A Brief Cg Tutorial (page 45). Once you have a basic understanding of the Cg language, use the Sample Shaders section (page 83) as a basis to build your own effects.
Release Notes

New with This Release

The additions and changes listed below have been made for this release of Cg.

- **Compiler changes**
  - Support for new profiles—arbvp1 (ARB_vertex_program), vp30 (NV_vertex_program2), and fp30 (NV_fragment_program)—has been added.
  - Support for binding semantics has been added. You are encouraged to use binding semantics notation to specify varying parameters for input and output.
  - Initialization of uniform globals is supported.
  - Initialization of static globals is supported.
  - Initialization of uniform parameters, except matrices, is supported.
  - The fp30 profile adds support for the fixed and half data types. To achieve maximum performance, you are strongly encouraged to use fixed and half instead of float.

- **Run-time changes**
  - The new functions cgAddProgramArgs() and cgAddProgramFromFileArgs() have been added. They are equivalent to cgAddProgram() and cgAddProgramFromFile() respectively, but take two additional arguments. These new functions allow the user to pass extra command-line options directly to the compiler. The two additional arguments are int nargs and const char **args. The nargs argument is the number of additional arguments, and args is an array of strings that contains the arguments.
  - A new function cgGetBindAddress() has been added. This function is useful for cgBindLocation's that need an additional integer address to uniquely identify a register (that is, cgBindc).
  - The new function cgBindIter *cgDuplicateBindIter(cgBindIter *bind) duplicates a given cgBindIter.
  - The new function cgProgramIter *cgGetLastProgram(cgContext *context) returns a cgProgramIter pointing to the last program that was successfully added to the context with cgAddProgram().
  - Two new profiles named vp30 and fp30 have been added. The enum values to specify them are cgVertex30Profile and cgFragmentProfile.

Unimplemented Features and Other Compiler Limitations

The current release of the compiler has these unimplemented features and limitations.

- **Limitations to be addressed in future releases**
  - Using binding semantics and #pragma bind in the same program may fail.
  - Initialization of structures is not supported.
  - Non-packed arrays are supported only in vertex profiles.
  - Structures with array members, and arrays of structures are supported only in vertex profiles.
Features not supported in any profile
- Function overloading by profile is not supported. (All other overloading is implemented.)
- Default values for function parameters cannot be used.
- Downcasting of vectors and matrices—such as casting a vector to a smaller vector or casting a matrix to a smaller matrix—is not possible.
- Preprocessor # or ## macro operators are not available.

Limitations of vertex shader profiles
- The if/else, while, for, and do statements are supported for vp30 profiles only.
- Indexing an array that is not declared const with an expression that is not constant is reported as an error. (It used to fail silently.)
- The % operator is not supported.
- General arrays (arrays other than vectors or matrices) are restricted to a single dimension and are read-only. In practice, arrays of floats, vectors, and matrices that are passed as uniform parameters to the top-level function are supported.
- Future releases will support for and while constructs in the dx8vs profile when the iteration count can be determined at compile time.
- Optimization is improved, but still not complete.
- The noise() standard library functions are not implemented and always return zero.
- The arbvp1 profile and run-time support for the ARB_vertex_program currently supports only program local parameters. Support for program environment parameters is planned for the future.

Features not implemented for the fp30 fragment program profile
- Uniform matrix parameters are not supported.
- General arrays are not supported.

Features not implemented for the DirectX 8 pixel profile
- Swizzling from the z (blue) component to create a scalar is not yet implemented. It will allow extracting the blue component of a register.
- Use of the depth-replace (DEPR) output is not available.
- Use of float2 is not yet supported.
- Optimization is poor in some cases.

Deprecated compiler functionality
- Although a few of the demo programs may use #pragma bind, this capability is now deprecated. You are encouraged to use binding semantics instead.
- The texobj* type is deprecated. Use the sampler* type instead.

Online Updates
Any changes, additions, or corrections will be posted at the NVIDIA Cg Web site

http://developer.nvidia.com/Cg

Refer to this site often to keep up on the latest changes and additions to the Cg language. Information on how to report any bugs you may find in the release is also available on this site.
Introduction to the Cg Language

Historically, graphics hardware has been programmed at a very low level. Fixed-function pipelines were configured by setting states such as the texture-combining modes. Programmers configured more recent programmable pipelines by using assembly-language-level programming interfaces. In theory, these low-level programming interfaces provided great flexibility. In practice, they were painful to use and presented a serious barrier to the effective use of hardware.

Using a high-level programming language, rather than the low-level languages of the past, provides several advantages:

- A high-level language speeds up the tweak-and-run cycle when a shader is developed. The ultimate test for a shader is “Does it look right?” To that end, the ability to quickly prototype and modify a shader is crucial to the rapid development of high-quality effects.
- The compiler optimizes code automatically and performs low-level tasks, such as register allocation, that are tedious and prone to error.
- Shading code written in a high-level language is much easier to read and understand. It also allows new shaders to be easily created by modifying previously written shaders. What better way to learn than from a shader written by the best artists and programmers?
- Shaders written in a high-level language are portable to a wider range of hardware platforms than shaders written in assembly code.

This chapter introduces Cg (C for Graphics), a new high-level language tailored for the programming of GPUs. Cg offers all the advantages just described, allowing programmers to finally combine the inherent power of the GPU with a language that makes GPU programming easy.
The Cg Language

Cg is based on C, but with enhancements and modifications that make it easy to write programs that compile to highly optimized GPU code. Cg code looks almost exactly like C code, with the same syntax for declarations, function calls, and most data types.

Before describing the Cg language in detail, it is important to explain the reason for some of the differences that exist between Cg and C. Fundamentally, it comes down to the difference in the programming models for GPUs and for CPUs.

Cg’s Programming Model for GPUs

CPUs normally have only one programmable processor. In contrast, GPUs have at least two programmable processors, the vertex processor and the fragment processor, plus other non-programmable hardware units. The processors, the non-programmable parts of the graphics hardware, and the application are all linked through data flows. Figure 1 illustrates Cg’s model of the GPU.

The Cg language allows you to write programs for both the vertex processors and the fragment processors. We refer to these programs as vertex programs and fragment programs, respectively. (Fragment programs are also known as pixel programs or pixel shaders, and in this document we use these terms interchangeably.) Cg code can be compiled into GPU assembly code, either on demand at run time or beforehand.

Cg makes it easy to combine a Cg fragment program with a handwritten vertex program, or even with the non-programmable OpenGL or DirectX vertex pipeline. Likewise, a Cg vertex program can be combined with a handwritten fragment program, or with the non-programmable OpenGL or DirectX fragment pipeline.
Cg Language Profiles

Because all CPUs support essentially the same set of basic capabilities, the C language supports this set on all CPUs. However, GPU programmability has not quite yet reached this same level of generality. For example, the current generation of programmable vertex processors supports a greater range of capabilities than do the programmable fragment processors. Cg addresses this issue by introducing the concept of language profiles. A Cg profile defines a subset of the full Cg language that is supported on a particular hardware platform or API. The current release of the Cg compiler supports the following profiles:

- **DirectX 8 vertex programs**
  - Run-time profile: `cgDX8VertexProfile`
  - Compiler option: `-profile dx8vs`
- **DirectX 8 pixel shaders**
  - Run-time profile: `cgDX8PixelProfile`
  - Compiler option: `-profile dx8ps`
- **OpenGL ARB vertex programs for the ARB_vertex_program extension**
  - Run-time profile: `cgARBVertexProfile`
  - Compiler option: `-profile arbv1`
- **OpenGL NV2X vertex programs for the NV_vertex_program extension**
  - Run-time profile: `cgVertexProfile`
  - Compiler option: `-profile vp20`
- **OpenGL NV30 vertex programs for the NV_vertex_program2 extension**
  - Run-time profile: `cgVertex30Profile`
  - Compiler option: `-profile vp30`
- **OpenGL NV30 fragment programs for the NV_fragment_program extension**
  - Run-time profile: `cgFragmentProfile`
  - Compiler option: `-profile fp30`

See Appendix B, Language Profiles, for detailed descriptions of these profiles.

Declaring Programs in Cg

CPU code generally consists of one program, specified by `main()` in C. In contrast, a Cg program can have any name. A program is defined using the following syntax:

```c
<return-type> <program-name>(<parameters>) { 
   ...
}
```
Program Inputs and Outputs

The programmable processors in GPUs operate on streams of data. For example, the vertex processor operates on a stream of vertices, and the fragment processor operates on a stream of fragments.

A programmer can think of the main program as being executed just once on a CPU. In contrast, a program is executed repeatedly on a GPU — once for each element of data in a stream. The vertex program is executed once for each vertex, and the fragment program is executed once for each fragment.

The Cg language adds several capabilities to C to support this stream-based programming model. For new Cg programmers, these capabilities often take some time to understand because they have no direct correspondence to C capabilities. However, the sample programs later in this document demonstrate that it really is easy to use these capabilities in Cg programs.

Two Kinds of Program Inputs

A Cg program can consume two different kinds of inputs:

- **Varying inputs**
  Varying inputs are used for data that is specified with each element of the stream of input data. For example, the varying inputs to a vertex program are the per-vertex values that are specified in vertex arrays. For a fragment program, the varying inputs are the interpolants, such as texture coordinates.

- **Uniform inputs**
  Uniform inputs are used for values that are specified separately from the main stream of input data, and don’t change with each stream element. For example, a vertex program typically requires a transformation matrix as a uniform input. Often, uniform inputs are thought of as graphics state.

The Cg compiler normally maps these two types of inputs to different sets of assembly-code registers. For example, when compiling to OpenGL for the NV2X vertex profile, the Cg compiler maps varying inputs to \( v \) registers, and uniform inputs to \( c \) registers.

Varying Inputs to a Vertex Program

A vertex program typically consumes several different per-vertex (varying) inputs. For example, a vertex program might require that the application specify the following varying inputs, typically in a vertex array:

- model space position for each vertex
- model space normal vector for each vertex
- texture coordinate for each vertex

In a fixed-function graphics pipeline, the set of possible per-vertex inputs is small and predefined. This predefined set of inputs is exposed to the application through the graphics API. For example, OpenGL 1.4 provides the ability to specify a vertex array of normal vectors.
But, in a programmable graphics pipeline, there is no longer a small set of predefined inputs. It is perfectly reasonable for the developer to write a vertex program that uses a per-vertex refractive index value as long as the application provides this value with each vertex.

Cg provides two different mechanisms for specifying these flexible per-vertex inputs.

The first mechanism provides a large set of predefined names. Each program input must be bound to a name from this set. For example, the following vertex-program definition binds its parameters to the predefined names POSITION, NORMAL, TANGENT, and TEXCOORD3. The application must provide vertex arrays for these predefined names.

```cpp
struct myinputs {
    float3 myPosition : POSITION;
    float3 myNormal : NORMAL;
    float3 myTangent : TANGENT;
    float  refractive_index : TEXCOORD3;
};

outdata foo(myinputs indata) {
    ...
    // within the program, the parameters are referred to as.
    //   'indata.myPosition', 'indata.myNormal', and so on.
    ...
};
```

We refer to the predefined names as binding semantics. The following set of binding semantics is supported in all Cg vertex-program profiles. Some Cg profiles support additional binding semantics.

- POSITION
- DIFFUSE
- SPECULAR
- BLENDWEIGHT
- NORMAL
- TANGENT
- BINORMAL
- TESSFACTOR
- PSIZE
- BLENDINDICES
- TEXCOORD0–TEXCOORD7

The binding semantic POSITION0 is equivalent to the binding semantic POSITION, and likewise for the other binding semantics.

In the OpenGL Cg profiles, binding semantics implicitly specify the mapping of varying inputs to particular hardware registers. However, in the DirectX-based Cg profile, there is no such implied mapping.

Binding semantics may also be specified directly on program parameters rather than on structure elements. Thus, the following vertex program definition is legal:
Cg also allows binding semantics to be omitted. When binding semantics are omitted from varying parameters, the application must refer to each parameter by its name rather than by its binding semantic. In the example below, the application provides vertex arrays for `myPosition, myNormal,` and so on. If binding semantics are omitted, all of the varying program parameters must appear in a single `struct` that must be annotated to indicate that it represents data being passed from the application to a vertex program:

```c
struct myinputs : application2vertex {
  float3 myPosition;
  float3 myNormal;
  float3 myTangent;
  float  refractive_index;
};
```

```c
outdata foo(myinputs indata) {
  ...
  // within the program, the parameters are referred to as...
  // 'indata.myPosition', 'indata.myNormal', etc.
  //
  ...}
```

### Varying Outputs to and from Vertex Programs

The outputs of a vertex program pass through the rasterizer and are made available to a fragment program as varying inputs. For a vertex program and fragment program to interoperate, they must agree on the data being passed between them.

As with the data flow between the application and vertex program, Cg provides two different mechanisms for specifying the data flow between the vertex and fragment programs. The first mechanism uses binding semantics. The second mechanism allows binding semantics to be omitted.
This example shows the use of binding semantics for vertex program output:

```c
// vertex program
struct myvf {
    float4 pout         : POSITION;
    float4 diffusecolor : COLOR0;
    float4 uv0          : TEXCOORD0;
    float4 uv1          : TEXCOORD1;
}
myout foo(...) {
    myvf outstuff;
    ...
    return outstuff;
}
```

And, this example shows how to use this same data as the input to a fragment program:

```c
// fragment program
struct myvf {
    float4 diffusecolor : COLOR0;
    float4 uv0          : TEXCOORD0;
    float4 uv1          : TEXCOORD1;
}
fragout bar(myvf indata) {
    float4 x = indata.uv0;
    ...
};
```

The following binding semantics are available in all Cg vertex profiles for output from vertex programs:

- POSITION
- PSIZE
- FOG
- COLOR0–COLOR1
- TEXCOORD0–TEXCOORD7

All vertex programs must have an output that uses the POSITION binding semantic because this information is needed for rasterization.

The following binding semantics are available in all Cg fragment profiles (except dx8ps) for input to fragment programs:

- COLOR0–COLOR1
- TEXCOORD0–TEXCOORD7

Cg optionally allows most binding semantics to be omitted. To ensure interoperability between vertex programs and fragment programs, both must use the same struct for their outputs and inputs respectively. For example,
struct myvf : vertex2fragment {
    float4 pout : POSITION; // Must always specify position
    float4 diffusecolor
    float4 uv0
    float4 uv1
}

// vertex program
myout foo(...) {
    myvf outstuff;
    ...
    return outstuff;
}

// fragment program
fragout bar(myvf indata) {
    float4 x = indata.uv0;
    ...
};

Varying Outputs from Fragment Programs

Binding semantics are always required on the outputs of fragment programs. The following binding semantics are available in all Cg fragment profiles:

- COLOR
- DEPTH

For example,

struct fragout {
    half4 col : COLOR;
    float depth: DEPTH;
}

fragout myfragprog(...) {
    fragout outval;
    outval.col = ...;
    outval.depth = ...
    return outval;
}

For convenience, the fragout struct is predefined by the Cg standard library and is therefore always available without having to explicitly define it.
Uniform Inputs

Cg vertex or fragment programs can also accept uniform inputs. For example, a Cg vertex program typically takes a modelview or projection matrix as one of its uniform inputs. A uniform input is specified using the `uniform` keyword, as shown.

```
myinterpolants foo(myvertarray vdata,
                  uniform float4x4 modelviewproj)
```

Uniform Input Example

The following example shows a simple vertex program that calculates diffuse and specular lighting. Two `struct` s for varying data, namely `appin` and `vertout`, are also declared. Don’t worry about understanding exactly what the program is doing— the goal here is simply to give you an idea of what Cg code looks like. A Brief Cg Tutorial starting on page 45 explains this shader in detail.

```
// simple.cg

// define inputs from application
struct appin
{
    float4 Position : POSITION;
    float4 Normal : NORMAL;
};

// define outputs from vertex shader
struct vertout
{
    float4 Hposition : POSITION;
    float4 Color0 : COLOR0;
};

vertout main(appin In,
             uniform float4x4 ModelViewProj : C0,
             uniform float4x4 ModelViewIT   : C4,
             uniform float4 LightVec)
{
    vertout Out;

    Out.Hposition = mul(ModelViewProj, In.Position);

    // transform normal from model space to view space
    float4 normal = normalize(mul(ModelViewIT, In.Normal).xyzz);

    // store normalized light vector
    float4 light = normalize(LightVec);

    // calculate half angle vector
    float4 eye = float4(0.0, 0.0, 1.0, 1.0);
```

float4 half = normalize(light + eye);

// calculate diffuse component
float diffuse = dot(normal, light);

// calculate specular component
float specular = dot(normal, half);
specular = pow(specular, 32);

// blue diffuse material
float4 diffuseMaterial = float4(0.0, 0.0, 1.0, 1.0);

// white specular material
float4 specularMaterial = float4(1.0, 1.0, 1.0, 1.0);

// combine diffuse and specular contributions
// and output final vertex color
Out.Color0 = diffuse * diffuseMaterial +
    specular * specularMaterial;

return Out;

Working with Data

Like C, Cg supports features that create and manipulate data:

- Basic types
- Structures
- Arrays
- Type conversions

Basic Data Types

Cg supports five basic data types:

- **float**
  A 32-bit IEEE floating point (s23e8). This type has one sign bit, a 23-bit mantissa, and an 8-bit exponent. This type is supported in all profiles, although the DirectX 8 pixel profile implements it with reduced precision and range for some operations.

- **half**
  A 16-bit IEEE-like floating point (s10e5). This type is only supported in the NV30 fragment profile.

- **fixed**
  A 12-bit fixed-point number (s1.10). This type is only supported in the NV30 fragment profile.
bool
Boolean data. It is produced by comparisons and is used in if and ? : constructs. This type is supported in all profiles.

sampler*
Handle to a texture object. This type comes in five variants: sampler1D, sampler2D, sampler3D, samplerCUBE, and samplerRECT. These types are supported in all pixel and fragment profiles, except that samplerRECT is not supported in the DirectX profiles.

Cg also includes built-in vector data types that are based on the basic data types. A sample of these built-in vector data types includes (but is not limited to) the following:

float4, float3, float2, float1
bool4, bool3, bool2, bool1

Additional support is provided for matrices of up to four-by-four elements. Here are some examples of matrix declarations:

float1x1 matrix1; // one element matrix
float2x3 matrix2; // two by three element matrix (six elements)
float4x2 matrix3; // four by two element matrix (eight elements)
float4x4 matrix4; // four by four element matrix (sixteen elements)

Note that the multi-dimensional array float M[4][4] is not type-equivalent to the matrix float4x4 M.

There is currently no int type. Most graphics hardware does not support an int type, so it is not included in the language. Generally, the float or bool types can be used instead of int. There are also no unions or bit fields in Cg at present.

Type Conversions
Type conversions in Cg work largely as they do in C. Type conversions may be explicitly specified using the C (newtype) cast operator.

Cg automatically performs type promotion in mixed-type expressions, just as C does. For example, the expression

floatvar * halfvar

is automatically compiled as

floatvar * (float) halfvar

Cg uses different type-promotion rules than C does in one case: A constant without an explicit type suffix does not cause type promotion. For example, the expression

halfvar * 2.0

is compiled as

halfvar * (half) 2.0
In contrast, C would compile it as \((\text{double}) \text{ halfvar}) \times 2.0\). Cg uses different rules than C to minimize the opportunity for inadvertent type promotions that cause computations to be performed in slower high-precision arithmetic. If the C behavior is desired, the constant should be explicitly typed, which forces the type promotion:

\[
\text{halfvar} \times 2.0f
\]
is compiled as
\[
((\text{float}) \text{ halfvar}) \times 2.0f
\]
Cg uses the following type suffixes for constants:

- \(f\) = float
- \(h\) = half
- \(x\) = fixed

**Structures**

Cg supports structures the same way as in C. Cg adopts the C++ convention of implicitly performing a `typedef` based on the tag name when a structure is declared:

```c
struct mystruct {
    ...
};
mystruct s; // Define 's' as a 'mystruct'
```

**Arrays**

Arrays are supported in Cg and are declared just as in C. Because Cg does not support pointers, arrays must always be defined using array syntax rather than pointer syntax:

```c
// Declare a function that accepts an array
// of five skinning matrices
returnType foo(uniform float4x4 mymatrix[5]) {...};
```

Cg for NV2X places substantial restrictions on array declaration and usage. General-purpose arrays can only be used as uniform parameters to a vertex program. The intent of this capability is to allow an application to pass arrays of skinning matrices and arrays of light parameters to a vertex program.

The most important difference from C is that arrays are first-class types. That means that array assignments actually copy the entire array, and that arrays that are passed as parameters are passed by value (the entire array is copied before making any changes), rather than by reference.
Statements and Operators

Cg supports the following types of statements and operators:

- Control Flow
- Function Definitions and Function Overloads
- Arithmetic Operators from C
- Multiplication Function
- Vector Constructor
- Boolean and Comparison Operators
- Swizzle Operator
- Write Mask Operator
- Conditional Operator

Control Flow

Cg uses the following C control constructs:

- function calls and the `return` statement
- `if/else`
- `while`
- `for`

These control constructs require that their conditional expressions be of type `bool`. Because Cg expressions such as `i <= 3` are of type `bool`, this change from C is normally not apparent.

The NV30 vertex hardware supports branch instructions, so `for` and `while` loops are fully supported in the NV30 vertex profile. For other profiles, `for` and `while` loops may only be used if the compiler can fully unroll them (that is, if the compiler can determine the iteration count at compile time). Likewise, `return` can only appear as the last statement in a function in these profiles. [Note: loop unrolling is not fully implemented in the current release of the Cg compiler.]

Function recursion (and co-recursion) is forbidden in Cg.

The `switch`, `case`, and `default` keywords are reserved, but they are not supported by any profiles in the current release of the Cg compiler.

Function Definitions and Function Overloading

To pass a modifiable function parameter in C, the programmer must explicitly use pointers. C++ provides a built-in pass-by-reference mechanism that avoids the need to explicitly use pointers, but this mechanism still implicitly assumes that the hardware supports pointers. Cg must use a different mechanism because the vertex and fragment hardware of the GPU does not support the use of pointers. Cg passes modifiable function parameters by value or result, instead of by reference. The difference between these two methods is subtle; it is only apparent when two function parameters are
aliased by a function call. In Cg, the two parameters have separate storage in the function, whereas in C++ they would share storage.

To reinforce this distinction, Cg uses a different syntax than C++ to declare function parameters that are modified:

- `function blah1(out float x);` // x is output-only
- `function blah2(inout float x);` // x is input and output
- `function blah3(in float x);` // x is input-only
- `function blah4(float x);` // x is input-only (the default, as in C)

Cg supports function overloading by the number of operands and by operand type. The choice of a function is made by matching one operand at a time, starting at the first operand. The formal language specification provides more details on the matching rules, but it is not normally necessary to study these rules—the overloading generally works in an intuitive manner. For example, the following code declares two versions of a function, one that takes two `bool` operands, and one that takes two `float` operands:

```c
bool same(float a, float b)  { return (a == b);}
bool same(bool a,  bool b)   { return (a == b);}
```

### Arithmetic Operators from C

Cg includes all the standard C arithmetic operators (`+ - * /`) and allows the operators to be used on vectors as well as on scalars. The vector operations are always performed in element-wise fashion. For example,

- `float3(a,b,c) * float3(A,B,C)` is equal to `float3(a*A, b*B, c*C)`

These operators can also be used in mixed scalar/vector form—the scalar is "smeared" to form a vector of the necessary size to perform an element-wise operation. Thus,

- `a * float3(A, B, C)` is equal to `float3(a*A, a*B, a*C)`

Note that the built-in arithmetic operators do not currently support matrix operands. In addition, it is important to remember that matrices are not the same as vectors, even if their dimensions are the same.

### Multiplication Function

Cg’s `mul()` function is for multiplying matrices by vectors, and matrices by matrices:

```c
// matrix by column-vector multiply
matrix-column vector:    mul(M, v);

// row-vector by matrix multiply
row vector-matrix:       mul(v, M);

// matrix-by-matrix multiply
matrix-matrix:           mul(M, N);
```
It is important to use the correct version of \texttt{mul()} Otherwise, you are likely to get unexpected results. More detail on the \texttt{mul()} functions is provided in Cg Standard Library Functions.

**Vector Constructor**

Cg allows vectors (up to size 4) to be constructed using the following notation. The vector constructor can appear anywhere in an expression:

\[
y = x \times \text{float4}(3.0, 2.0, 1.0, -1.0);
\]

**Boolean and Comparison Operators**

Cg includes three of the standard C Boolean operators:

\[
\begin{align*}
\&\& \\
\mathbf{||} \\
\mathbf{!}
\end{align*}
\]

In C, these operators consume and produce \texttt{int}-typed values, but in Cg they consume and produce \texttt{bool}-typed values. This difference is not normally noticeable, except when declaring a variable that will hold the value of a Boolean expression. Cg also supports the C comparison operators, which produce \texttt{bool}-typed values:

\[
\begin{align*}
\mathbf{<} \\
\mathbf{<=} \\
\mathbf{!=} \\
\mathbf{==} \\
\mathbf{>=} \\
\mathbf{>}
\end{align*}
\]

Unlike C, Cg allows all the Boolean operators to be applied to vectors, in which case the Boolean operation is performed in an element-wise fashion. Also unlike C, the \&\& and || operators cannot be used for short-circuiting evaluation; expressions with side effects are not allowed in the right-hand operand of \&\& and ||.

**Swizzle Operator**

Cg has a swizzle operator, , , that allows the components of a vector to be rearranged to form a new vector. The new vector need not be the same size as the original vector—elements can be repeated or omitted. The characters \texttt{x, y, z,} and \texttt{w} represent the first, second, third, and fourth components of the original vector, respectively. The characters \texttt{r, g, b,} and \texttt{a} can be used for the same purpose. Because the swizzle operator is implemented efficiently in the GPU hardware, its use is usually free.

The following are some examples of swizzling:

\[
\begin{align*}
\text{float3}(a, b, c) . \text{zyx} & \quad \text{yields} \quad \text{float3}(c, b, a) \\
\text{float4}(a, b, c, d) . \text{xxxy} & \quad \text{yields} \quad \text{float4}(a, a, b, b) \\
\text{float2}(a, b) . \text{yyxx} & \quad \text{yields} \quad \text{float4}(b, b, a, a) \\
\text{float4}(a, b, c, d) . \text{w} & \quad \text{yields} \quad d
\end{align*}
\]
The swizzle operator can also be used to create a vector from a scalar:

\[ a.xxxx \quad \text{yields} \quad \text{float4}(a,a,a,a) \]

The precedence of the . operator is the same as that of the array subcripting operator [].

**Write-Mask Operator**

When placed on the left hand side of an assignment, the . operator is used for write masking. It can be used to selectively overwrite the components of a vector. It is illegal to specify a particular component more than once in a write mask.

The following is an example of a write mask:

```cpp
float4 color = float4(1.0,1.0,0.0,0.0);
color.a = 1.0; // set alpha to 1.0, leaving RGB alone
```

The write-mask operator can be a powerful tool for generating efficient code because it maps well to the capabilities of GPU hardware.

**Conditional Operator**

Cg includes C's if/else functionality, as well as the ? : construct. With the ? : construct, the control variable may be a bool vector. In this case, the second and third operands must be similarly sized vectors, and selection is performed on an element-wise basis. Cg forbids the second and third operands from producing side effects.

Specifically, the second and third operands may not be expressions that call a function with an out parameter.

As an example, the following would be a very efficient way to implement a vector clamp function, if the \text{min()} and \text{max()} functions did not exist:

```cpp
float3 clamp(float3 x, float minval, float maxval) {
    x = (x < minval.xxx) ? minval.xxx : x;
    x = (x > maxval.xxx) ? maxval.xxx : x;
    return x;
}
```

**Texture Lookups in the NV30 Fragment Profile**

Cg's NV30 fragment profile provides a variety of texture-lookup functions. Please note that Cg uses a different set of texture-lookup functions for DirectX 8/ NV2X hardware because of the restricted pixel programmability of that hardware. We won't discuss DirectX 8/ NV2X texture-lookup functions in this introductory chapter.

The NV30 texture lookup functions always require at least two parameters:

- **Texture sampler**
  A texture sampler is a variable with the type sampler1D, sampler2D, sampler3D, samplerCUBE, or samplerRECT and represents the combination of a texture image with a filter, clamp, wrap, or similar configuration. Texture-sampler variables cannot be set directly within the Cg language; instead, they must be provided by the application as uniform parameters to a Cg program.
Texture coordinate
Depending on the type of texture lookup, the coordinate may be a scalar, a two-vector, a three-vector, or a four-vector.

The following short fragment program uses the \texttt{f4tex2D()} function to perform a 2D texture lookup to determine the fragment’s RGBA color. The \texttt{f4} prefix in the texture-function name indicates that the function returns a \texttt{float4} value.

```c
void applytex(uniform sampler2D mytexture,
              float2 uv : TEX0,
              out float4 outcolor : COL) {
  outcolor = f4tex2D(mytexture, uv);
}
```

Cg provides three different types of texture-lookup functions:

- **Standard non-projective texture lookup**
  
  \begin{align*}
  \%tex1D & \quad (\text{float} \ s) \\
  \%tex2D & \quad (\text{float2} \ s) \\
  \%tex3D & \quad (\text{float3} \ s) \\
  \%texCUBE & \quad (\text{float3} \ s) \\
  \%texRECT & \quad (\text{float2} \ s)
  \end{align*}

  where * stands for \texttt{f}, \texttt{h}, or \texttt{x} to indicate a return type of \texttt{float}, \texttt{half}, or \texttt{fixed}; and \% stands for \texttt{1}, \texttt{2}, \texttt{3}, or \texttt{4} to indicate a return type that is a scalar, two-vector, three-vector, or four-vector.

- **Standard projective texture lookup**
  
  \begin{align*}
  \%tex1Dproj & \quad (\text{float2} \ sq) \\
  \%tex2Dproj & \quad (\text{float4} \ sq) \\
  \%tex3Dproj & \quad (\text{float4} \ sq) \\
  \%texCUBEproj & \quad (\text{float4} \ sq) \\
  \%texRECT & \quad (\text{float2} \ sq)
  \end{align*}

- **Non-projective texture lookup with user-specified filter-kernel size**
  
  \begin{align*}
  \%tex1D & \quad (\text{float} \ s, \text{float} \ dsdx, \text{float} \ dsdy) \\
  \%tex2D & \quad (\text{float2} \ s, \text{float2} \ dsdx, \text{float2} \ dsdy) \\
  \%tex3D & \quad (\text{float3} \ s, \text{float3} \ dsdx, \text{float3} \ dsdy) \\
  \%texCUBE & \quad (\text{float3} \ s, \text{float3} \ dsdx, \text{float3} \ dsdy) \\
  \%texRECT & \quad (\text{float2} \ s, \text{float2} \ dsdx, \text{float2} \ dsdy)
  \end{align*}

  The filter size is specified by providing the derivatives of the texture coordinates with respect to pixel-coordinate x (\texttt{dsdx}) and pixel-coordinate y (\texttt{dsdy}).

More Details
The purpose of this chapter has been to give you a brief overview of Cg, so that you can quickly get started and try some experiments to gain hands-on experience. If you would like some more detail about any of the language features described in this chapter, see Appendix A, The Cg Language Specification.
Cg Standard Library Functions

Cg provides a set of built-in functions and predefined structures with binding semantics to simplify GPU programming. These functions are similar in spirit to the C standard library functions, providing a convenient set of common functions. In many cases, the functions map to a single native GPU instruction, meaning that they are executed very quickly. Some functions map to multiple native GPU instructions, but in the near future you can expect that the most useful functions will become more efficient.

You can write customized versions of specific functions for performance or quality reasons, but in general it is wiser to use the Standard Library because its functions will be optimized for future GPUs. This means that a shader you write today will automatically be compile-time optimized for future architectures! An additional feature of the Standard Library is that it provides a convenient unified interface for vertex and fragment programs.

This section describes the contents of the Cg Standard Library, including

- Predefined structs with binding semantics for varying input and output data
- Mathematical functions for DirectX 8/NV2X vertex shaders
- Geometric functions for DirectX 8/NV2X vertex shaders
- DirectX 8/NV2X fragment program functions
- Additional functions for NV30 vertex and fragment shaders

Where appropriate, functions are overloaded to support scalar and vector variations when the input and output types are the same.

**Structs Using Binding Semantics**

Several structs using binding semantics are predefined in the Standard Library, depending on the current profile. For NV2X, which supports the vp20, fp20, dx8vs, and dx8ps profiles, there are three such structs. The vp20 and dx8vs profiles only allow the use of av20 and vf20. The fp20 and dx8ps profiles allow the use of vf20 and fragout.

The av20 struct connects an application to a vertex program and contains the conventional list of parameters passed from applications to vertex programs. For the vp20 profile, it is defined as follows:
struct av20 {
    float4 P : ATTR0;
    float4 WEIGHT : ATTR1;
    float4 N : ATTR2;
    float4 C : ATTR3;
    float4 C2 : ATTR4;
    float4 FOG : ATTR5;
    float4 SIZE : ATTR6;
    float4 ATTR7 : ATTR7;
    float4 TEX0 : ATTR8;
    float4 TEX1 : ATTR9;
    float4 TEX2 : ATTR10;
    float4 TEX3 : ATTR11;
    float4 TEX4 : ATTR12;
    float4 TEX5 : ATTR13;
    float4 TEX6 : ATTR14;
    float4 TEX7 : ATTR15;
};

Next, there is a vertex-to-fragment connecting struct called vf20. Essentially the vf20 struct defines the standard set of parameters that has historically been passed from vertex processing to fragment processing. For the vp20 profile, it is defined as

struct vf20 {
    float4 HPOS : HPOS;
    float4 COL0 : COL0;
    float4 COL1 : COL1;
    float FOGC : FOGC;
    float PSIZ : PSIZ;
    float4 TEX0 : TEX0;
    float4 TEX1 : TEX1;
    float4 TEX2 : TEX2;
    float4 TEX3 : TEX3;
};

For dx8vs and dx8ps, it is defined as

struct vf20 {
    float4 HPOS : POSITION;
    float4 COL0 : COLOR0;
    float4 COL1 : COLOR1;
    float FOGC : FOG;
    float PSIZ : PSIZE;
    float4 TEX0 : TEXCOORD0;
    float4 TEX1 : TEXCOORD1;
    float4 TEX2 : TEXCOORD2;
    float4 TEX3 : TEXCOORD3;
};
For \texttt{vp30} and \texttt{fp30}, it is defined as

```c
struct vf20 {
    float4 HPOS : HPOS;
    float4 COL0 : COL0;
    float4 COL1 : COL1;
    float4 BCOL0 : BCOL0;
    float4 BCOL1 : BCOL1;
    float FOGP : FOGP;
    float FOGC : FOGC;
    float PSIZ : PSIZ;
    float CLP0 : CLP0;
    float CLP1 : CLP1;
    float CLP2 : CLP2;
    float CLP3 : CLP3;
    float CLP4 : CLP4;
    float CLP5 : CLP5;
    float4 TEX0 : TEX0;
    float4 TEX1 : TEX1;
    float4 TEX2 : TEX2;
    float4 TEX3 : TEX3;
    float4 TEX4 : TEX4;
    float4 TEX5 : TEX5;
    float4 TEX6 : TEX6;
    float4 TEX7 : TEX7;
};
```

The third type of \texttt{struct} with binding semantics defined in the Standard Library is fragment-to-frame buffer connecting \texttt{struct}. For \texttt{dx8ps}, there is a \texttt{struct} defined as

```c
struct fragout {
    float4 col : COLOR;
};
```

For \texttt{fp30}, there are two \texttt{struct}s defined as:

```c
struct fragout {
    half4 col : COL;
    float depth : DEPR;
};

struct fragout_float {
    float4 col : COL;
    float depth : DEPR;
};
```
Mathematical Functions

Table 1 lists the mathematical functions that the Cg Standard Library provides. The list includes functions useful for trigonometry, exponentiation, rounding, and vector and matrix manipulations, among others. All functions work on scalars and vectors of all sizes, except where noted.

<table>
<thead>
<tr>
<th>Mathematical Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs(x)</td>
<td>Absolute value of x</td>
</tr>
<tr>
<td>acos(x)</td>
<td>Arccosine of x in range ([-\pi/2,\pi/2]), x in ([-1,1])</td>
</tr>
<tr>
<td>all(x)</td>
<td>Returns 1 if every component of x is not equal to 0; Returns 0 otherwise</td>
</tr>
<tr>
<td>any(x)</td>
<td>Returns 1 if any component of x is not equal to 0; Returns 0 otherwise</td>
</tr>
<tr>
<td>asin(x)</td>
<td>Arcsine of x in range ([0,\pi]) ; x should be in ([-1,1])</td>
</tr>
<tr>
<td>atan(x)</td>
<td>Arctangent of x in range ([-\pi/2,\pi/2])</td>
</tr>
<tr>
<td>atan2(y,x)</td>
<td>Arctangent of y/x in range ([-\pi,\pi])</td>
</tr>
<tr>
<td>ceil(x)</td>
<td>Smallest integer not less than x</td>
</tr>
</tbody>
</table>
| clamp(x,a,b)           | x clamped to the range \([a,b]\) as follows:  
|                        | • Returns a if x is less than a  
|                        | • Returns b if x is greater than b  
|                        | • Otherwise returns x |
| cos(x)                 | Cosine of x |
| cosh(x)                | Hyperbolic cosine of x |
| cross(a,b)             | Cross product of vectors a and b  
|                        | a and b must be 3-component vectors |
| degress(x)             | Radian-to-degree conversion |
| determinant(M)         | Determinant of matrix M |
| dot(a,b)               | Dot product of vectors a and b |
| exp(x)                 | Exponential function \(e^x\) |
| exp2(x)                | Exponential function \(2^x\) |
| floor(x)               | Largest integer not greater than x |
| fmod(x,y)              | Remainder of x/y, with the same sign as x  
<p>|                        | If y is zero, the result is implementation-defined. |</p>
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>frac(x)</td>
<td>Fractional part of x</td>
</tr>
<tr>
<td>frexp(x, out exp)</td>
<td>Splits x into a normalized fraction in the interval [1/2,1), which is</td>
</tr>
<tr>
<td></td>
<td>returned, and a power of 2, which is stored in exp. If x is zero, both</td>
</tr>
<tr>
<td></td>
<td>parts of the result are zero.</td>
</tr>
<tr>
<td>isfinite(x)</td>
<td>True if x is finite</td>
</tr>
<tr>
<td>isnan(x)</td>
<td>True if x is NaN (Not a Number)</td>
</tr>
<tr>
<td>isnan(x)</td>
<td>True if x is NaN (Not a Number)</td>
</tr>
<tr>
<td>ldexp(x, n)</td>
<td>x * 2^n</td>
</tr>
<tr>
<td>lerp(a, b, f)</td>
<td>(1-f)<em>a + b</em>f where a and b are of matching vector/scalar type and f is a</td>
</tr>
<tr>
<td></td>
<td>scalar</td>
</tr>
<tr>
<td>lerp(a, b, f)</td>
<td>(1-f)<em>a + b</em>f where a, b, and f are of matching vector type</td>
</tr>
<tr>
<td>lit(ndotl, ndoth, m)</td>
<td>Computes lighting coefficients for ambient, diffuse, and specular light</td>
</tr>
<tr>
<td></td>
<td>contributions. Returns a 4-vector as follows:</td>
</tr>
<tr>
<td></td>
<td>• The x component of the result vector contains the ambient coefficient,</td>
</tr>
<tr>
<td></td>
<td>which is always 1.0</td>
</tr>
<tr>
<td></td>
<td>• The y component contains the diffuse coefficient which is 0 if (n • l) &lt; 0;</td>
</tr>
<tr>
<td></td>
<td>otherwise (n • l)</td>
</tr>
<tr>
<td></td>
<td>• The z component contains the specular coefficient which is 0 if either</td>
</tr>
<tr>
<td></td>
<td>(n • l) &lt; 0 or (n • h) &lt; 0; (n • h)m otherwise</td>
</tr>
<tr>
<td></td>
<td>• The w component is 1.0</td>
</tr>
<tr>
<td></td>
<td>There is no vectorized version of this function</td>
</tr>
<tr>
<td>log(x)</td>
<td>Natural logarithm ln(x) x must be greater than 0</td>
</tr>
<tr>
<td>log2(x)</td>
<td>Base 2 logarithm of x x must be greater than 0</td>
</tr>
<tr>
<td>log10(x)</td>
<td>Base 10 logarithm of x x must be greater than 0</td>
</tr>
<tr>
<td>max(a, b)</td>
<td>Maximum of a and b</td>
</tr>
<tr>
<td>min(a, b)</td>
<td>Minimum of a and b</td>
</tr>
<tr>
<td>modf(x, out ip)</td>
<td>Splits x into integral and fractional parts, each with the same sign as x</td>
</tr>
<tr>
<td></td>
<td>Stores the integral part in ip and returns the fractional part</td>
</tr>
<tr>
<td>mul(M, N)</td>
<td>Matrix product of matrix M and matrix N, as shown below:</td>
</tr>
</tbody>
</table>
|                  | \[
|                  | \begin{bmatrix}
|                  | M_{11} & M_{12} & M_{13} & M_{14} \\
|                  | M_{21} & M_{22} & M_{23} & M_{24} \\
|                  | M_{31} & M_{32} & M_{33} & M_{34} \\
|                  | M_{41} & M_{42} & M_{43} & M_{44} \\
|                  | \end{bmatrix} \begin{bmatrix}
|                  | N_{11} & N_{12} & N_{13} & N_{14} \\
|                  | N_{21} & N_{22} & N_{23} & N_{24} \\
|                  | N_{31} & N_{32} & N_{33} & N_{34} \\
|                  | N_{41} & N_{42} & N_{43} & N_{44} \\
|                  | \end{bmatrix} \]                                                     |
## Mathematical Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mul(M, v)</td>
<td>Product of matrix M and column vector v, as shown below:</td>
</tr>
</tbody>
</table>
|                   | \[
|                   | \begin{bmatrix}
|                   | M_{11} & M_{21} & M_{31} & \cdots & M_{41} \\
|                   | M_{12} & M_{22} & M_{32} & \cdots & M_{42} \\
|                   | \vdots & \vdots & \vdots & \ddots & \vdots \\
|                   | M_{14} & M_{24} & M_{34} & \cdots & M_{44} \\
|                   | \end{bmatrix}
|                   | \begin{bmatrix}
|                   | v_1 \\
|                   | v_2 \\
|                   | \vdots \\
|                   | v_s \\
|                   | \end{bmatrix}
|                   | \]
|                   | If M is an mxn matrix and v is an nx1 vector, mul(M, v) returns an mx1 vector |
| mul(v, M)         | Product of row vector v and matrix M, as shown below:                         |
|                   | \[
|                   | \begin{bmatrix}
|                   | v_1 & v_2 & v_3 & \cdots & v_s \\
|                   | \end{bmatrix}
|                   | \begin{bmatrix}
|                   | M_{11} & M_{21} & M_{31} & \cdots & M_{41} \\
|                   | M_{12} & M_{22} & M_{32} & \cdots & M_{42} \\
|                   | \vdots & \vdots & \vdots & \ddots & \vdots \\
|                   | M_{14} & M_{24} & M_{34} & \cdots & M_{44} \\
|                   | \end{bmatrix}
|                   | \]
|                   | If v is a 1xm vector and M is an mxn matrix, mul(v, M) returns a 1xn vector  |
| noise(x)          | Either a 1-, 2-, or 3-dimensional noise function depending on the type of its argument |
|                   | The returned value is between 0 and 1 and is consistent from one frame to another |
| pow(x, y)         | \(x^y\)                                                                     |
| radians(x)        | Degree-to-radian conversion                                                  |
| round(x)          | Closest integer to x                                                         |
| rsqrt(x)          | Reciprocal square root of x                                                  |
|                   | x must be greater than 0                                                     |
| sign(x)           | 1 if x > 0; 0 if x < 0; -1 if x < 0                                          |
| sin(x)            | Sine of x                                                                    |
| sincos(float x, out s, out c) | s is set to the sine of x, and c is set to the cosine of x |
|                   | If sin(x) and cos(x) are both needed, this function is more efficient than calculating each individually |
| sinh(x)           | Hyperbolic sine of x                                                         |
| smoothstep(min, max, x) | 0 if x < min; 1 if x >= max;  
Otherwise a smooth Hermite interpolation between 0 and 1 given by:  
\(-2((x-min)/(max-min))^3 + 3((x-min)/(max-min))^2\) |
| step(a, x)        | 0 if x < a; 1 if x >= a                                                     |
Mathematical Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqrt(x)</td>
<td>Square root of x, x must be greater than 0</td>
</tr>
<tr>
<td>tan(x)</td>
<td>Tangent of x</td>
</tr>
<tr>
<td>tanh(x)</td>
<td>Hyperbolic tangent of x</td>
</tr>
<tr>
<td>transpose(A)</td>
<td>Matrix transpose of matrix A. If A is an m x n matrix, the</td>
</tr>
<tr>
<td></td>
<td>transpose of A is an n x m matrix whose first column is the</td>
</tr>
<tr>
<td></td>
<td>first row of A, whose second column is the second row of A,</td>
</tr>
<tr>
<td></td>
<td>whose third column is the third row of A, etc.</td>
</tr>
</tbody>
</table>

Geometric Functions

Table 2 presents the geometric functions that are provided in the Cg Standard Library.

Table 2. Geometric Functions

<table>
<thead>
<tr>
<th>Geometric Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>distance(pt1, pt2)</td>
<td>Euclidean distance between points pt1 and pt2</td>
</tr>
<tr>
<td>faceforward(N, I, N)</td>
<td>N if dot(N, I) &lt; 0</td>
</tr>
<tr>
<td></td>
<td>Otherwise -N</td>
</tr>
<tr>
<td>length(v)</td>
<td>Euclidean length of a vector</td>
</tr>
<tr>
<td>normalize(v)</td>
<td>Returns a vector of length 1 that points in the same</td>
</tr>
<tr>
<td></td>
<td>direction a vector v</td>
</tr>
<tr>
<td>reflect(i, n)</td>
<td>Computes reflection vector from entering ray direction i and</td>
</tr>
<tr>
<td></td>
<td>surface normal n</td>
</tr>
<tr>
<td></td>
<td>Only valid for 3-component vectors</td>
</tr>
<tr>
<td>refract(i, n, eta)</td>
<td>Given entering ray direction i, surface normal n, and</td>
</tr>
<tr>
<td></td>
<td>relative index of refraction eta, computes refraction vector.</td>
</tr>
<tr>
<td></td>
<td>If the angle between i and n is too large for a given eta,</td>
</tr>
<tr>
<td></td>
<td>returns (0,0,0).</td>
</tr>
<tr>
<td></td>
<td>Only valid for 3-component vectors</td>
</tr>
</tbody>
</table>
Texture Map Functions

Table 3 presents the texture functions that are provided in the Cg Standard library. Currently, these texture functions are only supported by the \texttt{fp30} profile, but these functions will also be supported by all future profiles that have texture-mapping capability. Because of the limited pixel programmability of NV20 and DirectX 8 hardware, the \texttt{dx8ps} and \texttt{fp20} profiles use a different set of texture-mapping functions, which are described in Table 6.

In Table 3, \( t \) can be \( f \), \( h \), or \( x \) to indicate the return type \texttt{float}, \texttt{half}, or \texttt{fixed}, respectively; \( n \) stands for \( 1 \), \( 2 \), \( 3 \), or \( 4 \) to indicate a return type that is a scalar, two-vector, three-vector, or four-vector.

<table>
<thead>
<tr>
<th>Texture Map Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{tn tex1D(sampler1D tex, float s)}</td>
<td>1D non-projective</td>
</tr>
<tr>
<td>\texttt{tn tex1Dproj(sampler1D tex, float2 sq)}</td>
<td>1D projective</td>
</tr>
<tr>
<td>\texttt{tn tex1D(sampler1D tex, float s, float dsdx, float dsdy)}</td>
<td>1D non-projective w/derivatives</td>
</tr>
<tr>
<td>\texttt{tn tex2D(sampler2D tex, float2 s)}</td>
<td>2D non-projective</td>
</tr>
<tr>
<td>\texttt{tn tex2Dproj(sampler2D tex, float3 sq)}</td>
<td>2D projective</td>
</tr>
<tr>
<td>\texttt{tn tex2D(sampler2D tex, float2 s, float dsdx, float dsdy)}</td>
<td>2D non-projective w/derivatives</td>
</tr>
<tr>
<td>\texttt{tn tex3D(sampler3D tex, float3 s)}</td>
<td>3D non-projective</td>
</tr>
<tr>
<td>\texttt{tn tex3Dproj(sampler3D tex, float4 sq)}</td>
<td>3D projective</td>
</tr>
<tr>
<td>\texttt{tn tex3D(sampler3D tex, float3 s, float dsdx, float dsdy)}</td>
<td>3D non-projective w/derivatives</td>
</tr>
<tr>
<td>\texttt{tn texCUBE(samplerCUBE tex, float3 s)}</td>
<td>Cube-map non-projective</td>
</tr>
<tr>
<td>\texttt{tn texCUBEproj(samplerCUBE tex, float4 sq)}</td>
<td>Cube-map projective (not particularly useful; included for orthogonality)</td>
</tr>
<tr>
<td>\texttt{tn texCUBE(samplerCUBE tex, float3 s, float dsdx, float dsdy)}</td>
<td>Cube-map non-projective w/derivatives</td>
</tr>
<tr>
<td>\texttt{tn texRECT(samplerRECT tex, float2 s)}</td>
<td>2D non-projective for RECT textures (non-power-of-two size)</td>
</tr>
</tbody>
</table>
### Texture Map Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tn</code> <code>texRECTproj(samplerRECT tex, float3 sq)</code></td>
<td>2D projective for RECT textures (non-power-of-two size)</td>
</tr>
<tr>
<td><code>tn</code> <code>texRECT(samplerRECT tex, float2 s, float dsdx, float dsdy)</code></td>
<td>2D non-projective w/derivatives for RECT textures (non-power-of-two size)</td>
</tr>
<tr>
<td><code>t</code> <code>1texcompare2D(sampler2D tex, float2 sz)</code></td>
<td>Depth-compare texture mapping, non-projective (Texture unit must also be configured for depth compare)</td>
</tr>
<tr>
<td><code>t</code> <code>1texcompare2D(sampler2D tex, float3 sqz)</code></td>
<td>Depth-compare texture mapping, projective (Texture unit must also be configured for depth compare)</td>
</tr>
<tr>
<td><code>t</code> <code>1texcompare2D(sampler2D tex, float 2sz, float dsdx, float dsdy)</code></td>
<td>Depth-compare texture mapping, non-projective w/derivatives (Texture unit must also be configured for depth compare)</td>
</tr>
</tbody>
</table>

### Derivative Functions

Table 4 presents the derivative functions that are supported by the Cg standard library. These functions are supported only in the NV30 OpenGL fragment profile (`fp30`).

**Table 4. Derivative Functions**

<table>
<thead>
<tr>
<th>Derivative Functions Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ddx(a)</code></td>
<td>Approximate partial derivative of a with respect to screen-space x coordinate.</td>
</tr>
<tr>
<td><code>ddy(a)</code></td>
<td>Approximate partial derivative of a with respect to screen-space y coordinate.</td>
</tr>
<tr>
<td><code>fwidth(a)</code></td>
<td>Sum of absolute values of partial derivatives of a with respect to screen-space x coordinate and screen-space y coordinate.</td>
</tr>
</tbody>
</table>
Using the
Cg Run-Time Library

This chapter describes the Cg run-time library APIs for OpenGL and Direct3D, known as CgGL and CgD3D, respectively. It assumes you have some basic knowledge of the Cg language, vertex programs, fragment programs, and OpenGL or Direct3D—depending on the API that you plan to use in your applications.

Introduction

The Cg compiler can be used in one of two ways:

- **Offline compilation**
  The Cg compiler is used by the application developer as a command-line tool during application development to compile Cg code into GPU assembly-language code. The application is responsible for using this assembly-language code in the same way that the application would be responsible for using a handwritten vertex or fragment program.

- **Run-time compilation**
  The Cg compiler is invoked by the application at run time, through a run-time API. The run-time API provides additional capabilities beyond just compilation—most importantly, it allows the application to manipulate the inputs to a Cg program using the same parameter names that appear in the prototype of the Cg program. Without this capability, the inputs to a GPU program must be manipulated using hardware register names. For example, the run-time API allows a parameter `time` to be set by using the name `time`, rather than by using the name of the corresponding hardware register, such as `c[23]`. The run-time API is the easiest way to use Cg in an application; we recommend its use unless you are planning to hand-edit the assembly code generated by the offline Cg compiler.

---

1 Please refer to the included header files for the latest run-time API function prototypes.
Capabilities Provided by the Run-Time API

The Cg run time provides the many capabilities available to an application. Because of differences between OpenGL and Direct3D, some of these capabilities are implemented using different routines in the OpenGL and Direct3D versions of the run time. A [*] marks such capabilities in the list below.

- Compile a Cg program provided via a string or file.
- Send the compiler-generated assembly code to the underlying API (OpenGL or Direct3D). [*]
- Bind to a Cg program in preparation for rendering (that is, choose previously compiled Cg vertex and fragment programs to use for rendering a batch of geometry). [*]
- Query to determine the list of uniform and varying (per-vertex) parameters required by a program.
- Query the type (vector, matrix, and so on) of a particular parameter.
- Set the value of a uniform parameter to a program. [*]
- Make changes to one of the texture units/samplers used by a program. [*]
- Specify a vertex array that can be used for a varying (per-vertex) parameter to a program. [*]
- Query the hardware register used by a particular parameter.

Key Concepts

The run-time API is organized around three key concepts:

- Programs
- Parameters
- Contexts

Programs

A program is a handle to a source-code string that can be compiled to produce a vertex program or a fragment program. Each program has the following associated states:

- **Source code for the Cg program**
  The source code consists of the top-level function that forms the entry point for the Cg program, as well as any supporting functions and global variables.

- **Name of top-level function that forms entry point for the Cg program**
  In Cg, unlike in C, the top-level function does not need to be called `main()` (and it usually isn't).

- **Profile under which the Cg program should be compiled**
  Currently, the profile must be one of the following:
  
  - `cgDX8VertexProfile` DirectX 8 vertex programs
  - `cgDX8PixelProfile` DirectX 8 pixel programs (pixel shaders)
  - `cgARBVertexProfile` OpenGL ARB_vertex_program programs
Parameters

Associated with each program is a list of parameters (also referred to as bindings). Each parameter has the following information associated with it:

- Parameter name
- Kind of parameter
  - texture sampler/texture unit: \texttt{cgTexObjParam}
  - uniform parameter: \texttt{cgUniformParam}
  - per-vertex parameter: \texttt{cgConnectorMemberParam}
- Hardware register/unit to which the parameter is assigned
  - \texttt{cgBindTexN}: texture unit/sampler \(N\) (\(N=0..3\))
  - \texttt{cgBindAttrN}: vertex attribute \(N\) (\(N=0..15\))
  - \texttt{cgBindVertUniform}: one of the uniform-value registers
- Kind of parameter type
  - \texttt{cgFloatValueType}: scalar float
  - \texttt{cgFloat2ValueType}: 2-vector
  - \texttt{cgFloat3ValueType}: 3-vector
  - \texttt{cgFloat4ValueType}: 4-vector
  - \texttt{cgFloat1x1ValueType}: 1x1 matrix
  - \texttt{cgFloat1x2ValueType}: 1x2 matrix
  - \texttt{cgFloat1x3ValueType}: 1x3 matrix
  - \texttt{cgFloat1x4ValueType}: 1x4 matrix
  - \texttt{cgFloat2x1ValueType}: 2x1 matrix
  - \texttt{cgFloat2x2ValueType}: 2x2 matrix
  - \texttt{cgFloat2x3ValueType}: 2x3 matrix
  - \texttt{cgFloat2x4ValueType}: 2x4 matrix
  - \texttt{cgFloat3x1ValueType}: 3x1 matrix
  - \texttt{cgFloat3x2ValueType}: 3x2 matrix
  - \texttt{cgFloat3x3ValueType}: 3x3 matrix
  - \texttt{cgFloat3x4ValueType}: 3x4 matrix
  - \texttt{cgFloat4x1ValueType}: 4x1 matrix
  - \texttt{cgFloat4x2ValueType}: 4x2 matrix
  - \texttt{cgFloat4x3ValueType}: 4x3 matrix
  - \texttt{cgFloat4x4ValueType}: 4x4 matrix
Contexts

A context (cgContext) holds one or more programs. The application can add a program to a context or request a handle to a particular program within a context by specifying the name of the program's top-level function. A context may contain programs from several different profiles.

The Cg run time uses iterators to cycle through lists of programs and lists of program parameters, or to represent a handle to a single program or program parameter. A program iterator (cgProgramIter) is used as a handle to a program, or to cycle through the list of programs in a context. A binding iterator (cgBindIter) is used as a handle to a parameter, or to cycle through the list of parameters to a particular program.

Run-Time Setup

In this release of the Cg toolkit, the Cg run time invokes the command-line Cg compiler (cgc.exe) as a separate process. As a result, the run time must know the path to the compiler executable. On Windows, the default behavior of the Cg run time is to obtain this path from the following registry key:

```
HKEY_LOCAL_MACHINE\SOFTWARE\NVIDIA
Corporation\Global\Cg\CG_COMPILER_EXE
```

<table>
<thead>
<tr>
<th>type</th>
<th>STRING</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>A valid Windows path to the compiler binary</td>
</tr>
<tr>
<td>default</td>
<td>C:\Program Files\NVIDIA Corporation\NVSDK\Cg\bin\cgc.exe</td>
</tr>
</tbody>
</table>

This registry key is normally created by the Cg toolkit installer. This path may be overridden by setting the environment variable `CG_COMPILER_EXE`. Alternatively, the API routine `cgSetCompilerExe()` may be used to set the path:

```
cgSetCompilerExe("c:\\cg\\cgc.exe");
```

If neither method is used, the Cg run time looks for `cgc.exe` in the user's path. (Eventually, the Cg toolkit will have the Cg compiler built into the run-time library.)

An application can also choose which Cg compiler to use based on how an environment variable is set. Setting the environment variable `CG_COMPILER_EXE` to the full path of the `cgc` executable overrides all other methods of setting the path for the run time. Although it is not recommended, hard coding a path into your application can be done by simply setting the environment variable before making any Cg run-time function calls. You set the environment variable using the `SetEnvironmentVariable()` function in Win32 or `putenv()` in Linux.

If you would like to set the path after your application has already been compiled, you may use a wrapper script. In the script, set the environment variable and then launch your application. For example, in Win32 your `.bat` file might look like:

```bash
set CG_COMPILER_EXE={FULL PATH TO YOUR EXE}
myapp.exe
```
In Linux (or other Unix variant) it may look like:

```bash
#!/bin/csh -f
setenv CG_COMPILER_EXE (FULL PATH TO YOUR EXE)
myapp $*
```

Adding a Program to a Context

The following code sample shows how to

- Create a context
- Add two different programs to the context (using different profiles)
- Extract handles to these programs
- Perform cleanup at the end of the application

```c
#include "cs.h"

int main()
{
    cgContext *MyContext;
    cgProgram *MyProg1;
    cgProgram *MyProg2;
    cgProgramIter *MyHandle1;
    cgProgramIter *MyHandle2;
    cgError err;

    // Create a context
    MyContext = cgCreateContext();

    // Put the source code for the programs into strings
    char *Source1 = "… etc …"; // A Cg vertex
    // program, "MyVertProg"
    char *Source2 = "… etc …"; // A Cg fragment
    // program, "MyFragmentProg"

    // Add the vertex program to the context,
    // as a DirectX 8 vertex program.
    // The entry point for the program
    // must be "MyVertProg"
    err = cgAddProgram(MyContext, Source1,
                        cgDX8VertexProfile, "MyVertProg");
    if (err) fprintf("Error msg = '%s\n', cgErrorMsg(err));

    // Add the fragment program to the context,
    // as a DX8 pixel program.
    // The entry point for the program
    // must be "MyFragmentProg"
    err = cgAddProgram(MyContext, Source2,
                        cgDX8PixelProfile, "MyFragmentProg");
    if (err) fprintf("Error msg = '%s\n', cgErrorMsg(err));

    // Extract handles to the two programs
    MyHandle1 = cgProgramByName(MyContext,"MyVertProg");
    MyHandle2 = cgProgramByName(MyContext,"MyFragmentProg");

    return 0;
}
```

NVIDIA
// ... the rest of the application ...  

// Cleanup

cgFreeContext(MyContext);

cgFreeProgramIter(MyHandle1);

cgFreeProgramIter(MyHandle2);

General Parameter Management

The Cg run time provides a large set of OpenGL-specific and Direct3D-specific routines, described later in this document, for specifying the values of uniform and varying parameters. All of these routines refer to parameters using a handle of type cgBindIter. This handle is obtained using the following routine, which works under Direct3D and OpenGL (it is invoked slightly differently in Direct3D, as described later).

```c
.cgBindIter *

cgGetBindByName(const cgProgramIter *program,

  const char *parameter_name);
```

The Cg run time also provides several other functions for obtaining information about parameters:

```c
.cgBindIter *

cgGetNextBind(cgProgramIter *program,

  cgBindIter *iter);

void cgFreeBindIter(cgBindIter *iter);

const char * cgGetBindParamName(const cgBindIter *bind);

cgParamType cgGetBindParamType(const cgBindIter *bind);

cgBindLocation cgGetBindLocation(const cgBindIter *bind);

cgValueType cgGetBindValueType(const cgBindIter *bind,

  int *array_size);

cgBindIter *cgDuplicateBindIter(cgBindIter *bind);

cgProgramIter *cgGetLastProgram(cgContext *context);
```
Using the OpenGL Run-Time Library

Downloading Programs to the Hardware

The cgAddProgram() call causes the Cg program to be compiled for the appropriate profile, but does not do anything with the resulting assembly-language code except to store it internally. The next step is to feed this assembly-language code to the underlying hardware. For OpenGL, the cgGLLoadProgram() call is used to perform this task. This call is layered on top of the OpenGL routine glLoadProgramNV() and feeds the assembly-language code to the hardware as either a vertex program or fragment program, as appropriate. The prog_id is a user-specified integer that names the program for subsequent OpenGL calls.

```
// Load a program, and name it
//   for OpenGL as program #23
// The program (as identified by MyHandle1)
//   must have been compiled using an OpenGL profile,
//   via cgAddProgram
err = cgGLLoadProgram(MyHandle1, 23);
if (err) fprintf("Error msg = '%s'\n", cgErrorMsg(err));
```

Preparing to Render Within an Application

Because OpenGL can execute with either a configurable pipeline or a programmable pipeline, it is necessary to enable the programmable pipeline before any program can be used. The routine cgGLEnableProgramType() performs this task for a particular profile:

```
// Enable vertex programs
// This routine ends up calling
//   glEnable(GL_VERTEX_PROGRAM_NV)

cgGLEnableProgramType(cgGLVertexProfile);
```

The programmer chooses which program is active for rendering at any particular time by calling cgGLBindProgram(). Note that there may be two programs simultaneously active—one vertex program and one fragment program.

```
// Pick the program 'MyHandle1' as
//   the active program for rendering

cgGLBindProgram(MyHandle1);
```
Managing Uniform Parameters

To assign uniform parameters, the OpenGL run-time library offers the following family of functions, following the usual OpenGL naming conventions:

```c
cgError cgGLBindUniform4f(cgProgramIter *program,
                           cgBindIter *param,
                           float x,
                           float y,
                           float z,
                           float w);

cgError cgGLBindUniform4d(cgProgramIter *program,
                           cgBindIter *param,
                           double x,
                           double y,
                           double z,
                           double w);

cgError cgGLBindUniform4fv(cgProgramIter *program,
                           cgBindIter *param,
                           const float v[]);

cgError cgGLBindUniform4dv(cgProgramIter *program,
                           cgBindIter *param,
                           const double v[]);

cgError cgGLBindUniformMatrixrf(cgProgramIter *program,
                            cgBindIter *param,
                            const float *matrix);

cgError cgGLBindUniformMatrixrd(cgProgramIter *program,
                            cgBindIter *param,
                            const double *matrix);

cgError cgGLBindUniformMatrixcf(cgProgramIter *program,
                            cgBindIter *param,
                            const float *matrix);

cgError cgGLBindUniformMatrixcd(cgProgramIter *program,
                            cgBindIter *param,
                            const double *matrix);

cgError cgGLBindUniformStateMatrix(cgProgramIter *program,
                            cgBindIter *param,
                            cgGLMatrixType matrix_type,
                            int format);
```
The `cgGLBindUniform4{fd}[v]()` family of functions binds four component vector values to the given parameter binding object. The \textit{f}, \textit{d}, and \textit{v} characters in the function names represent \textit{float}, \textit{double}, and \textit{array} in the same style as OpenGL.

The `cgGLBindUniformMatrix{cr}{fd}()` family of functions binds a matrix represented by an array of values to a parameter-binding object. The \textit{c} and \textit{r} characters represent the convention in which the matrix is stored. If the \textit{c} function is used, the OpenGL run time assumes that the matrix is stored in column-major order. If \textit{r} is used, it assumes the matrix is stored in row-major order. The \textit{f} and \textit{d} characters represent the \textit{float} and \textit{double} data types, respectively. The array must contain the correct number of elements. The number of elements is relative to the specific matrix type of the binding object. The function `cgGetBindValueType()` may be used to determine the type of the matrix (\textit{cgFloat4x4ValueType}, \textit{cgFloat3x3ValueType}, and so on.).

The `cgGLBindUniformStateMatrix()` function extracts a 4x4 matrix from the OpenGL state and binds it to the given parameter. The specific matrix extracted from the OpenGL state may be selected with the \textit{matrix_type} parameter. One of the following enumerants may be used:

- `cgGLModelViewMatrix` Extracts the current modelview matrix.
- `cgGLProjectionMatrix` Extracts the current projection matrix.
- `cgGLTextureMatrix` Extracts the current texture matrix.
- `cgGLModelViewProjectionMatrix` Extracts the concatenated modelview and projection matrices.

The format parameter must be a value or combination of values from the `cgGLMatrixFormat` enum. It may have the following values:

- `cgGLMatrixIdentity` Does nothing to the matrix.
- `cgGLMatrixTranspose` Transposes the matrix before binding it.
- `cgGLMatrixInverse` Inverts the matrix before binding it.
- `cgGLMatrixInverse | cgGLMatrixTranspose` The two flags may be combined with a bitwise OR to yield a matrix that is both inverted and transposed.
Managing Varying Parameters

The following functions provide a method of binding varying parameters. They must be called inside a `glBegin()` and `glEnd()` block, once for each vertex, similar to the behavior for `glVertex()`.

```c
cgError cgGLBindVarying{1234}{fds}(cgProgramIter *program,
    cgBindIter *param,
    GLTYPE x,
    ...);
```

This prototype represents twelve functions. Each function may take one, two, three, or four values as explicit parameters. The parameters may be one of three types: `GLfloat`, `GLdouble`, or `GLshort`.

```c
cgError cgGLBindVarying{1234}{fds}v(cgProgramIter *program,
    cgBindIter *param,
    GLTYPE *v);
```

This prototype is similar to the previous prototype, but takes a pointer to an array of values instead of explicitly passing each component as a parameter.

```c
cgError cgGLBindVarying4ub(cgProgramIter *program,
    cgBindIter *param,
    GLubyte x,
    GLubyte y,
    GLubyte z,
    GLubyte w);
```

```c
cgError cgGLBindVarying4ubv(cgProgramIter *program,
    cgBindIter *param,
    GLubyte *v);
```

These two are similar to the previous functions, but are used with unsigned byte data. The unsigned byte functions are only available for four-component vectors. The following function allows per-vertex values to be specified using a vertex array. It is layered on top of `glVertexAttribPointerNV()`, and its second through fifth parameters have similar meaning.
The `cgGLEnableClientState()` function must be called with the parameter before `cgGLBindVaryingPointer()` is called. `cgGLDisableClientState()` should be called when `cgGLBindVaryingPointer()` is no longer going to be called with a given parameter.

```c
cgError cgGLBindVaryingPointer(cgProgramIter *program, 
    cgBindIter *param, 
    GLint fsize, 
    GLenum type, 
    GLsizei stride, 
    GLvoid *pointer);

cgError cgGLEnableClientState(cgProgramIter *program, 
    cgBindIter *param);

cgError cgGLDisableClientState(cgProgramIter *program, 
    cgBindIter *param);
```

### Managing Textures and Texture Units

The following functions allow manipulation of the OpenGL texture unit state. They can be used for assigning texture coordinates and textures.

```c
GLenum cgGLTextureUnit(const cgBindIter *param);

cgError cgGLActiveTexture(const cgBindIter *param);

cgError cgGLActiveClientTexture(const cgBindIter *param);
```

If `param` is a texture parameter, this function returns the texture unit the parameter is assigned to. The value is one of the GL enumerants of the form `GL_TEXTUREi_ARB`, where `i` is the texture unit number.

This function calls `glActiveTexture()` with the appropriate texture unit associated with `param`. Once this is called, the user may use any of the OpenGL functions that modify the texture state to assign a texture `param`.

This function calls `glActiveClientTexture()` with the appropriate texture unit associated with `param`. Once this is called, texture coordinates may be assigned in the standard GL fashion.
Using the Direct3D Run-Time Library

Managing Programs

The CgD 3D run-time library is object oriented, so the first thing you have to do is instantiate a CgD 3D object.

```cpp
cgDirect3D cg;
```

Once you have this object (let’s assume it’s called `cg`), you should create a context for your programs, using

```cpp
cgContextContainer * pContextContainer = cg.CreateContext(m_pd3dDevice);
```

As shown here, you have to pass in a pointer to your Direct3D device to create a context. Now, you can use the context to load programs, using `LoadCGProgramFromFile()` or `LoadCGProgramFromMemory()`:

```cpp
cgProgramContainer * LoadCGProgramFromFile(
    char * filename,
    char * name,
    cgProfileType type,
    cgVertexStreams * va=0,
    const char * entry=0);
```

```cpp
cgProgramContainer * LoadCGProgramFromMemory(
    const char * memory,
    const char *title,
    cgProfileType type,
    cgVertexAttribute * va = 0,
    DWORD * outIndex = 0,
    const char * entry = 0); // reserved, but not used
```

Here are two examples of how `LoadCGProgram*()` is used:

```cpp
// Load vertex program from file
pVertexProgramContainer = pContextContainer->
LoadCGProgramFromFile("ripple.cg", "Ripple Vertex Shader",
    cgDX8VertexProfile);

// Load fragment program from file
pPixelProgramContainer = pContextContainer->
LoadCGProgramFromFile("RipplePixel.cg", "test",
    cgDX8PixelProfile);

// Load vertex program from memory
const char * CgTextProgramInMemory;
pVertexProgramContainer = pContextContainer->
LoadCGProgramFromMemory(CgTextProgramInMemory,
    "Ripple Vertex Shader",
    cgDX8VertexProfile);
```
Managing Parameters

After a Cg program is compiled, you can inquire about where your data was loaded. Each data element is allocated a binding iterator, which provides an interface to the data. Binding associates a variable with a hardware location. An iterator is provided because the data is stored as a linked list. You use the binding iterator as a handle to modify your Cg variables. As an example, say we have the following vertex program:

```c
vfconn main(appdata I,
    uniform float3x3 mat, // These are your variables,
    uniform float4 vA,    // which you can reference by name.
    uniform float4 vD,    // Full name is main.vD
    uniform float4 vSin,  // Full name is main.vSin
    uniform float4 vCos,  // Full name is main.vCos
    uniform float4 Kd)    // Full name is main.Kd
{
    ...
}
```

After you have a `cgProgramContainer` for the program, you can get the binding iterator for each of the arguments to `main`. You can do this as follows:

- Get the binding iterator:
  ```c
  vertex_mat_iter = pVertexProgramContainer->GetParameterBindByName("mat");
  // this searches the linked list of iterators to return your
  // parameter named "mat"
  ```

- Set the data with the binding iterator by
  ```c
  // Now that you have the binding iterator(vertex_mat_iter),
  // use that to modify your data.
  // data is a pointer to data that is
  // the same size as mat,
  // in this case, a 3x3 matrix.
  pVertexProgramContainer->SetShaderConstant(vertex_mat_iter  data);
  ```

- or by
  ```c
  pVertexProgramContainer->SetShaderConstantD3DXMATRIX(vertex_mat_iter  data);
  // this class interface converts a D3DXMATRIX to the
  // type defined in your Cg program,
  // like this 3x3 matrix
  ```

The size of the data passed to `SetShaderConstant()` must match that defined in your Cg program. If you define a `float2x3` matrix, `SetShaderConstant()` expects six `floats`.

The full name of the `mat` parameter is `main.mat`; you can use this if you have to differentiate names.
Activating Programs

The **SetShaderActive**() function loads the shader handle associated with a program. The Cg compiler generates constants that need to be loaded to run. These are loaded when **SetShaderActive**() is called.

```c
pVertexProgramContainer->SetShaderActive();
```

For fragment programs, textures are assigned positions. Currently, these are zero to eight. You can set all the Direct3D states via the binding iterator of the texture. You may inquire the texture position with **GetTexturePosition**():

```c
pPixelProgramContainer->SetShaderActive();
LPDIRECT3DTEXTURE8 m_pNvidiaTexture;
tex0_iter = pPixelProgramContainer->GetTextureBindByName("tex0");
pPixelProgramContainer->SetTexture(tex0_iter, m_pNvidiaTexture);
```

or

```c
int t0 = pPixelProgramContainer->GetTexturePosition(tex0_iter);
m_pd3dDevice->SetTexture(t0, m_pNvidiaTexture);
```

Vertex Format

Your vertex buffer format is expected to match the input varying data structure that is specified in your Cg program:

```c
struct appdata {
    float3 position : POSITION;
    float2 texcoord0 : TEXCOORD0;
    float2 texcoord1 : TEXCOORD1;
};
```

// your vertex declaration should be
struct {
    float position[3];
    float texcoord0[2];
    float texcoord1[2]; // or D3DXVECTOR2
};

You can use **GetVertexDeclaration**() to return a text description of what the Cg program is expecting.
Specifying a Different Vertex Format

You may use the `cgVertexAttribute*` in `LoadCGProgramFromfile()` to specify a different vertex format, but your vertex `struct` must be a superset of what the Cg program expects. All connector information must be supplied in your vertex. Suppose your vertex is defined like this:

```c
typedef struct DisplayVertex
{
    D3DXVECTOR2 extra_data; // different VB than Cg
    D3DXVECTOR3 pos;
    D3DXVECTOR2 tex0;
    D3DXVECTOR2 tex1;
} DisplayVertex;
```

and your Cg program specifies this varying data input structure:

```c
struct appdata {
    float3 position : POSITION;
    float2 texcoord0 : TEXCOORD0;
    float2 texcoord1 : TEXCOORD1;
};
```

You would use this structure to indicate your vertex format:

```c
// shows how to use different vertex declaration
// than Cg specifies
// number of floats, vertex connector name, stream

cgVertexDefinition vertexDefinition[] = {
    {D3DVSDT_FLOAT3, "Position", 0},
    {D3DVSDT_FLOAT3, "Normal", 0},
    {D3DVSDT_D3DCOLOR, "Diffuse", 0},
    {D3DVSDT_FLOAT2, "TexCoord", 0},
    {D3DVSDT_FLOAT3, "T", 0},
    {D3DVSDT_FLOAT3, "B", 0},
    {D3DVSDT_FLOAT3, "N", 0},
    CGVERTEXDEFINITIONEND
};
```

An error is returned if all the connector elements are not specified.
Texture Operations

The Cg compiler determines what texture units are assigned. You should not have to know which texture unit is bound to your Direct3D texture pointer. The supported Direct3D functions that require the texture unit are:

```c
HRESULT SetTexture(cgBindIter * BindIter,
                   LPDIRECT3DBASETEXTURE8 pTexture);
```

```c
HRESULT SetTextureStageState(cgBindIter * BindIter,
                             D3DTEXTURESTAGESTATETYPE Type, DWORD Value);
```

// Examples:
pPixelProgramContainer->SetTexture(tex0_iter, pTexture);
pPixelProgramContainer->
SetTextureStageState(tex0_iter, type, value);

The `SetRenderState()` call with render states `D3DRS_WRAP0` through `D3DRS_WRAP7` requires the texture unit number, so that too is implemented in the run time with `SetTextureWrapMode()`.

```c
HRESULT SetTextureWrapMode(cgBindIter * BindIter,
                          DWORD WrapCoords);
```

`WrapCoords` are the same parameters that you specify with the render state using `D3DWRAPCOORD_0` and `D3DWRAPCOORD_1`. `SetStreamSource()`, `SetIndices()` and `DrawIndexedPrimitive()` calls should not be affected by adding a Cg program to an existing application.
This section walks you through the sample Cg Microsoft Visual Studio workspace we have provided, along with a simple Cg program that you can use for experimentation.

### Loading the Workspace

When you load the `Cg_Simple` file, your workspace should look like the page shown in Figure 2. As usual, the **FileView** shows you the various files in the project. What’s different in this case, though, is that in addition to the usual **Source Files** and **Header Files**, there is a list of **Cg Programs**.

This Cg Program list should contain one Cg program, `simple.cg`, which is what you can use for experimentation. Double-click `simple.cg` to open it for editing. While you are editing `simple.cg`, you can press Control+F7 at any time to compile it. Because of the way the project is set up, any errors in your code will be shown just as when you compile a normal C or C++ program. You can also double-click on an error, which takes you to the location in the source code that caused the error.

![The Cg_Simple Workspace](image)

**Figure 2.** The Cg_Simple Workspace
Understanding simple.cg

The Cg_Simple application runs the shader defined in simple.cg on a torus. The provided version of simple.cg calculates diffuse and specular lighting for each vertex. Figure 3 shows a screenshot of the shader.

![Figure 3. The simple.cg Shader](See Appendix D for a color version of this illustration)

Program Listing for simple.cg

The following is the program listing for simple.cg:

```c
// define inputs from application
struct appin
{
    float4 Position : POSITION;
    float4 Normal : NORMAL;
};

// define outputs from vertex shader
struct vertout
{
    float4 Hposition : POSITION;
    float4 Color0 : COLOR0;
};
```

vertout main(appin In,
    uniform float4x4 ModelViewProj : C0,
    uniform float4x4 ModelViewIT : C4,
    uniform float4 LightVec)
{
vertout Out;

Out.HPosition = mul(ModelViewProj, In.Position);

// transform normal from model-space to view-space
float4 normal = normalize(mul(ModelViewIT,
    In.Normal).xyzz);

// store normalized light vector
float4 light = normalize(LightVec);

// calculate half angle vector
float4 eye = float4(0.0, 0.0, 1.0, 1.0);
float4 half = normalize(light + eye);

// calculate diffuse component
float diffuse = dot(normal, light);

// calculate specular component
float specular = dot(normal, half);
specular = pow(specular, 32);

// blue diffuse material
float4 diffuseMaterial = float4(0.0, 0.0, 1.0, 1.0);

// white specular material
float4 specularMaterial = float4(1.0, 1.0, 1.0, 1.0);

// combine diffuse and specular contributions
// and output final vertex color
Out.Color0 = diffuse * diffuseMaterial +
    specular * specularMaterial;

return Out;
}
Definitions for Structures with Varying Data

The first thing to notice is the definitions of structures with binding semantics for varying data. Because our code is so simple, we have declared our own structures instead of using the predefined ones in the Standard Library.

Let’s take a look at the `appin` structure:

```cgm
// define inputs from application
struct appin
{
    float4 Position : POSITION;
    float4 Normal : NORMAL;
};
```

This structure contains only two members: `Position` and `Normal`. Because this data varies per-vertex, the binding semantics `POSITION` and `NORMAL` tell the compiler that the position information is associated with the predefined attribute `POSITION` and that the normal information is associated with the predefined attribute `NORMAL`.

The other structure that is defined in `simple.cg` is `vertout`, which connects the vertex to the fragment:

```cgm
// define outputs from vertex shader
struct vertout
{
    float4 Hposition : POSITION;
    float4 Color0 : COLOR0;
};
```

The `vertout` structure also contains only two members: `Hposition`, the vertex position in homogeneous coordinates, and `Color0`, the vertex color. Again, binding semantics are used to specify register locations for the variables. In this case, the homogeneous position information resides in the hardware register corresponding to `POSITION` and that the normal information resides in the hardware register corresponding to `COLOR0`.

Passing Arguments

Now let’s take a look at the body of the program, section by section, starting with the declaration of `main()`:

```cgm
vertout main(appin In,
    uniform float4x4 ModelViewProj,
    uniform float4x4 ModelViewIT,
    uniform float4 LightVec)
{
```

As required for a vertex program, `main()` takes an application-to-vertex connector as input and returns a vertex-to-fragment connector. In this case, we are using the two connector types we have already defined: `appin` and `vertout`. Notice that `main()`
takes in three uniform parameters: two matrices and one vector. All three parameters are passed to `simple.cg` by the application, using the run-time library.

The first matrix, `ModelViewProj`, is the concatenation of the modelview and projection matrices. Together, these matrices transform points from model space to clip space. The second matrix, `ModelViewIT`, is the inverse transpose of the modelview matrix. The third parameter, `LightVec`, is a vector that specifies the location of the light source.

### Basic Transformations

Now we start the body of the vertex program:

```cpp
vertout Out;
Out.HPosition = mul(ModelViewProj, In.Position);
```

A vertex program is responsible for calculating the homogenous clip-space position of the vertex (given the vertex's model-space coordinates). Therefore, the vertex's model-space position (given by `In.Position`) needs to be transformed by the concatenation of the modelview and projection matrices (called `ModelViewProj` in this example). The transformed position is assigned directly to `Out.HPosition`. Note that you are not responsible for the perspective division when using vertex programs. The hardware automatically performs the division after executing the vertex program.

Since we want to do our lighting in eye space, we have to transform the model space normal `In.Normal` to eye space:

```cpp
// transform normal from model-space to view-space
float4 normal = normalize(mul(ModelViewIT, In.Normal).xyzz);
```

Remember that when transforming normals, we need to multiply by the inverse transpose of the modelview matrix. After the multiplication, we use the swizzle operator to replicate the z component in the w component. Finally, we normalize the eye space normal vector and store it as `normal`.

### Prepare for Lighting

The subsequent steps prepare for lighting:

```cpp
// store normalized light vector
float4 light = normalize(LightVec);

// calculate half-angle vector
float4 eye = float4(0.0, 0.0, 1.0, 1.0);
float4 half = normalize(light + eye);
```

At this point we have to ensure that all our vectors are normalized. We start by normalizing `LightVec`. Then, in preparation for specular lighting, we have to define the “half-angle” vector `half`, which is the vector halfway between the light and the eye vectors (that is, `(light + eye)/2`). We normalize `half`, so we don’t need to bother with the division by two, because it cancels out after normalization anyway. In this example, we assume that the eye is at `(0, 0, 1)`, but an application would typically pass
the eye position also as a uniform parameter, since it would be unchanged from vertex to vertex. We use Cg's inline vector construction capability to build a 4-component float vector that contains the eye position, and then we assign this value to eye.

Calculating the Vertex Color

Now we have to calculate the vertex color to output. In this example, we're going to calculate just a simple combination of diffuse and specular lighting:

```c
// calculate diffuse component
float diffuse = dot(normal, normalize(light));

// calculate specular component
float specular = dot(normal, normalize(half));
specular = pow(specular, 10);
```

Here we use the Cg Standard Library to perform dot products (using `dot()`) and exponentiation (using `pow()`). Remember to take advantage of the Standard Library to help speed up your development cycle. You can always tailor specific versions of the functions later, if you find that you can take shortcuts to improve performance.

Once the diffuse and specular lighting contributions `diffuse` and `specular` have been calculated, we need to modulate them with the object's material properties:

```c
// blue diffuse material
float4 diffuseMaterial = float4(0.0, 0.0, 1.0, 1.0);

// white specular material
float4 specularMaterial = float4(1.0, 1.0, 1.0, 1.0);

// combine diffuse and specular contributions
// and output final vertex color
Out.Color0 = diffuse * diffuseMaterial +
              specular * specularMaterial;
```

We define the object's diffuse material color as blue. In this case, we also define the alpha channel to be 1.0, in case blending is enabled, so that our object will be opaque. The object's specular highlight color is set to white, also with a 1.0 alpha value. Last, we modulate the lighting contributions with the material properties to get the final vertex color, and we assign it to the output connector's color field, `Out.Color0`.

Further Experimentation

Use `simple.cg` as a framework to try more advanced experiments, perhaps by adding more parameters to the program or by performing more complex calculations in the vertex program. Have fun experimenting!
This chapter provides a set of NV30 sample shaders written in Cg. Each shader comes with an accompanying snapshot, description, and source code.

Examples shown are

- Improved Skinning
- Improved Water
- Melting Paint
- MultiPaint
- Raytraced Refraction
- Skin
Improved Skinning

Description

This shader takes in a set of all the transformation matrices that can affect a particular bone. Each bone also sends in a list of matrices that affect it. There is then a simple loop that for each vertex goes through each bone that affects that vertex and transforms it. This allows just one Cg program to do the entire skinning for vertices affected by any number of bones, instead of having one program for one bone, another program for two bones, and so on.

Figure 4. Example of Improved Skinning
**Vertex Shader Cg Source Code**

```cgs
struct vert2frag
{
    float4 hPosition : HPOS;
    float4 color : COL0;
};

struct app2vert
{
    float4 position : ATTR0;
    float4 weights : ATTR1;
    float4 normal : ATTR2;
    float4 matrixIndices : ATTR5;
    float4 numBones : ATTR4;
};

vert2frag main(
    app2vert IN,
    uniform float4x4 modelViewProj : C0,
    const uniform float4 boneMatrices[90],
    uniform float4 color,
    uniform float4 lightPos)
{
    vert2frag OUT;

    float4 index = IN.matrixIndices;
    float4 weight = IN.weights;

    float4 position;
    float3 normal;

    float i;

    // loop over the number of bones
    for (i = 0; i < IN.numBones.x; i = i+1)
    {
        // transform the offset by bone i
        position = position + weight.x * float4(
            dot(boneMatrices[index.x+0], IN.position),
            dot(boneMatrices[index.x+1], IN.position),
            dot(boneMatrices[index.x+2], IN.position), 1);

        // transform the normal
        normal = normal + weight.x * float3(
            dot(boneMatrices[index.x+0].xyz, IN.normal.xyz),
```
dot(boneMatrices[index.x+1].xyz, IN.normal.xyz),
dot(boneMatrices[index.x+2].xyz, IN.normal.xyz));

    // shift over the index variable
    index = index.yzwx;
    weight = weight.yzwx;
}

normal = normalize(normal);

    // calculate homogenous position
OUT.hPosition = mul(modelViewProj, position);

    // calculate simple diffuse lighting
OUT.color = dot(normal, lightPos.xyz) * color;
return OUT;
}
Improved Water

Description

This demo gives the appearance that the viewer is surrounded by a large grid of vertices (because of the free rotation), but switching to wireframe or increasing the frustum angle makes it apparent that the vertices are a static mesh with the height, normal, and texture coordinates being calculated on-the-fly based on the direction and height of the viewer. This technique allows for very GPU-friendly water animations because the static mesh can be precomputed. The vertices are displaced using sine waves, and in this example a loop is used to sum five sine waves to achieve realistic effects.

Figure 5. Example of Improved Water
Vertex Shader Cg Source Code

```
struct app2vert
{
  float4 Position     : ATTR0;
};

struct vert2frag
{
  float4 HPosition    : HPOS;
  float4 TexCoord0    : TEX0;
  float4 TexCoord1    : TEX1;
  float4 Color0       : COL0;
  float4 Color1       : COL1;
};

void calcWave(out float disp,
              out float2 normal,
              float dampening,
              float3 viewPosition,
              float waveTime,
              float height,
              float frequency,
              float2 waveDirection)
{
  float   distance1 = dot(viewPosition.xy, waveDirection);
  distance1 = frequency * distance1 + waveTime;
  disp = height * sin(distance1) / dampening;
  normal = -cos(distance1) * height * frequency *
                (waveDirection.xy) / (.4*dampening);
}

vert2frag main(
    app2vert IN,
    uniform float4x4 ModelViewProj,
    uniform float4x4 ModelView,
    uniform float4x4 ModelViewIT,
    uniform float4x4 TextureMat,
    uniform float   Time,
    uniform float4  Wave1,
    uniform float4  Wave1Origin,
    uniform float4  Wave2,
    uniform float4  Wave2Origin,
    const uniform float4 WaveData[5])
{
```
}
vert2frag OUT;

float4 position = float4(IN.Position.x, 0, IN.Position.y,1);
float4 normal = float4(0,1,0,0);
float dampening = (dot(position.xyz, position.xyz)/1000+1);
float disp;
float2 norm;

// use a loop to add up some sine waves
float i;
for (i = 0; i < 5; i = i + 1)
{
  float waveTime  = Time.x * WaveData[i].z;
  float frequency = WaveData[i].z;
  float height    = WaveData[i].w;
  float2 waveDir  = WaveData[i].xy;
  calcWave(disp, norm, dampening, IN.Position.xyz,
           waveTime, height, frequency, waveDir);
  position.y = position.y + disp;
  normal.xz = normal.xz + norm;
}

// write out homogenous position
OUT.HPosition = mul(ModelViewProj, position);

// transform normal into eye-space
normal.xyz = normalize(normal.xyz);
normal = mul(ModelViewIT, normal);

// get a vector from the vertex to the eye
float3 eyeToVert = mul(ModelView, position).xyz;
eyeToVert = normalize(eyeToVert);

// calculate the reflected vector for cubemap look-up
float4 reflected = mul(TextureMat,
                       reflect(eyeToVert, normal.xyz).xyzz);
OUT.TexCoord0 = reflected;
OUT.TexCoord1 = reflected;

// Calculate a fresnel term (note that f0 = 0)
float fres = 1+dot(eyeToVert, normal.xyz);
fres = pow(fres, 5);

// set the two color coefficients
// (the magic constants are arbitrary)
OUT.Color0 = (fres*1.4 + min(reflected.y, 0)).xxxx +
    float4(.2, .3, .3, 0);
OUT.Color1 = (fres*1.26).xxxx;

return OUT;
Melting Paint

Description

This shader uses an environment map with procedurally modified texture lookups to create a melting effect on the surface texture (the NVIDIA logo in this example). The reflection vector is shifted using a noise function, giving the appearance of a bumpy surface. The surface texture's texture coordinates are shifted in a time-dependent manner, also based on a noise texture.

Figure 6. Example of Melting Paint
// define inputs from application
struct app2vert
{
    float4 Position : ATTR0;
    float4 Normal : ATTR2;
    float4 Color0 : ATTR3;
    float4 TexCoord0 : ATTR8;
};

struct vert2frag
{
    float4 HPosition : HPOS;
    float3 OPosition : TEX2;
    float3 EPosition : TEX3;
    float3 Normal : TEX1;
    float3 TexCoord0 : TEX0;
    float4 Color0 : COL0;

    float3 LightPos : TEX4;
    float3 ViewerPos : TEX5;
};

vert2frag main(app2vert In,
   uniform float4x4 ModelViewProj : C0,
   uniform float4x4 ModelViewIT : C4,
   uniform float4x4 ModelView : C8,
   uniform float4x4 ModelViewI : C12,
   uniform float4 ViewerPos,
   uniform float4 LightPos)
{
    vert2frag Out;

    // Vertex positions:
    // In clip space
    Out.HPosition = mul(ModelViewProj, In.Position);
    // In object space
    Out.OPosition = In.Position.xyz;
    // In eye space

    Out.Normal = normalize(In.Normal.xyz);
    // Copy the texture coordinates
    Out.TexCoord0 = In.TexCoord0.xyz;

    // Generate a white color
Out.Color0 = LightPos;//In.Color0;

Out.LightPos = mul(ModelViewI, LightPos).xyz;
Out.ViewerPos = mul(ModelViewI, float4(0,0,0,1)).xyz;

return Out;

Pixel Shader Cg Source Code

struct vert2frag
{
    float4 HPosition : HPOS;
    float3 OPosition : TEX2;
    float3 EPosition : TEX3;
    float3 Normal : TEX1;
    float3 TexCoord0 : TEX0;
    float4 Color0 : COL0;

    float3 LightPos : TEX4;
    float3 ViewerPos : TEX5;
};

struct frag2frame
{
    half4 COL;
};

void calcLighting(out float diffuse,
    out float specular,
    float3 normal,
    float3 fragPos,
    float3 lightPos,
    float3 eyePos,
    float specularExp)
{
    float3 light = lightPos - fragPos;
    float len = length(light);
    light = light / len;

    float3 eye = normalize(eyePos - fragPos);
    float3 halfVec = normalize((eye + light) / 2);
    len = len * .3;

    diffuse = dot(light, normal);
diffuse = clamp(diffuse, 0, 1);
diffuse = diffuse / len;

specular = dot(halfVec, normal);
specular = specular * sign(diffuse);
specular = clamp(specular, 0, 1);
specular = pow(specular, specularExp);
specular = specular / len;

}

frag2frame main(vert2frag IN,
    uniform float4 LightPos,
    uniform texobj3D noise_map : texunit0,
    uniform texobj2D nv_map : texunit1,
    uniform texobjCUBE cube_map : texunit2,
    uniform float4 interpolate
)
{
    frag2frame OUT;
    float diffuse, specular;

    float3 biVariate = float3(IN.OPosition.x-IN.OPosition.z,
                              IN.OPosition.y+IN.OPosition.z, 0);

    float3 uniVariate = float3(IN.OPosition.x+IN.OPosition.z,
                              0, 0);

    float3 normal = normalize(IN.Normal);
    float3 noiseSum = f3tex3D(noise_map, biVariate/3)/12 +
                     f3tex3D(noise_map,
                     float3((IN.OPosition.x+IN.OPosition.z)*6,
                         IN.OPosition.y/2, 0))/18 + f3tex3D(noise_map,
                         biVariate*6)/18;
    normal = normalize(normal + noiseSum);

    float3 eye = normalize(IN.ViewerPos - IN.OPosition);
    calcLighting(diffuse, specular, normal,
                  IN.OPosition, IN.LightPos, IN.ViewerPos, 32);

    float3 nvShift = f3tex3D(noise_map, uniVariate/3)/2 +
                    f3tex3D(noise_map, uniVariate)/4 +
                    f3tex3D(noise_map, biVariate*3)/16;
    nvShift.x = nvShift.x*nvShift.x*interpolate.x*3;
    nvShift.y = 0;
    biVariate = float3(IN.OPosition.x-IN.OPosition.z,
                      IN.OPosition.y+IN.OPosition.z, 0);
float2 texCoord = biVariate.xy/4 + float2(1.1,.5) + 
nvShift.yx + float2(0,interpolate.x/8);

float3 nvDecal = f3tex2D(nv_map,
    float2(1-texCoord.x, texCoord.y)) *
    (1-interpolate.x*.7).xxx;

float3 lightMetal = f3texCUBE(cube_map,
    reflect(normal, eye));

float3 darkMetal = (diffuse * float3(.5,.25,0) +
    specular * float3(.7,.4,0));

OUT.COL.xyz = ((1-nvDecal.x) * lightMetal +
    nvDecal.x * darkMetal);

OUT.COL.w = 1;

return OUT;
}
MultiPaint

Description

MultiPaint presents a single-pass solution to a common production problem: mixing multiple kinds of materials on a single polygonal surface. MultiPaint provides a simple BRDF (bidirectional reflectance distribution function) that is still complex enough to represent many common metallic and dielectric surfaces, and controls all key factors of the variable BRDF through texturing. This permits you to create multiple materials without switching shaders, splitting your model, or resorting to multiple passes.

Uses for MultiPaint might include complex armor built of inlaid metals, woods, and stones—all modeled on a single, simple poly mesh; buildings composed of multiple types of stone, glass, and metal, expressed as simple cubes; cloth with inlaid metallic threads; or as in this demo, metal partially covered with peeling paint.

Using multiple BRDFs is common in the offline world, but rarely optimized; instead, two different shaders may be evaluated and their results blended using a mask texture or chained through if statements. For maximum real-time performance, MultiPaint instead integrates all of the key parts of the BRDFs as multiple painted textures so that only one pass through the shader is required to create the mixed appearance. This permits a single-pass shader containing diffuse, specular, and environmental lighting effects in a compact, fast-executing package.

Figure 7. Example of MultiPaint
// define inputs from vertex buffer
struct appin : application2vertex
{
    float4 Position : ATTR0;
    float4 UV : ATTR8;
    float4 Tangent : ATTR1;
    float4 Binormal : ATTR3;
    float4 Normal : ATTR2;
};

// output -- same struct is the input to "cg_multipaint.cg"
struct MultiPaintV2F {
    float4 HPosition : HPOS;    // clip space position
    float4 TexCoords : TEX0;    // base ST coordinates
    float3 OPosition : TEX1;    // obj-coords location
    float3 Normal : TEX2;       // eye-space normal
    float3 VPosition : TEX3;    // viewer pos in obj coords
    float3 T : TEX4;            // tangent in obj coordinates
    float3 B : TEX5;            // binormal in obj coordinates
    float3 N : TEX6;            // normal in obj coordinates
    float4 LightVecO : TEX7;    // light direction in
                               // obj coords
    float4 Color0 : COL0;       // color potentially passed
                                // from vertices
};

MultiPaintV2F main(appin IN,

uniform float4x4 ModelViewProj : C0,
uniform float4x4 ModelViewIT : C4,
uniform float4x4 ModelView : C8,
uniform float4x4 ModelViewI : C12,
uniform float4 TexRepeats,
uniform float4 ViewerPos,
uniform float4 LightVec)    // in EYE coords
{
    MultiPaintV2F OUT;
    // calculate clip space position for rasterizer use
    OUT.HPosition = mul(ModelViewProj, IN.Position);

    // object space -- just pass through
    OUT.OPosition = IN.Position.xyz;

    // transform normal to view space
    OUT.Normal = normalize(mul(ModelViewIT, IN.Normal).xyz);
    OUT.TexCoords = IN.UV * TexRepeats;
}
OUT.N = normalize(IN.Normal.xyz);   // obj space
OUT.T = IN.Tangent.xyz;   // obj space
OUT.B = IN.Binormal.xyz;   // obj space

// transform from eye coordinates to object coordinates
OUT.VPosition = mul(ModelViewI, float4(0,0,0,1)).xyz;

// transform from eye space to object space
OUT.LightVecO = mul(ModelViewI, LightVec);

// OUT.ColorO = IN.Color;
return OUT;

Pixel Shader Cg Source Code

// input -- same struct is the output from "cg_multipaintVP.cg"
struct MultiPaintV2F {
    float4 HPosition    : HPOS; // clip space position
    float4 TexCoords    : TEX0; // base ST coordinates
    float3 OPosition    : TEX1; // obj-coords location
    float3 Normal       : TEX2; // eye-space normal
    float3 VPosition    : TEX3; // viewer pos in obj coords
    float3 T            : TEX4; // tangent in obj coordinates
    float3 B            : TEX5; // binormal in obj coordinates
    float3 N            : TEX6; // normal in obj coordinates
    float4 LightVecO    : TEX7; // light direction in
                               // obj coords
    float4 Color0       : COL0; // color potentially passed
                               // from vertices
};

struct PixelOut {
    float4 COL;
    float DEPR;
};

// Helper Functions

// A handy way to visualize vectors at the surface,
// for debugging purposes.
// Not normally actually used in this program
//
float4 vector_as_color(float4 theVector)
{
    float4 nv = 0.5f+(0.5f*theVector);
    return nv;
}

// overloaded version for float3 vectors
float4 vector_as_color(float3 theVector)
{
    float4 nv =
        0.5f+(0.5f*float4(theVector.x,theVector.y,theVector.z,0.0f));
    return nv;
}
#include <Cg.cg>

void main()
{
    PixelOut OUT;
    float4 surfCol = f4tex2D(ColorMap, IN.TexCoords.xy);
    float4 material = f4tex2D(MaterialMap, IN.TexCoords.xy);
    float3 Nt = f3tex2D(NormalMap, IN.TexCoords.xy) - float3(0.5f, 0.5f, 0.5f);
    float specStr = material.SPEC_STR * SpecData.MAXSPEC;
    float specPower = SpecData.MINPOWER + material.NORM_SPEC_EXPON * (SpecData.MAXPOWER - SpecData.MINPOWER);

    // Calculation...
}
// calculate normalized vectors
float3 Vn = -normalize(IN.VPosition - IN.OPosition);
float3 Ln = normalize(IN.LightVecO).xyz;
float3 Nb = normalize(BumpData.BUMP_SCALE *
    (Nt.x * IN.T + Nt.y * IN.B) +
    (Nt.z * IN.N));

// calculate diffuse color
float diff = max(0.0f, -dot(Ln, Nb));
float4 diffResult = diff * surfCol;
float isLit = (diff > 0.0f) ? 1.0f : 0.0f;

// calculate specular color
float3 Hn = normalize(Vn + Ln);
float spec = pow(abs(dot(Hn, Nb)), specPower) * specStr;
float4 WHITE = float4(1f, 1f, 1f, 1f);
float4 specCol = lerp(WHITE, surfCol, material.METALNESS);
float4 specResult = (spec * isLit) * specCol;
float3 reflVect = reflect(Vn, Nb);
float4 reflColor = f4texCUBE(EnvMap, reflVect);
float fakeFresnel = ReflData.FRESNEL_MIN +
    ReflData.FRESNEL_MAX * pow((1.0f - dot(-Vn, IN.N)),
    ReflData.FRESNEL_EXPON);
float4 paintShine = fakeFresnel * reflColor;
float4 metalShine = surfCol * reflColor;
float4 shineCol = ReflData.REFL_STRENGTH *
    lerp(paintShine, metalShine, material.METALNESS);
float4 finalColor = diffResult + shineCol;

// finalColor = vector_as_color(Ln);
finalColor = specResult + diffResult + shineCol;
finalColor.w = 1.0f;
OUT.COL = finalColor;
return OUT;
}
Ray-Traced Refraction

Description

This shader presents a method for adding high-quality details to small objects using a single-bounce, ray-traced pass. In this example, the polygonal surface is sampled and a refraction vector is calculated. This vector is then intersected with a plane that is defined as being perpendicular to the object's x-axis. The intersection point is calculated and used as texture indices for a painted iris.

The demo permits varying the index of refraction, the depth and density of the lens. Note that the choice of geometry is arbitrary—this sample is a sphere, but any polygonal model can be used.

![Example of Ray-Traced Refraction](image)

Figure 8. Example of Ray-Traced Refraction

Vertex Shader Cg Source Code

```cg
// define inputs from vertex buffer
struct appin : application2vertex
{
    float4 Position : ATTR0;
    float4 Normal : ATTR2;
};
```
// output -- same struct is the input to "cg_eye.cg"
struct EyeV2F {
    float4 HPosition : HPOS; // clip space position
    float3 OPosition : TEX1; // obj coords location
    float3 VPosition : TEX3; // viewer pos in obj
                   // coordinates
    float3 N        : TEX6; // normal in obj coordinates
    float4 LightVecO : TEX7; // light direction in
                   // obj coords
};

// subfields in "BallData"
#define RADIUS x
#define IRIS_DEPTH y
#define ETA z
#define LENS_DENSITY w

EyeV2F main(appin IN,
    uniform float4x4 ModelViewProj  : C0,
    uniform float4x4 ModelViewIT    : C4,
    uniform float4x4 ModelView      : C8,
    uniform float4x4 ModelViewI    : C12,
    uniform float4 BallData,
       // components: (radius,irisDepth,eta,lensDensity)
    uniform float4 ViewerPos,
    uniform float4 LightVec) // in EYE coords
{
    EyeV2F OUT;

    // calculate clip space position for rasterizer
    OUT.HPosition = mul(ModelViewProj, IN.Position);

    // pass through the object space coordinates
    OUT.OPosition = IN.Position.xyz;

    // output normal in object space
    OUT.N = normalize(IN.Normal.xyz);

    // transform from eye coordinates to object coordinates
    OUT.VPosition = mul(ModelViewI, float4(0,0,0,1)).xyz;

    // transform from eye coordinates to object coordinates
    OUT.LightVecO = normalize(mul(ModelViewI, LightVec));

    return OUT;
}
// Helper Functions

float3 desaturate(float3 origColor, float saturation)
{
    float3 newCol = (1.0 - saturation).xxx + origColor * saturation;
    return newCol;
}

// Assume ray direction is normalized.
// Vector "planeEq" is encoded float3(A,B,C,D) where
// (Ax+By+Cz+D)=0 and float3(A,B,C) has been normalized
//
float intersect_plane(float3 rayOrigin, float3 rayDir, float4 planeEq)
{
    float3 planeN = planeEq.xyz;
    float denominator = dot(planeN, rayDir);
    float result = -1.0; // if return value negative,
    // ignore any ray because it's
    // BEHIND the eye
    // d==0 -> parallel || d>0 -> faces away
    if (denominator < 0.0) {
        float top = dot(planeN, rayOrigin) + planeEq.w;
        result = -top/denominator;
    }
    return result;
}

// A handy way to visualize vectors at the surface,
// for debugging purposes.
// Not normally actually used in this program

float3 vector_as_color(float4 theVector)
{
    float4 nv = 0.5f + (0.5f * theVector);
    return nv.rgb;
}

// overloaded version for float3 vectors
float3 vector_as_color(float3 theVector)
{
    float4 nv = 0.5f + (0.5f * float4(theVector.x, theVector.y, theVector.z, 1.0f));
    return nv.rgb;
}
theVector.z,
0.0f));
return nv.rgb;
}

///////////////////////////////////////////////
// Actual Fragment Program Here //
///////////////////////////////////////////////

// channels in our material map:
#define SPEC_STR x
#define METALNESS y
#define NORM_SPEC_EXPON z

// subfields in "BallData"
#define RADIUS x
#define IRIS_DEPTH y
#define ETA z
#define LENS_DENSITY w

// subfields in "SpecData"
#define PHONG x
#define GLOSS1 y
#define GLOSS2 z
#define DROP w

// input -- same struct is the output from "cg_eyeVP.cg"
struct EyeV2F {
    float4 HPosition    : HPOS; // clip space position
    float3 OPosition    : TEX1;   // obj coords location
    float3 VPosition    : TEX3; // viewer pos in obj
    // coordinates
    float3 N            : TEX6; // normal in obj coordinates
    float4 LightVecO    : TEX7;   // light direction in
    // obj coords
    }
};

struct PixelOut {
    float4 COL;
    float DEPR;
};

PixelOut main(
    EyeV2F IN,
    uniform sampler2D ColorMap : texunit0, // color
    uniform float4 BallData,
    // components: {radius,irisDepth,eta,lensDensity}
uniform float4 GlossData,
       // components: {phongExp,gloss1,gloss2,drop}
uniform float3 AmbiColor,
uniform float3 DiffColor,
uniform float3 SpecColor,
uniform float3 LensColor,
uniform float3 BgColor

) {
    float3 baseTex = float3(1.0f,1.0f,1.0f); // user parameter
    float GRADE = 0.05; // user parameter
    float3 yAxis = float3(0,1,0);
    float3 xAxis = float3(1,0,0);
    float3 ballCtr = float3(0,0,0);
    // all of these actually evaluate to constants
    // and could be done in VP or on CPU
    // calculate radius of pupil disk
    float irisSize = BallData.RADIUS * 
        sqrt(1.0-BallData.IRIS_DEPTH * 
            BallData.IRIS_DEPTH);
    float irisScale = 0.3333f/max(0.01,irisSize);
    float irisDist = BallData.RADIUS*BallData.IRIS_DEPTH;
    float3 pupilCenter = ballCtr + float3(irisDist,0,0);
    float D = -dot(pupilCenter,xAxis);
    // if x axis, returns simple -irisDist
    float slice = IN.OPosition.x - irisDist;
    float4 planeEquation = float4(xAxis.x,xAxis.y,xAxis.z,D);

    // okay, now actual per-pixel data enters the stage
    float3 Vn = normalize(IN.OPosition - IN.VPosition);
    // view vector TO surface
    float3 Nf = normalize(IN.N);
    float3 Ln = IN.LightVecO.xyz;
    // (It's already normalized by vertex program)
    float3 DiffLight = DiffColor * max(0, dot(Nf, -Ln)).xxx;
    float3 missColor = AmbiColor + baseTex * DiffLight;
    float3 DiffPupil = AmbiColor + max(0, dot(xAxis, -Ln)).xxx;
    // full-blast float3
float3 halfAng = normalize(-Ln-Vn);
float ndh = abs(dot(Nf,halfAng));
float spec1 = pow(ndh,GlossData.PHONG);
// “spec1” would provide us a nice Blinn/Phong-style
// highlight. By adding a threshold, we can create the
// *illusion* of the glossy reflection of a large
// circular light source. We use smoothstep() to
// get a rolloff as “spec1” crosses the interval
// between GlossData.GLOSS1 and GlossData.GLOSS2...
float s2 = smoothstep(GlossData.GLOSS1,
    GlossData.GLOSS2,
    spec1);
// ...and then use that value to knock-down “spec1” values
// at or below GlossData.GLOSS1 by a percentage which
// was passed in GlossData.DROP
spec1 = spec1 * ((1.0-GlossData.DROP) + GlossData.DROP*s2);

// params
float3 SpecularLight = SpecColor * spec1.xxx;

float3 hitColor = missColor;

if (slice >= 0.0f) {
    float gradedEta = BallData.ETA;
    gradedEta = 1.0/gradedEta; // test hack
    float3 faceColor = BgColor; // blown out - go to BG color
    float c1 = dot(-Vn,Nf);
    float cs2 = 1.0f-gradedEta*gradedEta*(1.0f-c1*c1);
    // faceColor = float3(cs2.x,-cs2.x,c1);
    if (cs2 >= 0.0) {
        float3 refVector = gradedEta*Vn+
            ((gradedEta*c1-sqrt(cs2))*Nf);
        // now let's intersect with the iris plane
        float irisT = intersect_plane(IN.OPosition,
            refVector,
            planeEquation);
        float fadeT = irisT * BallData.LENS_DENSITY;
        fadeT = fadeT * fadeT;
        faceColor = DiffPupil.xxx; // temporary
        if (irisT > 0) {
            float3 irisPoint = IN.OPosition + irisT*refVector;
            float3 irisST = (irisScale*irisPoint) +
                float3(0.0f,0.5f,0.5f);
            faceColor = f3tex2D(ColorMap,irisST.yz);
        }
    }
}
faceColor = lerp(faceColor,LensColor,fadeT);
   hitColor = lerp(missColor,
                   faceColor,
                   smoothstep(0.0,GRADE,slice));

hitColor = hitColor + SpecularLight;
// hitColor = DiffLight + SpecularLight;
PixelOut OUT;
OUT.COL = float4(hitColor.x, hitColor.y, hitColor.z, 1.0f);
   return OUT;
Skin

Description

This effect demonstrates some techniques for rendering skin ranging from simple Blinn-Phong Bump-Mapping to more complex Subsurface Scattering lighting models. It also illustrates the use of “Rim” lighting and simple translucency for capturing some of the more subtle properties of skin resulting from complex, non-local lighting interactions. Finally, it shows how the various techniques can be combined to produce compelling, stylized skin.

Figure 9. Example of Skin

Pixel Shader Cg Source Code

```c
struct fragin
{
    float2 texcoords     : TEX0;
    float4 shadowcoords  : TEX1;
    float4 tangentToEyeMat0 : TEX4;
    float3 tangentToEyeMat1 : TEX5;
    float3 tangentToEyeMat2 : TEX6;
    float3 eyeSpacePosition : TEX7;
};
```
struct FragOut
{
    half4 COL;
};

float3 hqphase( float3 v1, float3 v2, float3 g )
{
    float costheta;
    float3 g2;
    float3 gtemp;

    costheta = dot( -v1, v2 );
    g2 = g*g;
    gtemp = 1.0.xxx + g2 - 2.0*g*costheta;
    gtemp = pow( gtemp, 1.5.xxx );
    gtemp = (1.0.xxx - g2) / gtemp;
    return gtemp;
}

// Computes the single-scattering approximation to scattering
// from a one-dimensional volumetric surface.
float3 singleScatter( float3 wi, float3 wo, float3 n, float3 g,
    float3 albedo, float thickness )
{
    float win = abs(dot(wi,n));
    float won = abs(dot(wo,n));
    float  eterm;
    float3  result;

    eterm = 1.0 - exp( -((1./win)+(1./won))*thickness) );
    result = eterm * (albedo * hqphase( wo, wi, g ) / (win + won));

    return result;
}

// i is the incident ray
// n is the surface normal
// eta is the ratio of indices of refraction
// r is the reflected ray
// t is the transmitted ray

float fresnel( float3 i, float3 n, float eta, out float3 r, out
    float3 t )
{
    float result;
    float c1;
    float cs2;
float tflag;

// Refraction vector courtesy Paul Heckbert.
c1 = dot(-i,n);
cs2 = 1.0-eta*eta*(1.0-c1*c1);
tflag = (float) (cs2 >= 0.0);
t = tflag * (((eta*c1-sqrt(cs2))*n) + eta*i);
// t is already unit length or (0,0,0)

// Compute Fresnel terms
// (From Global Illumination Compendium.)
float ndott;
float cosr_div_cosi;
float cosi_div_cosr;
float fs;
float fp;
float kr;

ndott = dot(-n,t);
cosr_div_cosi = ndott / c1;
cosi_div_cosr = c1 / ndott;
fs = (cosr_div_cosi - eta) / (cosr_div_cosi + eta);
fs = fs * fs;
fp = (cosi_div_cosr - eta) / (cosi_div_cosr + eta);
fp = fp * fp;
kr = 0.5 * (fs+fp);
result = tflag*kr + (1.-tflag);
r = reflect( i, n );

return result;

FragOut main( fragin In,
uniform sampler2D tex0,
uniform sampler2D tex1,
uniform sampler2D tex2,
uniform sampler2D tex3,
uniform float3 eyeSpaceLightPosition,
uniform float thickness,
uniform float4 ambient )
{
  FragOut O;

  float bscale = In.tangentToEyeMat0.w;

  float eta = (1.0/1.4);
float m = 34.;     // specular exponent
float4 lightColor = { 1, 1, 1, 1 };  // light color
float4 sheenColor = { 1, 1, 1, 1 };  // surface sheen color
float4 skinColor = f4tex2D( tex1, In.texcoords );
float3 g = { 0.8, 0.3, 0.0 };
float3 albedo = { 0.8, 0.5, 0.4 };

float4 oiliness = 0.9 * f4tex2D( tex2, In.texcoords);

float3 v = normalize( -In.eyeSpacePosition );
float3 l = normalize( eyeSpaceLightPosition – In.eyeSpacePosition );
float3 h = normalize( v + l );

float3 tangentSpaceNormal = f3tex2D( tex0, In.texcoords );
float3 bumpscale = { bscale, bscale, 1.0 };
tangentSpaceNormal = tangentSpaceNormal * bumpscale;

float3 n;
n[0] = dot( In.tangentToEyeMat0.xyz, tangentSpaceNormal );
n[1] = dot( In.tangentToEyeMat1, tangentSpaceNormal );
n[2] = dot( In.tangentToEyeMat2, tangentSpaceNormal );

float ndotl = max( dot(n,l), 0 );   // clamp 0 to 1
float ndoth = max( dot(n,h), 0 );   // clamp 0 to 1
float flag  = (float)(ndotl > 0);
// if (ndotl <= 0) specular = 0

float4 oil;            // Compute oil, sheen, subsurface scattering contributions.
float4 sheen;
float4 subsurf;
float Kr, Kr2;
float Kt, Kt2;
float3 T, T2;
float3 R, R2;
// Compute fresnel at sheen layer and then ramp it up a bit.
Kr = fresnel( -v, n, eta, R, T );
Kr = smoothstep( 0.0, 0.5, Kr );
Kt = 1.0 - Kr;

// Compute the refracted light ray and the refraction
// coefficient.
Kr2 = fresnel( -l, n, eta, R2, T2 );
Kr2 = smoothstep( 0.0, 0.5, Kr2 );
Kt2 = 1.0 - Kr2;

// For the oil contribution modulate the oiliness mask by a
// specular term.
oil = 0.5 * oiliness * pow( ndoth, m );

// For the sheen contribution, modulate the Fresnel term by
// the sheen color times specular. Modulate by the additional
// diffuse term to soften it a bit.
sheen = 2.5 * Kr * sheenColor *
( ndotl * (0.2 + pow( ndoth, m )) );

// Compute single scattering approximation to subsurface
// scattering. Here we compute 3 scattering terms in
// simultaneously and the results end up in the x,y,z
// components of a float3. Using 3 terms approximates the
// distribution of multiply-scattered light. For details
// see: Matt Pharr’s SIGGRAPH 2001 RenderMan course notes
// "Layered Media for Surface Shaders".
float3 temp = singleScatter( T2, T, n, g, albedo, thickness );
subsurf = 2.5 * skinColor * ndotl * Kt * Kt2 *
(temp.x+temp.y+temp.z);

// Add contributions from oil, sheen, and subsurface
// scattering and modulate by the light color and the result
// of a shadow map lookup.
O.COL = lightColor * fltexcompare2D( tex3, In.shadowcoords ) *
(oil + sheen + subsurf);

return O;
This chapter provides a set of NV2X sample shaders written in Cg. Each shader comes with an accompanying snapshot, description, and source code.

Examples shown are:
- Anisotropic Lighting
- Bump Dot3x2 Diffuse and Specular
- Bump Reflection Mapping
- Fresnel
- Grass
- Refraction
- Shadow Mapping
- Shadow Volume Extrusion
- Sine Wave Demo
Anisotropic Lighting

Description

The anisotropic lighting effect (Figure 4) shows the vertex program’s half-angle vector calculation. It uses (H dot N) and (L dot N) per-vertex to look up into a 2D texture to achieve interesting lighting effects.

Figure 10. Example of Anisotropic Lighting

(See Appendix D for a color version of this illustration)
Vertex Shader Cg Source Code

```cg
struct appdata {
    float3 Position : POSITION;
    float3 Normal : NORMAL;
};

struct vpconn {
    float4 Hposition : POSITION;
    float4 TexCoord0 : TEXCOORD0;
};

vpconn main(appdata IN,
    uniform float4x4 WorldViewProj,
    uniform float3x3 WorldIT,
    uniform float3x4 World,
    uniform float3 LightVec,
    uniform float3 EyePos)
{
    vpconn OUT;

    float3 worldNormal = normalize(mul(WorldIT,  IN.Normal));

    //build float4
    float4 tempPos;
    tempPos.xyz = IN.Position.xyz;
    tempPos.w   = 1.0;

    //compute world space position
    float3 worldSpacePos = mul(World, tempPos);

    //vector from vertex to eye, normalized
    float3 vertToEye = normalize(EyePos - worldSpacePos);

    //h = normalize(l + e)
    float3 halfAngle = normalize(vertToEye + LightVec);

    OUT.TexCoord0.x = max(dot(LightVec,worldNormal),0.0);
    OUT.TexCoord0.y = max(dot(halfAngle,worldNormal),0.0);

    // transform into homogeneous-clip space
    OUT.HPosition = mul(WorldViewProj, tempPos);

    return OUT;
}
```
Bump Dot3x2 Diffuse and Specular

The bump dot3x2 diffuse and specular effect mixes bump mapping with diffuse and specular lighting based on the **texm3x2tex** DirectX 8 pixel shader instruction (**DOT_PRODUCT_TEXTURE_2D** in OpenGL). This instruction computes the dot product of the normal and the light vector, corresponding to the diffuse light component, and the dot product of the normal and the half angle vector, corresponding to the specular light component. This results into two scalar values that are used as texture coordinates to look up a 2D illumination texture containing the diffuse color and the specular term in its alpha component. Since the normal fetched from the normal map is in tangent space, both the light vector and the half angle vector are transformed to this space by the vertex shader (Figure 11).

![Example of Bump Dot3x2 Diffuse and Specular](image)

(See Appendix D for a color version of this illustration)
Vertex Shader Cg Source Code

```cg
struct a2v {
    float4 Position : POSITION; //in object space
    float3 Normal : NORMAL; //in object space
    float2 TexCoord : TEXCOORD0;
    float3 T : TEXCOORD1; //in object space
    float3 B : TEXCOORD2; //in object space
    float3 N : TEXCOORD3; //in object space
};

struct v2f {
    float4 Position : POSITION; //in projection space
    float4 Normal : COLOR0; //in tangent space
    float4 LightVectorUnsigned : COLOR1; //in tangent space
    float3 TexCoord0 : TEXCOORD0;
    float3 TexCoord1 : TEXCOORD1;
    float4 LightVector : TEXCOORD2; //in tangent space
    float4 HalfAngleVector : TEXCOORD3; //in tangent space
};

v2f main(a2v IN,
         uniform float4x4 WorldViewProj,
         uniform float4 LightVector, //in object space
         uniform float4 EyePosition //in object space
) {
    v2f OUT;

    // pass texture coordinates for
    // fetching the diffuse map
    OUT.TexCoord0.xy = IN.TexCoord.xy;

    // pass texture coordinates for
    // fetching the normal map
    OUT.TexCoord1.xy = IN.TexCoord.xy;

    // compute the 3x3 transform from
    // tangent space to object space
    float3x3 objToTangentSpace;
    objToTangentSpace[0] = IN.T;
    objToTangentSpace[1] = IN.B;
    objToTangentSpace[2] = IN.N;

    // transform normal from
    // object space to tangent space
    OUT.Normal.xyz = 0.5*mul(objToTangentSpace, IN.Normal) + 0.5.xxx;
}
```
// transform light vector from
// object space to tangent space
float3 lightVectorInTangentSpace =
mul(objToTangentSpace, LightVector.xyz);
OUT.LightVector.xyz = lightVectorInTangentSpace;
OUT.LightVectorUnsigned.xyz =
0.5 * lightVectorInTangentSpace + 0.5.xxx;

// compute view vector
float3 viewVector =
normalize(EyePosition.xyz - IN.Position.xyz);

// compute half angle vector
float3 halfAngleVector =
normalize(LightVector.xyz + viewVector);

// transform half-angle vector from
// object space to tangent space
OUT.HalfAngleVector.xyz = mul(objToTangentSpace, halfAngleVector);

// transform position to projection space
OUT.Position = mul(WorldViewProj, IN.Position);

return OUT;

}
// fetch base color
float4 color = tex2D(DiffuseMap);

// fetch bump normal and expand it to [-1,1]
float4 bumpNormal = expand(tex2D(NormalMap));

// compute the dot product between
// the bump normal and the light vector,
// compute the dot product between
// the bump normal and the half angle vector,
// fetch the illumination map using
// the result of the two previous dot products
// as texture coordinates

// returns the diffuse color in the
// color components and the specular color in the
// alpha component
float4 illumination = tex2D_dp3x2(IlluminationMap,
                                      IN.LightVector, bumpNormal);

// expand iterated normal to [-1,1]
float4 normal = expand(IN.Normal);

// compute self-shadowing term
float shadow = uclamp(4 * dot3(normal.xyz, 
                         IN.LightVectorUnsigned.xyz));

// compute final color
OUT.col = mad(Ambient, color, shadow * 
              mad(illumination, color, illumination.wwww));

return OUT;
Bump Reflection Mapping

Description

This effect mixes bump mapping and reflection mapping based on the `texm3x3vspec` DirectX 8 pixel shader instruction (`DOT_PRODUCT_REFLECT_CUBE_MAP` in OpenGL). This instruction computes three dot products to transform the normal fetched from the normal map into the environment cube space, reflects the transformed normal with respect to the eye vector and fetches a cube map to get the final color. The vertex shader is responsible for computing the transform matrix and the eye vector (Figure 12).

![Example of Bump Reflection Mapping](image)

Figure 12. Example of Bump Reflection Mapping

(See Appendix D for a color version of this illustration)
struct a2v {
    float4 Position : POSITION;  // in object space
    float2 TexCoord : TEXCOORD0;
    float3 T : TEXCOORD1;   // in object space
    float3 B : TEXCOORD2;   // in object space
    float3 N : TEXCOORD3;   // in object space
};

struct v2f {
    float4 Position : POSITION; // in projection space
    float4 TexCoord : TEXCOORD0;

    // first row of the 3x3 transform
    // from tangent to cube space
    float4 TangentToCubeSpace0 : TEXCOORD1;

    // second row of the 3x3 transform
    // from tangent to cube space
    float4 TangentToCubeSpace1 : TEXCOORD2;

    // third row of the 3x3 transform
    // from tangent to cube space
    float4 TangentToCubeSpace2 : TEXCOORD3;
};

v2f main(a2v IN,
    uniform float4x4 WorldViewProj,
    uniform float3x4 ObjToCubeSpace,
    uniform float3 EyePosition, // in cube space
    uniform float BumpScale)
{
    v2f OUT;

    // pass texture coordinates for
    // fetching the normal map
    OUT.TexCoord.xy = IN.TexCoord.xy;

    // compute 3x3 transform from tangent space to object space
    float3x3 objToTangentSpace;

    // first rows are the tangent and binormal
    // scaled by the bump scale
    objToTangentSpace[0] = BumpScale * IN.T;
    objToTangentSpace[1] = BumpScale * IN.B;
    objToTangentSpace[2] = IN.N;
// compute the 3x3 transform from
// tangent space to cube space:
// TangentToCubeSpace
//    = object2cube * tangent2object
//    = object2cube * transpose(objToTangentSpace)
// (since the inverse of a rotation is its transpose)
//
// So a row of TangentToCubeSpace is the transform by
// objToTangentSpace of the corresponding row of
// ObjToCubeSpace

OUT.TangentToCubeSpace0.xyz =
    mul(objToTangentSpace, ObjToCubeSpace[0].xyz);

OUT.TangentToCubeSpace1.xyz =
    mul(objToTangentSpace, ObjToCubeSpace[1].xyz);

OUT.TangentToCubeSpace2.xyz =
    mul(objToTangentSpace, ObjToCubeSpace[2].xyz);

// compute the eye vector
// (going from eye to shaded point) in cube space
float3 eyeVector =
    mul(ObjToCubeSpace, IN.Position) - EyePosition;

OUT.TangentToCubeSpace0.w = eyeVector.x;
OUT.TangentToCubeSpace1.w = eyeVector.y;
OUT.TangentToCubeSpace2.w = eyeVector.z;

// transform position to projection space
OUT.Position = mul(WorldViewProj, IN.Position);

return OUT;
Pixel Shader Cg Source Code

```c
struct v2f {
    float4 Position : POSITION; // in projection space
    float4 TexCoord : TEXCOORD0;

    // first row of the 3x3 transform
    // from tangent to cube space
    float4 TangentToCubeSpace0 : TEXCOORD1;

    // second row of the 3x3 transform
    // from tangent to cube space
    float4 TangentToCubeSpace1 : TEXCOORD2;

    // third row of the 3x3 transform
    // from tangent to cube space
    float4 TangentToCubeSpace2 : TEXCOORD3;
};

fragout main(v2f IN,
             uniform sampler2D NormalMap,
             uniform samplerCUBE EnvironmentMap,
             uniform float3 EyeVector)
{
    fragout OUT;

    // fetch the bump normal from the normal map
    float4 normal = tex2D(NormalMap);

    // transform the bump normal into cube space
    // then use the transformed normal and eye vector
    // to compute the reflection vector that is
    // used to fetch the cube map
    OUT.col = texCUBE_reflect_eye_dp3x3(EnvironmentMap,
                                         IN.TangentToCubeSpace0,
                                         IN.TangentToCubeSpace1,
                                         normal,
                                         EyeVector);

    return OUT;
}
```
Fresnel

Description

This effect computes a reflection vector to lookup into an environment map for reflections, and modulates this by a Fresnel term. The result is reflections only at grazing angles.

Figure 13. Example of Fresnel
(See Appendix D for a color version of this illustration)
Vertex Shader Cg Source Code

```cpp
struct app2vert
{
    float4 Position : POSITION;
    float4 Normal : NORMAL;
    float4 TexCoord0 : TEXCOORD0;
};

struct vert2frag
{
    float4 Hposition : POSITION;
    float4 Color0 : COLOR0;
    float4 TexCoord0 : TEXCOORD0;
};

vert2frag main(app2vert IN,
uniform float4x4 ModelViewProj : C0,
uniform float4x4 ModelView : C4,
uniform float4x4 ModelViewIT : C8)
{
    vert2frag OUT;

    OUT.HPosition = mul(ModelViewProj, IN.Position);
    float3 normal = IN.Normal.xyz;
    normal = normalize(mul(ModelViewIT, IN.Normal).xyz);
    float3 eyeToVert = normalize(mul(ModelView, IN.Position).xyz);

    // reflect the eye vector across
    // the normal vector for reflection
    OUT.TexCoord0.xyz = reflect(eyeToVert, normal);
    OUT.TexCoord0.w = 1;
    float f0 = .1;

    // compute the Fresnel term
    float oneMCosAngle = 1 + dot(eyeToVert, normal);
    oneMCosAngle = pow(oneMCosAngle, 5);
    OUT.Color0 = lerp(oneMCosAngle, 1, f0).xxxx;

    return OUT;
}
```
Grass

Description

This effect shows procedural animation of geometry using a Sine function, along with calculation of a normal for the procedurally deformed geometry (Figure 14).

Figure 14. Example of Grass
(See Appendix D for a color version of this illustration)
Vertex Shader Cg Source Code

```c
struct app2vert {
    float4 Position : POSITION;
    float4 Normal : NORMAL;
    float4 TexCoord0 : TEXCOORD0;
    float4 Color0 : COLOR0;
};

struct vertout {
    float4 Hposition : POSITION;
    float4 Color0 : COLOR0;
    float4 TexCoord0 : TEXCOORD0;
};

vertout main(app2vert IN, uniform float4x4 ModelViewProj, uniform float4x4 ModelView, uniform float4x4 ModelViewIT, uniform float4 Constants)
{
    vertout OUT;

    // we need to figure OUT what the position is
    float4 position = IN.Position;
    position.z = 0;
    position.y = 0;

    // add IN the actual base location of
    // the straw (stored IN Color0.xz)
    position.x = position.x + IN.Color0.x;
    position.z = position.z + IN.Color0.z;

    // figure OUT where the wind is coming from
    float4 origin = float4(20,0,20,0);
    float4 dir = position - origin;

    // find the intensity of the wind
    float inten = sin(Constants.x + .2*length(dir))*IN.Position.y;
    dir = normalize(dir);

    // we need to do some Bezier curve stuff here.
    float4 ctrl1 = float4(0,0,0,0);
    float4 ctrl2 = float4(0,IN.Color0.y/2,0,0);
    float4 ctrl3 = float4(dir.x*inten,
                           IN.Color0.y,
                           dir.z*inten,
                           0);
```

```
// do the Bezier linear interpolation steps
float t = IN.Color0.w;
float4 temp = lerp(ctrl1, ctrl2, t);
float4 temp2 = lerp(ctrl2, ctrl3, t);
float4 result = lerp(temp, temp2, t);

// add IN the height and wind displacement components
position = position + result;
position.w = 1;

// transform for sending to the reg. combiners
OUT.HPosition = mul(ModelViewProj, position);

// calculate the texture coordinate
// from the position passed IN
OUT.TexCoord0 = float4((IN.Position.x + .05)*10, t, 1, 1);

// find the normal
// we need one more point to do a partial
temp = lerp(ctrl1, ctrl2, t+0.05);
temp2 = lerp(ctrl2, ctrl3, t+0.05);
float4 newResult = lerp(temp, temp2, t+0.05);

// do a crossproduct with a vector that
// is horizontal across the screen
float4 normal = cross((result - newResult).xyz, float3(1,0,0)).xyzz;
normal.w = 0;
normal = normalize(normal);

// calculate diffuse lighting off the normal
// that was just calculated
float4 lightPos = float4(0,5,15,0);
float4 lightVec = normalize(lightPos - position);
float diffuseInten = dot(lightVec, normal);

// Set up the final color
// The first term is a semi random term based
// on the total height of this straw
// The second term is the diffuse lighting component

OUT.Color0 = normalize(ctrl3) * diffuseInten*IN.Position.z;
return OUT;
Refraction

Description

This effect performs custom texture coordinate generation to compute a refracted vector per-vertex that is then used to look up in a cube map. Fresnel is also calculated to blend between reflection and refraction (Figure 15).

Figure 15. Example of Refraction
(See Appendix D for a color version of this illustration)
struct app2vert
{
    float4 Position : POSITION;
    float4 Normal : NORMAL;
    float4 TexCoord0 : TEXCOORD0;
};

struct vert2frag
{
    float4 Hposition : POSITION;
    float4 Color0 : COLOR0;
    float4 TexCoord0 : TEXCOORD0;
    float4 TexCoord1 : TEXCOORD1;
};

vert2frag main(app2vert IN,
              uniform float4x4 ModelViewProj,
              uniform float4x4 ModelView,
              uniform float4x4 ModelViewIT,
              uniform float4x4 TextureMat,
              uniform float4 Constants)
{ 
    vert2frag OUT;

    OUT.HPosition = mul(ModelViewProj, IN.Position);

    // convert the position and normal
    // into appropriate spaces
    float3 eyeToVert = mul(ModelView, IN.Position).xyz;
    eyeToVert = normalize(eyeToVert);
    float3 normal = mul(ModelViewIT, IN.Normal).xyz;
    normal = normalize(normal);

    // do the actual refraction
    // T = aI+bN T = refracted ray
    // a = theta theta = ratio of indexes of refraction
    // b = theta * cos(Ai) - sqrt(1+theta^2*(cos^2(Ai)-1))
    // Ai = angle of incidence
    // cos(Ai) = eyeToVert.normal

    float theta = Constants.x;
    float eyeDotNormal = dot(normal, eyeToVert);
    float k = eyeDotNormal * eyeDotNormal - 1;
    k = 1 + theta * theta * k;
    k = clamp(k, 0, 1);
float a = theta;
float b = theta * eyeDotNormal + sqrt(k);
float3 result = a*eyeToVert + b*normal;

OUT.TexCoord0.xyz = result;
OUT.TexCoord0.w = 1;

// calculate the Fresnel reflection
OUT.TexCoord1.xyz = reflect(eyeToVert, normal);
OUT.TexCoord1.w = 1;

float f0 = .1;
// compute the Fresnel term
float oneMCosAngle = 1+dot(eyeToVert,normal);
oneMCosAngle = pow(oneMCosAngle, 5);
OUT.Color0 = lerp(oneMCosAngle, 1, f0).xxxx;

return OUT;
}
Shadow Mapping

This effect shows generating texture coordinates for shadow mapping, along with using the shadow map in the lighting equation per pixel (Figure 16).

Figure 16. Example of Shadow Mapping
(See Appendix D for a color version of this illustration)
Vertex Shader Cg Source Code

```plaintext
struct appdata {
    float3 Position : POSITION;
    float3 Normal : NORMAL;
};

struct vpconn {
    float4 Hposition : POSITION;
    float4 TexCoord0 : TEXCOORD0;
    float4 TexCoord1 : TEXCOORD1;
    float4 Color0 : COLOR0;
};

tvpconn main(appdata IN,
    uniform float4x4 WorldViewProj,
    uniform float4x4 TexTransform,
    uniform float3x3 WorldIT,
    uniform float3 LightVec)
{
    vpconn OUT;

    float3 worldNormal = normalize(mul(WorldIT, IN.Normal));
    float ldotn = max(dot(LightVec, worldNormal), 0.0);
    OUT.Color0.xyz = ldotn.xxx;

    float4 tempPos;
    tempPos.xyz = IN.Position.xyz;
    tempPos.w = 1.0;

    OUT.TexCoord0 = mul(TexTransform, tempPos);
    OUT.TexCoord1 = mul(TexTransform, tempPos);

    OUT.HPosition = mul(WorldViewProj, tempPos);

    return OUT;
}
```
Pixel Shader Cg Source Code

```cg
struct v2f_simple {
    float4 Hposition : POSITION;
    float4 TexCoord0 : TEXCOORD0;
    float4 TexCoord1 : TEXCOORD1;
    float4 Color0 : COLOR0;
};

struct myfragout {
    float4 COL : COLOR;
};

myfragout main(v2f_simple IN,
               uniform sampler2D ShadowMap,
               uniform sampler2D SpotLight)
{
    myfragout OUT;

    float4 shadow    = tex2D(ShadowMap);
    float4 spotlight = tex2D(SpotLight);
    float4 lighting  = IN.Color0;

    OUT.COL = shadow * spotlight * lighting;

    return OUT;
}
```
Shadow Volume Extrusion

Description

This effect uses vertex programs to generate shadow volumes by extruding geometry along the light vector (Figure 17).

Figure 17. Example of Shadow Volume Extrusion
(See Appendix D for a color version of this illustration)
// Define register bindings
#define CV_WORLDVIEWPROJ_0 2
#define CV_WORLDVIEWPROJ_1 3
#define CV_WORLDVIEWPROJ_2 4
#define CV_WORLDVIEWPROJ_3 5
#define CV_SHDVOL_DIST 6
#define CV_FACTORS 7
#define CV_CONSTS_1 8
#define CV_LIGHT_COLOR 19
#define CV_LIGHT_POS_OSPACE 21
#define CV_EYE_POS_OSPACE 22
#define CV_LIGHT_CONST 24
#define CV_FATNESS_SCALE 27
#define CV_COLOR 28
#define CV_COLORSWITCH 29

struct appdata
{
  float4 Position : POSITION;
  float3 Normal : NORMAL;
  float4 DiffuseColor : COLOR0;
  float2 TexCoord0 : TEXCOORD0;
};

struct vpconn
{
  float4 Hposition : POSITION;
  float4 Color0 : COLOR0;
  float2 TexCoord0 : TEXCOORD0;
};

vpconn main( appdata IN,
uniform float4x4 WorldViewProj : C2,
uniform float4 Consts_0512 : C8, // { 0, 0.5, 1, 2 }
uniform float4 LightPos : C21, // (in object space)
uniform float4 Fatness : C27,
uniform float4 ShadowExtrudeDist : C6,
uniform float4 Factors : C7
)
{
  vpconn OUT;
  // Create normalized vector from vertex to light
  float4 light_to_vert = normalize( IN.Position - LightPos );
// N dot L to decide if point should be moved away
// from the light to extrude the volume
// .xyz will use DP3 dot-product
// dot().xxxx smears DP3 result to
// all fields of the output
float4 r4 = dot( -light_to_vert.xyz, IN.Normal.xyz ).xxxx;

// Inset the position along
// the normal vector direction
// This moves the shadow volume points
// inside the model slightly to minimize
// popping of shadowed areas as
// each facet comes in and out of shadow.
// The Fatness value should be negative
float4 inset_pos = (IN.Normal * Fatness.xyz +
                        IN.Position.xyz ).xyzz;
inset_pos.w = IN.Position.w;

// scale the vector from light to vertex
float4 extrusion_vec = light_to_vert * ShadowExtrudeDist;

// if r4 < 0 then the vertex faces
// away from the light, so move it.
// It will be moved along the direction from
// light to vertex to extrude the shadow volume.
// Consts_0512 = { 0.0f, 0.5f, 1.0f, 2.0f };
// So this does r5 = N dot L < 0 ? 1.0 : 0.0
float r5 = (float) ( r4.x < Consts_0512.x );

// Move the back-facing shadow volume points
float4 new_position = extrusion_vec * r5 + inset_pos;

// Transform position to hclip space;
OUT.HPosition = mul( WorldViewProj, new_position );

// Set the color to blue for when the shadow volume
// is rendered in color for illustrative purposes
float4 color = Consts_0512.xxxx;
color.z = Factors.x;

OUT.Color0  = color;
OUT.TexCoord0.xy = IN.TexCoord0;

return( OUT );
Sine Wave Demo

This effect modifies the vertex positions using a sine function based on the current time. It demonstrates use of the built-in \texttt{sin()} function. It also computes a normal based on the perturbed mesh, and uses this to compute a reflection vector to look up in a cube map (Figure 18).

![Example of Sine Wave](image)

Figure 18. Example of Sine Wave
(See Appendix D for a color version of this illustration)
Vertex Shader Cg Source Code

```cg
struct appdata {
    float4 TexCoord0 : TEXCOORD0;
};

struct vpconn {
    float4 HPOS : POSITION;
    float4 COL0 : COLOR0;
    float4 TEX0 : TEXCOORD0;
};

vpconn main(appdata IN,
    uniform float4x4 WorldViewProj,
    uniform float3x4 WorldView,
    uniform float3x3 WorldViewIT,
    uniform float4     WavesX,
    uniform float4     WavesY,
    uniform float4     WavesH,
    uniform float4     Time
) {
    vpconn OUT;

    float4 angle;

    angle = WavesX.xyzw * IN.TexCoord0.x +
            WavesY.xyzw * IN.TexCoord0.y;
    angle = angle + Time;

    float4 sine, cosine;

    sincos(angle.x, sine.x, cosine.x);
    sincos(angle.y, sine.y, cosine.y);
    sincos(angle.z, sine.z, cosine.z);

    // position is: (u, sum(hi * sin(anglei)), v, 1)
    float4 position;
    position.xz = IN.TexCoord0.xy;
    position.w  = 1.0f;
    position.y  = dot(WavesH, sine);

    OUT.HPOS = mul(WorldViewProj, position);
}
```
// normal is (t h WaveX cos(angle),
// -1,
// t h WaveY cos(angle))
float3 normal;
normal.x = dot(WavesH * WavesX, cosine);
normal.y = -1.0f;
normal.z = dot(WavesH * WavesY, cosine);

// transform normal into eye-space
normal = mul(WorldViewIT, normal);
normal = normalize(normal);

// Transform vertex to eye-space and
// compute the vector from the eye to the vertex.
// Because the eye is at 0, no subtraction is
// necessary. Because the reflection of this vector
// looks into a cube-map normalization is also
// unnecessary!
float3 eyeVector = mul(WorldView, position);
OUT.TEX0.xyz = reflect(eyeVector, normal);

return OUT;
Matrix Palette Skinning

Description

This effect performs matrix palette skinning using two bones per vertex. All the bones for the mesh are set in the constant memory, and each vertex includes two indices that indicate which bones influence this vertex. The final skinned positions are computed using these bones, along with the weights supplied per vertex. Tangent-space bases are skinned in a similar fashion and then used to transform the light vector into tangent space for per-pixel bump mapping (Figure 19).

Figure 19. Example of Matrix Palette Skinning
(See Appendix D for a color version of this illustration)
Vertex Shader Cg Source Code

```c
struct appdata {
    float3 Position : POSITION;
    float2  Weights : BLENDWEIGHT0;
    float2 Indices : BLENDINDICES;
    float3 Normal : NORMAL;
    float2 TexCoord0 : TEXCOORD0;
    float3 S : TEXCOORD1;
    float3 T : TEXCOORD2;
    float3 SxT : TEXCOORD3;
};

struct vpconn {
    float4 Hposition : POSITION;
    float4 TexCoord0 : TEXCOORD0;
    float4 TexCoord1 : TEXCOORD1;
    float4 Color0 : COLOR0;
};

vpconn main(appdata IN,
    uniform float4x4 WorldViewProj,
    uniform float3x4 Bones[26],
    uniform float3 LightVec
)
{
    vpconn OUT;

    float4 tempPos;
    tempPos.xyz = IN.Position.xyz;
    tempPos.w = 1.0;

    // grab first bone matrix
    float i = IN.Indices.x;

    //transform position
    float3 pos0 = mul(Bones[i], tempPos);

    //create 3x3 version of bone matrix
    float3x3 m;
    m.m_00_01_02 = Bones[i].m_00_01_02;
    m.m_10_11_12 = Bones[i].m_10_11_12;
    m.m_20_21_22 = Bones[i].m_20_21_22;
}
```
// transform S, T, SxT
float3 s0 = mul(m, IN.S);
float3 t0 = mul(m, IN.T);
float3 sxt0 = mul(m, IN.SxT);

// next bone
i = IN.Indices.y;

// create 3x3 version of bone
m.m_00_01_02 = Bones[i].m_00_01_02;
m.m_10_11_12 = Bones[i].m_10_11_12;
m.m_20_21_22 = Bones[i].m_20_21_22;

float3 pos1 = mul(Bones[i], tempPos);

// transform S, T, SxT
float3 s1 = mul(m, IN.S);
float3 t1 = mul(m, IN.T);
float3 sxt1 = mul(m, IN.SxT);

// final blending

// blend s, t, sxt
float3 finalS = s0 * IN.Weights.x + s1 * IN.Weights.y;
float3 finalT = t0 * IN.Weights.x + t1 * IN.Weights.y;
float3 finalSxT = sxt0 * IN.Weights.x + sxt1 * IN.Weights.y;

// blend between the two positions
float3 finalPos = pos0 * IN.Weights.x + pos1 * IN.Weights.y;

float3x3 worldToTangentSpace;
worldToTangentSpace.m_00_01_02 = finalS;
worldToTangentSpace.m_10_11_12 = finalT;
worldToTangentSpace.m_20_21_22 = finalSxT;

float3 tangentLight = normalize(mul(worldToTangentSpace, LightVec));

// scale and bias, add bit of ambient
tangentLight = ((tangentLight + 1.0) * 0.5) + 0.2;

// create float4 with 1.0 alpha
float4 tempLight;
tempLight.xyz = tangentLight.xyz;
tempLight.w = 1.0;
OUT.Color0 = tempLight;
// pass through texcoords
OUT.TexCoord0.xy = IN.TexCoord0.xy;
OUT.TexCoord1.xy = IN.TexCoord0.xy;

float4 tempPos2;
tempPos2.xyz = finalPos.xyz;
tempPos2.w = 1.0;

OUT.HPosition = mul(WorldViewProj, tempPos2);
return OUT;
Appendix A.
Cg Language Specification

Language Overview

The Cg language is primarily modeled on ANSI C, but adopts some ideas from modern languages such as C++ and Java, and from earlier shading languages such as RenderMan and the Stanford shading language. The language also introduces a few new ideas. In particular, it includes features designed to represent data flow in stream-processing architectures such as GPUs. Profiles (specified at compile time) may subset certain features of the language, for example, the ability to implement loops, or the precision at which certain computations are performed.

Silent Incompatibilities

Most of the changes from ANSI C are either omissions or additions, but there are a few potentially silent incompatibilities. These are changes within Cg that could cause a program that compiles without errors to behave in a manner different from C:

- The type promotion rules for constants are different when the constant is not explicitly typed using a type cast or type suffix. In general, a binary operation between a constant that is not explicitly typed and a variable is performed at the variable's precision, rather than at the constant's default precision.
- Declarations of `struct` perform an automatic `typedef` (as in C++) and thus could override a previously declared type.
- Arrays are first-class types that are distinct from pointers. As a result, array assignments semantically perform a copy operation for the entire array.

Similar Operations That Must be Expressed Differently

There are several changes that force the same operation to be expressed differently in Cg than in C:

- A Boolean type (`bool`) is introduced, with corresponding implications for operators and control constructs.
- Arrays are first-class types, because Cg does not support pointers.
- Functions pass values by value/result, and thus use an `out` or `inout` modifier in the formal parameter list to return a parameter. By default, formal parameters are `in`, but it is acceptable to specify this explicitly. Parameters can also be specified as `in out`, which is semantically the same as `inout`. 
Differences from ANSI C

The Cg language was developed based on the ANSI-C language with the following major additions, deletions, and changes. (This is a summary—more detail is provided later in this document):

- Language profiles (described in the Profiles section on page 117) may subset language capabilities in a variety of ways. In particular, language profiles may restrict the use of `for` and `while` loops. For example, some profiles may only support loops that can be fully unrolled at compile time.

- A binding semantic may be associated with a structure tag, with a variable, or with a structure element, to denote that object's mapping to a specific hardware or API resource. Binding semantics are described in the Binding Semantics section.

- `goto`, `break`, and `continue` are not supported, but are reserved keywords.

- `switch`, `case`, and `default` are not supported, but are reserved keywords. Labels are not supported either.

- No pointers or pointer-related capabilities (such as the `&` and `->` operators)

- Arrays are supported, but with some limitations on size and dimensionality. Restrictions on the use of computed subscripts are also permitted. Arrays may be designated as `packed`. The operations allowed on packed arrays may be different from those allowed on unpacked arrays. Predefined `packed` types are provided for vectors and matrices. It is strongly recommended these predefined types be used.

- There is a built-in swizzle operator: `.xyzw` or `.rgba` for vectors. This operator allows the components of a vector to be rearranged and/or replicated. It also allows the creation of a vector from a scalar.

- For an lvalue, the swizzle operator allows components of a vector or matrix to be selectively written.

- There is a similar built-in swizzle operator for matrices:
  
  `.m<row><col>[_m<row><col>][...]]`

  This operator allows access to individual matrix components, and allows the creation of a vector from elements of a matrix. For compatibility with DirectX 8 notation, there is a second form of matrix swizzle, described later.

- Numeric data types are different, and implementations are given some freedom to support different data types.

- Many operators support per-element vector operations.

- The `?:`, `||`, `&&`, `!`, and comparison operators can be used with `bool` four-vectors to perform four conditional operations simultaneously.

- Non-static global variables and parameters to top-level functions (such as `main()`), may be designated as `uniform`. A `uniform` variable may be read and written within a program, just like any other variable. However, the `uniform` modifier indicates that the initial value of the variable or parameter is expected to be constant across a large number of invocations of the program.

- A new set of `sampler*` types represents handles to texture objects.

- No `int` type or `int` operators, although some of C's `int` operators have become `bool` operators.

- Function overloading is supported.
No `enum` or `union`.

No bit-field declarations in structures.

No structure initializers.

Variables may be defined anywhere before they are used, rather than just at the beginning of a scope as in C. (That is, we adopt the C++ rules that govern where variable declarations are allowed.)

Vector constructors (such as the form `float4(1,2,3,4)` ) may be used anywhere in an expression.

A `struct` definition automatically performs a corresponding `typedef` (just like C++).

C++-style `//` comments are allowed in addition to C-style `/* ... */` comments.

---

Detailed Language Specification

Definitions

The following definitions are based on the ANSI C standard:

- **Object**: An object is a region of data storage in the execution environment, the contents of which can represent values. When referenced, an object may be interpreted as having a particular type.

- **Declaration**: A declaration specifies the interpretation and attributes of a set of identifiers.

- **Definition**: A declaration that also causes storage to be reserved for an object or code that will be generated for a function named by an identifier is a definition.

Profiles

Compilation of a Cg program always occurs in the context of a compilation profile. The profile specifies whether certain optional language features are supported. These optional language features include certain data types, control structures, and Standard Library functions. The choice of a compilation profile is made externally to the language (for example, via a compiler command-line switch).
As an example, the Cg compiler has the following profiles:

- **fp30** for compiling fragment programs to `NV_fragment_program`
- **vp30** for compiling vertex programs to `NV_vertex_program2`
- **dx8vs** for compiling fragment programs to DirectX 8.1 vertex shaders
- **dx8ps** for compiling fragment programs to DirectX 8.1 pixel shaders
- **arbvp1** for compiling fragment programs to `ARB_vertex_program`
- **vp20** for compiling vertex programs to `NV_vertex_program`

Other profiles may be supported in the future.

Each profile must have a separate specification that describes its characteristics and limitations.

### The Uniform Modifier

Non-static global variables and parameters passed to functions (such as `main()`) can be declared with an optional qualifier `uniform`. To specify a `uniform` variable, use the syntax:

```plaintext
uniform <type> <variable>
```

For example,

```plaintext
uniform float4 myVector;
fragout foo(uniform float4 uv);
```

If the `uniform` qualifier is specified for a non-top-level function, it is meaningless and is ignored. The intent of this rule is to allow a function to serve either as a top-level function or as a non-top-level function.

Note that uniform variables may be read and written just like non-uniform variables. The uniform qualifier simply provides information about how the initial value of the variable will be specified and stored, through a mechanism external to the language.

Variables and/or parameters may be implicitly designated as uniform through the binding semantic mechanism, as discussed in the Binding Semantics section.

Typically, the initial value of a uniform variable/parameter is stored in a different class of hardware register. Furthermore, the external mechanism for specifying the initial value of uniform variables/parameters may be different than that used for specifying the initial value of non-uniform variables/parameters. Uniform parameters are normally treated as persistent state, while non-uniform parameters are treated as streaming data, with a new value specified for each stream record (such as within a vertex array).
Function Declarations

Functions are declared essentially as in C. A function that does not return a value must be declared with a `void` return type. A function that takes no parameters may be declared in one of two ways:

- As in C, using the `void` keyword: `functionName(void)`
- With no parameters at all: `functionName()`

Functions may be declared as `static`. If so, they may not be compiled as a program, and are not visible from other compilation units.

Overloading of Functions by Profile

Cg supports overloading of functions by compilation profile. This capability allows a function to be implemented differently for different profiles. It is also useful because different profiles may support different subsets of the language capabilities, and because the most efficient implementation of a function may be different for different profiles.

Only static functions may be overloaded by profile, and the profile name must follow the `static` keyword. For example, to define two different versions of the function `foobar()` for the `dx8vs` and `dx8ps` profiles:

```
static dx8vs float foobar(float x) {...};
static dx8ps float foobar(float x) {...};
```

If a function definition does not include a profile, then the function is referred to as an open-profile function. Open-profile functions apply to all profiles.

It is not legal to declare overloaded functions that are distinguished only by one function being open-profile, and the other function having a specific profile. That is, an open profile function precludes the definition of a corresponding profile-specific function.

Function Calls

A function call returns an rvalue. Therefore, if a function returns an array, the array may be read but not written. For example, the following is allowed:

```
y = myfunc(x)[2];
```

But not this:

```
myfunc(x)[2] = y;
```

For multiple function calls within an expression, the calls can occur in any order— it is undefined.
Types

- The `int` type is 32-bit two’s-complement. This type is included in the specification for completeness. It is not currently possible to declare objects of type `int`.
- The `float` type is as close as possible to the IEEE single precision (32-bit) floating point. All fully conformant profiles must support the `float` data type.
- The `half` type is lower-precision IEEE-like floating point. Profiles may omit support for the `half` data type.
- The `fixed` type is a signed, two’s-complement fixed-point type of range at least \([-1,1]\), with at least 8 bits of fractional precision. Overflow operations on the data type clamp rather than wrap. Profiles may omit support for the `fixed` data type.
- The `bool` type is Boolean. Objects of `bool` type are either true or false.
- The `cint` type is 32-bit two’s-complement. This type is meaningful only at compile time; it is not possible to declare objects of type `cint`.
- The `cfloat` type is an IEEE single-precision (32-bit) floating point. This type is meaningful only at compile time; it is not possible to declare objects of type `cfloat`.
- The `void` type may not be used in any expression. It may only be used as the return type of functions that do not return a value.
- The `sampler*` types are handles to texture objects. Formal parameters of a program or function may be of type `sampler*`. No other definition of `sampler*` variables is permitted. A `sampler*` variable may only be used by passing it to another function as an `in` parameter. Assignment to `sampler*` variables is not permitted, and `sampler*` expressions are not permitted. In a profile that supports texture mapping, the following `sampler*` types must be provided: `sampler1D`, `sampler2D`, `sampler3D`, and `samplerCUBE`. Profiles are allowed to define additional `sampler*` types.
- An array type is a collection of one or more elements of the same type. An array variable has a single index.
- Some array types may be optionally designated as packed, using the `packed` type modifier. The storage format of a `packed` type may be different from the storage format of the corresponding `unpacked` type. The storage format of packed types is implementation dependent, but must be consistent for any particular compiler/profile combination. The operations supported on a `packed` type in a particular profile may be different than the operations supported on the corresponding `unpacked` type in that same profile.
- When declaring an array of arrays in a single declaration, the `packed` modifier only refers to the outermost array. However, it is possible to declare a packed array of packed arrays by declaring the first level of array in a `typedef` using the `packed` keyword, and then declaring a packed array of this type in a second statement. It is not possible to have a packed array of unpacked arrays.
For any supported numeric data type TYPE, implementations must support the following packed array types, which we call vector types. Implementations must also predefine type identifiers (in the global scope) to represent these types:

```c
packed TYPE TYPE1[1]; // TYPE1 == TYPE[1];
packed TYPE TYPE2[2]; // TYPE2 == TYPE[2];
packed TYPE TYPE3[3]; // TYPE3 == TYPE[3];
packed TYPE TYPE4[4]; // TYPE4 == TYPE[4];
```

For example, fully conformant implementations must have predefined type identifiers `float1`, `float2`, `float3`, and `float4`.

For any supported numeric data type TYPE, implementations must support the following packed array types, which we call matrix types. Implementations must also predefine type identifiers (in the global scope) to represent these types:

```c
packed TYPE1 TYPE1x1[1];
packed TYPE2 TYPE1x2[1];
packed TYPE3 TYPE1x3[1];
packed TYPE4 TYPE1x4[1];
packed TYPE1 TYPE2x1[2];
packed TYPE2 TYPE2x2[2];
packed TYPE3 TYPE2x3[2];
packed TYPE4 TYPE2x4[2];
packed TYPE1 TYPE3x1[3];
packed TYPE2 TYPE3x2[3];
packed TYPE3 TYPE3x3[3];
packed TYPE4 TYPE3x4[3];
packed TYPE1 TYPE4x1[4];
packed TYPE2 TYPE4x2[4];
packed TYPE3 TYPE4x3[4];
packed TYPE4 TYPE4x4[4];
```

For example, fully conformant implementations must have predefined type identifiers `float2x1`, `float3x3`, `float4x4`, etc. Note that the typedefs follow the usual matrix-naming convention of `TYPE_rows_X_columns`. Thus, we also have

```c
a[3] = a._30_31_32_33; // Extract 3rd row
```

Implementations are required to support indexing of vectors and matrices with constant indices. Likewise, implementations must allow matrix types to be swizzled.

The multi-dimensional array `float M[4][4]` is not type-equivalent to the matrix `float4x4 M`.

**Note:** The storage format of packed arrays is implementation dependent, so row access may be efficient for some matrix types, while column access is efficient for others. The intent is to allow implementation flexibility, particularly for 3x4, 4x3, and 2x2 matrices.

A **struct** type is a collection of one or more members of possibly different types.
Type Support

Some implementations may be unable to support some data types, especially in fragment-program profiles. If so, sub-setting of the types will be necessary, and additional types may be introduced. However, implementations must always support the \texttt{cfloat} and \texttt{cint} types. Fully conformant implementations must also support the \texttt{float} type. If \texttt{float} is not supported, implementations may have to modify the promotion rules for \texttt{cfloat} and \texttt{cint}, especially in function overloading.

Type Categories

- The integral type category includes type \texttt{cint} and \texttt{int} (\texttt{int} is not currently supported).
- The floating type category includes types \texttt{cfloat}, \texttt{float}, \texttt{half}, \texttt{fixed}, and \texttt{clamp1u}. (Note that floating really means floating or fixed/fractional.)
- The numeric type category includes integral and floating types.
- The compile-time type category includes types \texttt{cfloat} and \texttt{cint}. These types are used by the compiler for constant type conversions.
- The concrete type category includes all types that are not included in the compile-time type category.
- The scalar type category includes all types in type category numeric, \texttt{bool} type, and all types in type category compile-time.

In this specification, a reference to a <category> type (such as a reference to a numeric type) means one of the types included in the category (such as \texttt{float}, \texttt{half}, \texttt{fixed}, and \texttt{clamp1u}).

Constants

A constant may be explicitly typed or implicitly typed. Explicit typing of a constant is performed as in C, by suffixing the constant with a single character indicating the type of the constant:

- \texttt{f = float}
- \texttt{h = half}
- \texttt{x = fixed}

Additional suffixes may be defined by particular profiles.

Any constant that is not explicitly typed is implicitly typed. If the constant includes a decimal point, then it is implicitly typed as \texttt{cfloat}. If it does not include a decimal point, it is implicitly typed as \texttt{cint}.

By default, constants are base 10. Hexadecimal constants may be specified by prefixing the constant with \texttt{0x}, as in C. Octal constants are not supported; constants with a leading zero are base 10.
Type Qualifiers

The type of an object may be qualified with one or more qualifiers. Qualifiers apply only to objects. Qualifiers are removed from the value of an object when used in an expression. The qualifiers are

- **const**
  The value of a `const` qualified object cannot be changed after its initial assignment. The definition of a `const` qualified object that is not a parameter must contain an initializer. Named compile time values are inherently qualified as `const` but an explicit qualification is allowed as well.

- **in** and **out**
  Formal parameters may be qualified as `in`, `out`, or both. By default, formal parameters are `in` qualified. An `in` qualified parameter is equivalent to a call by value parameter. An `out` qualified parameter is equivalent to a call by result parameter, and an `inout` qualified parameter is equivalent to a value result parameter. `out` qualified parameters cannot be `const` qualified.

Type Casts

An expression of one type can be explicitly converted to another type with a type cast, as in C. These casts are

- Compile-time type: applied to expressions of compile-time type.
- Numeric type: applied to expressions of numeric or compile-time types.
- Numeric vector type: applied to another vector type of the same number of elements.
- Numeric matrix type: applied to another matrix type of the same number of rows and columns.

Several additional casts are allowed that are not so cleanly analogous to those provided by C. These casts must also be specified explicitly:

- A **cast** can convert a vector to a shorter vector by keeping the lower-numbered elements.
- A **cast** can convert a matrix to a smaller matrix by keeping the lower-numbered rows and columns (the upper-left corner of the matrix).
- A **bool** may be explicitly cast to a numeric scalar type. For example,

  ```
  bool   b;
  float  f;
  b = false;
  f = (float)b;
  ```

- A numeric scalar may be explicitly cast to a **bool** type. The cast `b = (bool)f` is defined as:

  ```
  bool   b;
  float  f;
  b = ((f == 0) ? false : true);
  ```
No other casts are allowed. In particular, a cast cannot convert a vector to a scalar or to a matrix. Also, a cast cannot convert a scalar to a vector or to a matrix.

**Type Equivalency**

Type T1 is equivalent to type T2 if

- T2 is equivalent to T1, or
- T1 and T2 are the same scalar, vector, or structure type.
- T1 is a typedef name of T2, or
- T1 and T2 are arrays of equivalent types with the same number of elements, or
- The unqualified types of T1 and T2 are equivalent and both types have the same qualifications, or
- T1 and T2 are functions with equivalent return types, the same number of parameters, and all corresponding parameters are pair-wise equivalent.

A vector type (such as `float4`) is not equivalent to the correspondingly sized array type (such as `float[4]`). The same applies to packed matrix types.

**Type-Promotion Rules**

`cfloat` and `cint` types behave like `float` and `int`, except for usual arithmetic conversion behavior (defined below) and function-overloading rules (defined later).

We define usual arithmetic conversions for binary operators. Profiles that support data types other than `float`, `cfloat`, and `cint` are permitted to specify an alternate set of conversions.

- If either operand is `float`, the other is converted to `float`.
- Otherwise, if either operand is `cfloat`, the other operand is converted to `cfloat`.
- Otherwise, both operands have type `cint`.

Note that conversions happen prior to performing the operation.

**Assignment**

Assignment of an expression to an object or compile-time typed value casts the expression to the type of the object or value. The resulting value is then assigned to the object or value.
Smearing of Scalars to Vectors

If a binary operator is applied to a vector and a scalar, the scalar is automatically type-promoted to a same-sized vector by replicating the scalar into each component. The ternary ?: operator also supports smearing. The binary rule is applied to the second and third operands first, and then the binary rule is applied to this result and the first operand.

Namespaces

Just as in C, there are two namespaces. Each has multiple scopes, as in C.

- Tag namespace
  - struct tags
- Regular namespace
  - typedefs (including automatic typedefs from struct declarations)
  - variables
  - function names

Arrays and Subscripting

Arrays are declared as in C, except that they may optionally be declared to be packed, as described earlier. Arrays in Cg are first-class types, so array parameters to functions and programs must be declared using array syntax, rather than pointer syntax. Likewise, assignment of an array-typed object implies an array copy, rather than a pointer copy.

Arrays with size [1] may be declared, but are considered a different type from the corresponding non-array type.

Because the language does not currently support pointers, the storage order of arrays is only visible when an application passes parameters to a vertex or fragment program. Therefore, the compiler is currently free to allocate temporary variables as it sees fit.

The declaration and use of arrays of arrays is in the same style as in C. That is, if the 2D array $A$ is declared as

```c
float A[4][4];
```

The following statements are true:

- The array is indexed as $A[\text{row}][\text{column}]$;
- The array can be built with a constructor using
  ```c
  A = {{A[0][0], A[0][1], A[0][2], A[0][3]},
       {A[1][0], A[1][1], A[1][2], A[1][3]},
       {A[3][0], A[3][1], A[3][2], A[3][3]});
  ```
  then, $A[0]$ would be equivalent to $(A[0][0], A[0][1], A[0][2], A[0][3])$

Support must be provided for structs containing arrays.
Minimum Array Requirements for Fully Conformant Profiles

Fully conformant profiles are required to provide partial support for certain kinds of arrays. This partial support is designed to support vectors and matrices, as well as the passing of arrays of light state (indexed by light number) to vertex programs, and the passing of arrays of skinning matrices to vertex programs.

Fully conformant profiles must support subscripting, copying, and swizzling of vectors and matrices. However, subscripting with run-time computed indices is not required to be supported.

Fully conformant vertex profiles must support the following operations for any non-packed array that is a uniform parameter to the program, or is an element of a structure that is a uniform parameter to the program. This requirement also applies when the array is indirectly a uniform program parameter (that is, it and/or the structure containing it has been passed via a chain of in function parameters). The two operations that must be supported are:

- rvalue subscripting by a run-time computed value or a compile-time value.
- Passing the entire array as a parameter to a function, where the corresponding formal function parameter is declared as in.

Note that the following operations are explicitly not required to be supported:

- lvalue-subscripting
- copying
- other operators, including multiply, add, compare, etc.

Note that when the array is rvalue subscripted, the result is an expression, and this expression is no longer considered to be a uniform program parameter. Therefore, if this expression is an array, its subsequent use must conform to the standard rules for array usage.

These rules are not limited to arrays of numeric types, and thus imply support for arrays of structs, arrays of matrices, and arrays of vectors when the array is a uniform program parameter. Maximum array sizes may be limited by the number of available registers or other resource limits, and compilers are permitted to issue error messages in these cases. However, implementations must support sizes of at least float arr[8], float4 arr[8], and float4x4 arr[4][4].

Fragment profiles are not required to support any operations on arbitrarily sized arrays; only support for vectors and matrices is required.
Function Overloading
Multiple functions may be defined with the same name, as long as the definitions can be distinguished by unqualified parameter types, and do not have an open-profile conflict (as described in the section on open functions).

Function-matching rules:
- Add all visible functions in the calling scope to the set of function candidates.
- Eliminate functions whose profile conflicts with current compilation profile.
- Eliminate functions with the wrong number of formal parameters.
- If the set is empty, fail.
- For each actual parameter expression in sequence, perform the following:
  - If the type of the actual parameter matches the unqualified type of the corresponding formal parameter in any function in the set, then remove all functions whose corresponding parameter do not match exactly.
  - If there is a defined promotion for the type of the actual parameter to the unqualified type of the formal parameter of any function, then remove all functions for which this is not true from the set.
  - If there is a valid implicit cast that will convert the type of the actual parameter to the unqualified type of the formal parameter of any function, then remove all functions without this cast.
  - Fail.
- If there is exactly one function in the set after all parameters have been processed, and there are no formal parameters left to consider, accept.
- Otherwise fail.

Global Variables
Global variables are declared and used as in C. Uniform non-static variables may have a semantic associated with them. Uniform non-static variables may have their value set through the run-time API.

Use of Uninitialized Variables
It is incorrect for a program to use an uninitialized variable. However, the compiler is not obligated to detect such errors, even if it would be possible to do so by compile-time data-flow analysis. The value obtained from reading an uninitialized variable is undefined. This same rule applies to the implicit use of a variable that occurs when it is returned by a top-level function. In particular, if a top-level function returns a `struct`, and some element of that `struct` is never written, then the value of that element is undefined.
Preprocessor

Cg implementations must support C-preprocessor capabilities: \#if, \#define, etc. However, Cg implementations are not required to support macro-like \#define, or the use of \#include statements.

Overview of Binding Semantics and Connectors

In stream-processing architectures, data packets flow between different programmable units. For example, on a GPU, packets of vertex data flow from the application to the vertex program.

Because packets are produced by one program (such as the application), and consumed by another (such as the vertex program), there must be some method for defining the interface between the two programs. Cg allows the user to choose between two different approaches to define these interfaces.

The first approach is to associate a binding semantic with each element of the packet. This approach is a bind-by-name approach. For example, an output with the binding semantic \texttt{FOO} is fed to an input with the binding semantic \texttt{FOO}. Profiles may allow the user to define arbitrary identifiers in this "semantic namespace", or they may restrict the allowed identifiers to a predefined set. Often, these predefined names correspond to the names of hardware registers or API resources.

In some cases, predefined names may control non-programmable parts of the hardware. For example, vertex programs normally compute a position that is fed to the rasterizer, and this position is stored in an output with the binding semantic \texttt{POSITION}.

The second approach to defining data packets is to describe all data present in a packet, and let the compiler decide how to store it. In Cg, the user can describe the contents of a data packet by placing all of its contents into a \texttt{struct}. When a \texttt{struct} is used in this manner, we refer to it as a connector. The two approaches are not mutually exclusive, as we will describe in more detail later. The connector approach allows the user to rely on a combination of user-specified semantic bindings and compiler-determined bindings.

Binding Semantics

A binding semantic may be associated with an input to a top-level function in one of three ways:

- The binding semantic is specified in the formal parameter declaration for the function. The syntax for formal parameters to a function is

  \[
  \textbf{[const]} \textbf{[in | out | inout]} \\
  \textbf{<type>} \textbf{<identifier>} \textbf{=} \textbf{<initializer>} \textbf{[ : <binding-semantic>]} \\
  \]

- If the formal parameter is a \texttt{struct}, the binding semantic may be specified with an element of the \texttt{struct} when the \texttt{struct} is defined:

  \[
  \text{struct} \textbf{<struct-tag>} \{ \\
  \textbf{<type>} \textbf{<identifier>}[ \textbf{: <binding-semantic>}]; \\
  \ldots \\
  \};
  \]
If the input to the function is implicit (a non-static global variable that is read by the function), the binding semantic may be specified when the non-static global variable is declared:

```
<type> <identifier> [= <initializer>] [: <binding-semantic>];
```

If the non-static global variable is a `struct`, the binding semantic may be specified when the `struct` is defined, as described in the second way above.

Non-uniform binding semantics may not be applied to a global variable. (Non-static global variables may only be implicitly or explicitly uniform).

A binding semantic may be associated with the output of a top-level function in a similar manner. If the output is the return value of the function, then the only method available for specifying a semantic is to return a `struct`, and to specify the binding semantic(s) with elements of the `struct` when the `struct` is defined. If the output is a formal parameter, then the binding semantic may be specified using the same approach used to specify binding semantics for inputs.

**Aliasing of Semantics**

Semantics must honor copy-on-input and copy-on-output model. Thus, if the same input binding semantic is used for two different variables, those variables will be initialized with the same value, but the variables will not be aliased thereafter. Output aliasing is illegal, but implementations are not required to detect it. If the compiler does not issue an error on a program that aliases output binding semantics, the results are undefined.

**Restrictions on Semantics Within Structs**

For a particular profile, it is illegal to mix input binding semantics and output binding semantics within a particular `struct`. That is, for a particular top-level function, a `struct` must be either input-only or output-only. Likewise, a `struct` must consist exclusively of uniform inputs or exclusively of non-uniform inputs. It is illegal to use binding semantics to mix the two within a single `struct`.

**Detailed Rules on Semantics**

The following rules are somewhat redundant, but provide extra clarity:

- All input varying semantics must belong to parameters to the main function. These parameters can be of type `scalar, vector, matrix, or struct` with members of these types.
- All varying output semantics are attached to the return value of the main function or to `out` parameters to the main function.
- All other parameters to the main function have uniform semantics, either explicitly declared on implicitly inferred.
- Non-static global variables have either implicit or explicit uniform semantics.
- Semantics attached to parameters to non-main functions are ignored.
Structure tags are allowed to have semantics (thereby specifying rules for automatic semantics).
- All other semantic declarations are illegal.
- Members in a single `struct` cannot have both varying and uniform semantics.
- Members in a single `struct` cannot have both input and output varying semantics.
- Input semantics may be aliased by multiple variables.
- Output semantics may not be aliased.

Using a `struct` to Define Binding Semantics (Connectors)

Cg implementations may optionally allow the user to avoid the requirement that a binding semantic be specified for every non-uniform input (or output) variable to a top-level program. To avoid this requirement, all the non-uniform variables must be included within a single `struct`. The compiler automatically allocates the elements of this structure to hardware resources in a manner that allows any program that returns this `struct` to interoperate with any program that uses this `struct` as an input.

For various reasons, it is desirable to be able to choose a particular set of rules for automatically performing the allocation of structure elements to hardware resources. The set of rules to be used `<allocation-rule-identifier>` may be specified by an identifier attached to the structure tag using the colon-identifier notation:

```
struct <struct-tag> : <allocation-rule-identifier> { 
  ...
}
```

The allocation of structure elements to hardware resources may be performed in any reproducible manner that depends only on this structure definition (and allocation-rule-identifier). In particular, the allocation algorithm may not depend on how a particular function reads or writes to the elements of the structure.

Implementations may choose to make the specification of an allocation-rule-identifier optional; the omission of the allocation-rule-identifier implies the use of a default set of allocation rules.

It is permissible for some elements of the `struct` to have an explicitly specified binding semantic. The compiler's automatic allocation must honor these explicit bindings. The allowed set of explicitly specified binding semantics is defined by the allocation-rule-identifier. The most common use of this capability is to bind variables to hardware/API resources that write to, or read from, non-programmable parts of the hardware. For example, in a typical vertex-program profile, the output `struct` would contain an element with an explicitly specified `POSITION` semantic. This element is used to control the hardware rasterizer.

For any particular allocation-rule-identifier, additional restrictions may be associated with particular binding semantics. For example, it might be illegal to read from a variable with a particular binding semantic.
How Programs Receive and Return Data

A program is just a non-static function that has been designated as the main entry point at compilation time. A program may receive its input as `in` or `inout` parameters with binding semantics applied to them, or by receiving a connector as one of its input parameters.

Note that a function is allowed to write to a variable that has a binding semantic applied to it. However, this can cause the compiler to create a temporary variable to store the new value. Parameters of type `sampler*` are implicitly `const` variables, and may not be written by a program.

A program may return its output either by specifying a return type for the function that is a connector with appropriate semantics, or returning the output values by passing one or more `out` parameters from the function that have binding semantics applied to them.

Statements

- Statements are expressed just as in C, unless an exception is stated elsewhere in this document.
- `if`, `while`, and `for` require `bool` expressions in the appropriate places.
- Assignment is performed using `=`. Assignment may be used for vectors of size 4 or less, and matrices of size 4x4 or less.
- There is a new statement `discard` that terminates execution of the program and suppresses its output. Profiles are not required to support `discard`. In particular, it is more relevant for fragment profiles than vertex profiles.

New Vector Operators

These new operators are defined for vector types:

- Construction operator: `<typeID>()(...)`
  - This operator builds a vector out of scalar expressions
  - `float4(scalar, scalar, scalar, scalar)`
  - or, out of shorter vectors and scalars
  - `float4(float3, scalar)`
- Swizzle operator: `.`

```c
a = b.xxyz;  // A swizzle operator example
```

- At least one swizzle character must follow the operator.
- These swizzle characters may not be applied to matrices. They may only be applied to vectors or to scalars (to explicitly smear a scalar into a vector):
  - `xyzw = 0123`
  - `rgba = 0123`
- Care is required when swizzling a constant scalar because of ambiguity in the use of the `.` symbol. For example, to create a three-vector from a scalar, use the following:
  - `1..xxx` or `1.0.xxx` or `1.0f.xxx`
- The size of the returned vector is determined by the number of characters. It may be larger than the size of the original vector. For example,
  - `float2(0,1).xxyy = float4(0,0,1,1)`
If only one swizzle character is specified, the result is a scalar rather than an array of size one. For example, the expression `b.y` returns a scalar.

The operator can be applied to a scalar, in which case the result is the same as applying the operator to an array of size 1. For example:

```c
l.xxx = float3(1,1,1);
```

Matrix types also support an element swizzle.

For any matrix type of the form `<type><rows>x<columns>`, the notation `<matrixObject>.m_<row><col>[_<row><col>][…]` can be used to access individual matrix elements (in the case of only one `<row>,<col>` pair), or to construct vectors from elements of a matrix (in the case where more than one `<row>,<col>` pair is present). The row and column numbers are zero-based.

For example,

```c
float4x4 myMatrix;
float myFloatScalar;
float4 myFloatVec4;

// Set myFloatScalar to myMatrix[3][2]
myFloatScalar = myMatrix.m_32;

// Assign the main diagonal of myMatrix to myFloatVec4
myFloatVec4 = myMatrix.m_00_11_22_33;
```

For compatibility with the D3DMatrix data type, Cg also allows one-based swizzles, using a form with the `m` omitted after the `.` symbol:

`<matrixObject>._<row><col>[_<row><col>][…]`

Note that in this form, the indexes for `<row>` and `<col>` are 1-based, rather than the C-standard zero-based. So, the two statements are functionally equivalent:

```c
float4x4 myMatrix;
float4 myVec;

// These two statements are functionally equivalent:
myVec = myMatrix.m_00_23_11_31;
myVec = myMatrix._11_34_22_41;
```

Because of the confusion that can be caused by the 1-based indexing, use of the latter notation is strongly discouraged.

The matrix swizzles may only be applied to matrices. When multiple components are extracted from a matrix using a swizzle, the result is an appropriately sized vector. When a single component is extracted from a matrix using a swizzle, the result is a scalar.

The write-mask operator. can only be applied to an lvalue that is a vector. It selectively suppresses modification of some components and also allows swizzling. The only restriction is that a component cannot be repeated.
Arithmetic Precision and Range

Some hardware may not conform exactly to IEEE arithmetic rules. Fixed-point data types do not have IEEE-defined rules. Meaning optimizations are allowed to produce slightly different results than unoptimized code. Constant folding must be done with approximately the correct precision and range, but is not required to produce bit-exact results.

It is recommended that compilers provide an option to forbid these optimizations, or to guarantee that they are made in bit-exact fashion.

Operator Precedence

This is exactly the same as in C for existing operators. The swizzle/write-mask operator . has the same precedence as the array index [] and structure member operators ..

Expanded Operators

The standard C arithmetic operators (+, -, *, /, %) are extended to support vectors and matrices. Sizes of vectors and matrices must be appropriately matched, according to standard mathematical rules. Smearing, as described earlier, allows relaxation of these rules.

- $M[n][m]$ Matrix with $n$ rows and $m$ columns
- $V[n]$ Vector with $n$ elements
- $\neg V[n] \rightarrow V[n]$ Unary vector negate
- $\neg M[n] \rightarrow M[n]$ Unary matrix negate
- $V[n] \ast V[n] \rightarrow V[n]$ Componentwise $\ast$
- $V[n] \% V[n] \rightarrow V[n]$ Componentwise $\%$
- $V[n] + V[n] \rightarrow V[n]$ Componentwise $+$
- $V[n] - V[n] \rightarrow V[n]$ Componentwise $-$
- $M[n][m] \ast M[n][m] \rightarrow M[n][m]$ Componentwise $\ast$
- $M[n][m] / M[n][m] \rightarrow M[n][m]$ Componentwise $/$
- $M[n][m] \% M[n][m] \rightarrow M[n][m]$ Componentwise $\%$
- $M[n][m] + M[n][m] \rightarrow M[n][m]$ Componentwise $+$
- $M[n][m] - M[n][m] \rightarrow M[n][m]$ Componentwise $-$
Operators

Boolean

| & & | || | ! |

Boolean operators may be applied to `bool` vectors of size four or less, in which case they are applied in element wise fashion to produce a result vector of the same size. Each operand must be a `bool` vector of the same size.

Comparisons

| < > | <= | >= | != | = |

Comparison operators may be applied to numeric vectors. Both operands must be vectors of the same size. The comparison operation is performed in elementwise fashion to produce a `bool` vector of the same size.

Comparison operators may also be applied to `bool` vectors. For the purpose of relational comparisons, `true` is treated as one and `false` is treated as zero. The comparison operation is performed in elementwise fashion to produce a `bool` vector of the same size. Comparison operators may also be applied to numeric or `bool` scalars.

Arithmetic

| + - * / % | ++ | -- | unary- | unary+ |

The arithmetic operator `%` is the remainder operator, as in C. It may only be applied to two operands of `cint` type.

When `/` or `%` is used with two `cint` operands, the C rules for integer `/` and `%` apply.

The C operators that combine assignment with arithmetic operations (such as `+=`) are also supported when the corresponding arithmetic operator is supported by Cg.

Conditional Operator

| ? : |

The expressions in the second and third operands are forbidden to produce side effects. More specifically, these expressions are forbidden to call functions with `out` parameters.

If the first operand is of type `bool`, one of the following must hold for the second and third operands:

- Both operands have compatible structure types.
- Both operands are scalars with numeric or `bool` type.
- Both operands are vectors with numeric or `bool` type, where the two vectors are of the same size, `<= 4`.

If the first operand is a vector (up to size 4) of `bool`, then the conditional selection is performed on an element wise basis. Both the second and third operands must be numeric vectors of the same size as the first operand.
Type Cast Operator
()

The C Comma Operator
,

Reserved Words
- :: (scope operator) [reserved for future use]
- const
- function
- uniform
- in
- out
- inout
- int [reserved for future use]
- float
- bool
- void
- do
- while
- for
- if
- else
- switch [reserved for future use]
- case [reserved for future use]
- default [reserved for future use]
- continue [reserved for future use]
- break [reserved for future use]
- goto [reserved for future use]
- typedef
- struct
- enum [reserved for future use]
- union [reserved for future use]
- discard
- return
- static
- volatile
- extern

Not reserved (but predefined to bool values)
- true
- false
Appendix B.
Language Profiles

This release of the Cg compiler supports following profiles: DirectX 8 vertex, DirectX 8 pixel (PS 1.1), OpenGL ARB vertex, NV2X OpenGL vertex, NV30 OpenGL vertex, and NV30 OpenGL fragment. This appendix describes the language capabilities that are available in each of these three profiles. In each case, these capabilities are a subset of the full capabilities described by the Cg language specification in Appendix A.

The DirectX 8 Vertex Shader Profile

The DirectX 8 Vertex Shader Profile is used to compile for Cg source code to DirectX 8.1 Vertex Shaders. The profile is called dx8vs, and is invoked with the compiler option -profile dx8vs. The dx8vs profile limits Cg to match the capabilities of DirectX Vertex Shaders. This section describes how using the dx8vs profile affects the Cg source code that the user writes. To understand the capabilities of DirectX 8 Vertex Shaders and the code produced by the compiler, refer to the Vertex Shader Reference in the DirectX 8.1 SDK documentation.

Binding Semantics

Binding semantics for varying input data to dx8vs consist of POSITION, BLENDWEIGHT, NORMAL, COLOR0, COLOR1, TESSFACTOR, PSIZE, BLENDINDICES, and TEXCOORD0–TEXCOORD7. One can also use TANGENT and BINORMAL instead of TEXCOORD6 and TEXCOORD7. These map to the input registers in DirectX 8.1 vertex shaders.

Table 5. dx8vs Input Binding Semantics

<table>
<thead>
<tr>
<th>Binding Semantics Name</th>
<th>Corresponding Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>POSITION</td>
<td>Vertex shader input register: v0</td>
</tr>
<tr>
<td>BLENDWEIGHT</td>
<td>Vertex shader input register: v1</td>
</tr>
<tr>
<td>NORMAL</td>
<td>Vertex shader input register: v2</td>
</tr>
<tr>
<td>DIFFUSE, COLOR0</td>
<td>Vertex shader input register: v3</td>
</tr>
<tr>
<td>SPECULAR, COLOR1</td>
<td>Vertex shader input register: v4</td>
</tr>
<tr>
<td>TESSFACTOR</td>
<td>Vertex shader input register: v5</td>
</tr>
<tr>
<td>Binding Semantics Name</td>
<td>Corresponding Data</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>PSIZE</td>
<td>Vertex shader input register: v6</td>
</tr>
<tr>
<td>BLENDINDICES</td>
<td>Vertex shader input register: v7</td>
</tr>
<tr>
<td>TEXCOORD0–TEXCOORD7</td>
<td>Vertex shader input register: v8–v15</td>
</tr>
<tr>
<td>TANGENT</td>
<td>Vertex shader input register: v14</td>
</tr>
<tr>
<td>BINORMAL</td>
<td>Vertex shader input register: v15</td>
</tr>
</tbody>
</table>

Binding semantics for varying output data of `dx8vs` consist of `POSITION`, `FOG`, `COLOR0`, `COLOR1`, `PSIZE`, and `TEXCOORD0–TEXCOORD7`. These map to output registers in DirectX 8.1 vertex shaders.

This profile also supports the additional predefined uniform semantics `Cn`, where `n` is a number in the range `[0..95]` that can be used to tell the compiler to allocate the associated uniform variable in constant register of index `n`.

**Memory**

DirectX 8 vertex shaders have a limited amount of memory for instructions and data. They are limited to 128 instructions. If the compiler needs to produce more than 128 instructions to compile a program, it will report an error. Likewise, there are limited numbers of registers to hold program parameters and temporary results. Specifically, there are 96 read-only vector registers and 12 read/write vector registers. If the compiler needs more registers to compile a program than are available, it will generate an error.

**Statements and Operators**

The `if`, `while`, `do`, and `for` statements are not allowed because there is no branching in DirectX 8 vertex shaders. There are no subroutine calls either, so all functions are inlined. Comparison operators are allowed (`>`, `<`, `>=`, `<=`, `==`, `!=`) and Boolean operators (`|`, `&`, `?:`) are allowed. However, integer operators such as the logic operators (`&`, `|`, `^`, `~`) and `%` are not.
Variable Types

Only float (single-precision floating point) and bool (Boolean) types are allowed. Scalars, vectors, matrices, and arrays are supported. The scalar types are float and bool. The vector types allowed are float2, float3, float4, bool2, bool3, and bool4. Matrices of floatnxm or boolnxm can be declared where n and m can range from 1 to 4. Arrays of scalar, vector and matrix types are allowed for read-only uniform parameters. Structures are not supported yet, except in the case of connectors.

Using Conditionals

Currently, the most efficient use of conditional operators is achieved by casting the result of the conditional operator to a float type for further use. A comparison of two vector values yields a vector of 1's and 0's that can be used to select segments of another vector quantity by multiplication. For example, to zero out all the negative values in the float4 variable a:

\[ a = a * (\text{float4}(a >= 0)); \]

Using Arrays

There are a few differences between array support in this profile and array support in other language such as C. First, in this profile arrays are read-only because only program parameter registers can be indexed and they are read-only. Second, arrays are indexed with floating point expressions rather than with integers. Third, only arrays of scalars, vectors, and matrices are permitted.

Array data is not packed because vertex program indexing does not permit it. Each element of the array takes a single 4-float program parameter register. For example, float arr[10], float2 arr[10], float3 arr[10], and float4 arr[10] all consume 10 program parameter registers.

It is more efficient to access an array of vectors than an array of matrices. Accessing a matrix requires a floor calculation, followed by a multiply by a constant to compute the register index. Because vectors (and scalars) take one register, neither the floor nor the multiply is needed. It is faster to do matrix skinning using arrays of vectors with a premultiplied index than using arrays of matrices.

Constants

Literal constants can be used with this profile, but it is not possible to store them in the program itself. Instead the compiler will issue, as comments, a list of program parameter registers and the constants that need to be loaded into them. The Cg run-time system will handle loading the constants, as directed by the compiler. **If the Cg run-time system is not used, it is the responsibility of the programmer to make sure that the constants are loaded properly.**
The DirectX 8 Pixel Shader Profile

The Cg compiler (`cgc.exe`) has a dx8ps profile that generates DirectX 8 pixel shader output. (This can be accomplished with the command line option—`-profile dx8ps`). Currently, the output is generated for version 1.1 of the DirectX 8 pixel shaders.

The underlying instruction set and machine architecture limit programmability, compared to what is allowed by Cg's constructs. Thus, the dx8ps profile has several restrictions on what can be used and what cannot be used. For more details about underlying instruction set, its capabilities, and its limitations, please refer to MSDN’s documentation of DirectX 8 pixel shaders. This document describes the capabilities and restrictions of Cg when using the dx8ps profile.

The capabilities of the dx8ps profile can be categorized as texture shader capabilities and arithmetic operation capabilities. In a dx8ps profile program, you are limited to using a maximum of four texture shader stages and eight arithmetic operation stages. Since these numbers are quite small, users need to be very aware of this limitation while writing Cg code. The profile also restricts when a texture shader operation or arithmetic operation can occur in the program.

A texture shader stage may not have any dependency on the output of an arithmetic stage. There are certain simple arithmetic operations that can be applied to inputs of texture shader stages and to inputs and outputs of arithmetic operation stages without consuming an arithmetic operation stage. From here on, these capabilities are referred to as input modifiers and output modifiers.

Data Types

Most DirectX 8 operations occur on signed clamped [−1.0, 1.0] or unsigned clamped [0.0, 1.0] floats. Because there are no valid operations allowed in the instruction set for DirectX 8 on the float2 type, the float2 type is not allowed in this profile. The dx8ps profile allows only float as a base type. The Boolean type, bool, is not allowed in this profile.

Modifiers

You can use modifiers to do some simple arithmetic operations without consuming an arithmetic operation stage. The modifiers get rolled into the operation to which they are applied, thus optimizing code and freeing resources in some cases. Some input modifiers are also allowed in texture shader stages which otherwise do not allow input from output of an arithmetic operation stage. The dx8ps arithmetic operation stages allow operand, expand(operand), expand_inv(operand), halfbias(operand), halfbias_inv(operand), and uclamp_inv(operand) as input modifiers. The dx8ps texture operation stages, with the exception of offsettex* built-ins, allow only expand(operand) as input modifier. dx8ps arithmetic stages allow uclamp(result), 2*result, 4*result, and 0.5*result as output modifiers. Table 7 summarizes how different DirectX 8 instruction set modifiers relate to built-ins provided by the Cg compiler.
Table 7. DirectX 8 Instruction Set Modifiers

<table>
<thead>
<tr>
<th>Instruction Set Modifiers</th>
<th>Cg Compiler Built-ins</th>
</tr>
</thead>
<tbody>
<tr>
<td>expand</td>
<td>_bx2 input modifier</td>
</tr>
<tr>
<td>expand_inv</td>
<td>1 - _bx2 input modifier</td>
</tr>
<tr>
<td>halfbias</td>
<td>_bias input modifier</td>
</tr>
<tr>
<td>halfbias_inv</td>
<td>1 - _bias input modifier</td>
</tr>
<tr>
<td>uclamp_inv</td>
<td>1 - operand input modifier</td>
</tr>
<tr>
<td>uclamp</td>
<td>_sat output modifier</td>
</tr>
<tr>
<td>~operand</td>
<td>negate input modifier</td>
</tr>
<tr>
<td>2<em>result,4</em>result,0.5*result</td>
<td>_x2, _x4, _d2 output modifiers respectively</td>
</tr>
</tbody>
</table>

Appendix B. Language Profiles

DU-00504-001-51

NVIDIA
Functions supported by $\text{dx8ps}$

Table 8 presents the $\text{dx8ps}$ profile functions that are provided by the Cg compiler. These functions are not supported in any other profiles.

Table 8. $\text{dx8ps}$ Profile Functions

<table>
<thead>
<tr>
<th>Fragment Program Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dot3(a,b)</td>
<td>dot product of 3-vectors $a$ and $b$ Returns a scalar</td>
</tr>
<tr>
<td>dot3_rgb(a,b)</td>
<td>dot product of 3-vectors $a$ and $b$ Returns a 3-vector</td>
</tr>
<tr>
<td>dot3_rgba(a,b)</td>
<td>dot product of 3-vectors $a$ and $b$ Returns a 4-vector</td>
</tr>
<tr>
<td>mad(a,b,c)</td>
<td>$a \times b + c$ where $a$, $b$, and $c$ are of matching scalar/vector type</td>
</tr>
<tr>
<td>mad(a,b,c)</td>
<td>$a \times b + c$ where $b$ and $c$ are of matching scalar/vector type and $a$ is a scalar</td>
</tr>
<tr>
<td>lerp(a,b,f)</td>
<td>$(1-f) \times a + b \times f$ where $a$, $b$, and $f$ are of matching scalar/vector type</td>
</tr>
<tr>
<td>lerp(a,b,f)</td>
<td>$(1-f) \times a + b \times f$ where $a$ and $b$ are of matching type and $f$ is a scalar</td>
</tr>
<tr>
<td>select(testval,val1,val2)</td>
<td>Chooses val1 or val2 depending on value of testval: if ( testval &gt; 0.5 ) val1 else val2 where val1 and val2 are of matching scalar/vector type and testval is a scalar</td>
</tr>
<tr>
<td>expand(val)</td>
<td>$2 \times (\max(0,\text{val})-0.5)$ Can be used to get _bx2 input modifier functionality in DirectX 8 pixel shaders</td>
</tr>
<tr>
<td>expand_inv(val)</td>
<td>$-(2 \times (\max(0,\text{val})-0.5))$ Can be used to get negated _bx2 input modifier functionality in DirectX 8 pixel shaders</td>
</tr>
</tbody>
</table>
## Fragment Program Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uclamp(val)</code></td>
<td>( \min(1, \max(0, val)) )</td>
</tr>
<tr>
<td></td>
<td>Can be used to get _sat output modifier functionality in DirectX 8 pixel shaders</td>
</tr>
<tr>
<td><code>uclamp_inv(val)</code></td>
<td>( 1-\min(\max(0,val),1) )</td>
</tr>
<tr>
<td><code>halfbias(val)</code></td>
<td>( \max(0,val) - 0.5 )</td>
</tr>
<tr>
<td></td>
<td>Can be used to get _bias input modifier functionality in DirectX 8 pixel shaders</td>
</tr>
<tr>
<td><code>halfbias_inv(val)</code></td>
<td>( -\max(0,val) + 0.5 )</td>
</tr>
<tr>
<td></td>
<td>Can be used to get ( 1 - (\text{val}) _\text{bias} ) input modifier functionality in DirectX 8 pixel shaders</td>
</tr>
<tr>
<td><code>tex1D(sampler1D tex)</code></td>
<td>1D texture lookup using the texture coordinates associated with texture tex</td>
</tr>
<tr>
<td><code>tex2D(sampler2D tex)</code></td>
<td>2D texture lookup using the texture coordinates associated with texture tex</td>
</tr>
<tr>
<td><code>tex3D(sampler3D tex)</code></td>
<td>3D texture lookup using the texture coordinates associated with texture tex</td>
</tr>
<tr>
<td><code>texRECT(samplerRECT tex)</code></td>
<td>Non-power-of-two texture lookup using the texture coordinates associated with texture tex</td>
</tr>
<tr>
<td><code>texCUBE(samplerCUBE tex)</code></td>
<td>Cube-map texture lookup using the texture coordinates associated with texture tex</td>
</tr>
<tr>
<td><code>kill(float4 texcoords)</code></td>
<td>Discards a pixel if any component of texcoords is less than 0</td>
</tr>
<tr>
<td><code>passthrough(float4 texcoords)</code></td>
<td>Returns a 4-vector result with ( \text{result}.r = \text{texcoords}.x ) ( \text{result}.g = \text{texcoords}.y ) ( \text{result}.b = \text{texcoords}.z ) ( \text{result}.a = \text{texcoords}.w )</td>
</tr>
<tr>
<td></td>
<td>Can be used to get functionality of texcoord in DirectX 8 pixel shaders</td>
</tr>
</tbody>
</table>
### Fragment Program Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{offsettex2D}</td>
<td>Performs a 2D texture lookup with ( s' = s + m00 \times \text{prevlookup}.x + m01 \times \text{prevlookup}.y ) ( t' = t + m10 \times \text{prevlookup}.x + m11 \times \text{prevlookup}.y ) where ((s,t)) are texture coordinates associated with texture \texttt{tex} \newline Returns a 4-vector result with \newline \texttt{result}.r = \texttt{texlookup}(s',t').r \texttt{result}.g = \texttt{texlookup}(s',t').g \texttt{result}.b = \texttt{texlookup}(s',t').b \texttt{result}.a = \texttt{texlookup}(s',t').a \newline Can be used to get functionality of \texttt{texbem} in DirectX 8 pixel shaders</td>
</tr>
<tr>
<td>\texttt{offsettex2DScaleBias}</td>
<td>Performs a 2D texture lookup with ( s' = s + m00 \times \text{prevlookup}.x + m01 \times \text{prevlookup}.y ) ( t' = t + m10 \times \text{prevlookup}.x + m11 \times \text{prevlookup}.y ) where ((s,t)) are texture coordinates associated with texture \texttt{tex} \newline Returns a 4-vector result with \newline \texttt{result}.r = M \times \texttt{texlookup}(s',t').r \texttt{result}.g = M \times \texttt{texlookup}(s',t').g \texttt{result}.b = M \times \texttt{texlookup}(s',t').b \texttt{result}.a = M \times \texttt{texlookup}(s',t').a \newline where (M = \text{scale} \times \text{prevlookup}.z + \text{bias}) \newline Can be used to get functionality of \texttt{texbem} in DirectX 8 pixel shaders</td>
</tr>
<tr>
<td>\texttt{offsettexRECT}</td>
<td>Performs a RECT texture lookup with ( s' = s + m00 \times \text{prevlookup}.x + m01 \times \text{prevlookup}.y ) ( t' = t + m10 \times \text{prevlookup}.x + m11 \times \text{prevlookup}.y ) where ((s,t)) are texture coordinates associated with texture \texttt{tex} \newline Returns a 4-vector result with \newline \texttt{result}.r = \texttt{texlookup}(s',t').r \texttt{result}.g = \texttt{texlookup}(s',t').g \texttt{result}.b = \texttt{texlookup}(s',t').b \texttt{result}.a = \texttt{texlookup}(s',t').a \newline Can be used to get functionality of \texttt{texbem} in DirectX 8 pixel shaders</td>
</tr>
</tbody>
</table>
Fragment Program Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>offsettexRECTScaleBias(samplerRECT tex, float4 prevlookup, float m00, float m01, float m10, float m11, float scale, float bias)</code></td>
<td>Performs a RECT texture lookup with $s' = s + m00 \cdot prevlookup.x + m01 \cdot prevlookup.y$ and $t' = t + m10 \cdot prevlookup.x + m11 \cdot prevlookup.y$ where $(s,t)$ are the texture coordinates associated with texture <code>tex</code>. Returns a 4-vector <code>result</code> with $result.r = M \cdot texlookup(s',t').r$, $result.g = M \cdot texlookup(s',t').g$, $result.b = M \cdot texlookup(s',t').b$, and $result.a = M \cdot texlookup(s',t').a$. Where $M = scale \cdot prevlookup.z + bias$. Can be used to get functionality of <code>texbeml</code> in DirectX 8 pixel shaders.</td>
</tr>
<tr>
<td><code>dependent_ar(sampler2D tex, float4 prevlookup)</code></td>
<td>Performs a 2D texture lookup with texture coordinates $s' = prevlookup.a$ and $t' = prevlookup.r$. Can be used to get functionality of <code>texreg2ar</code> in DirectX 8 pixel shaders.</td>
</tr>
<tr>
<td><code>dependent_gb(sampler2D tex, float4 prevlookup)</code></td>
<td>Performs a 2D texture lookup with texture coordinates $s = prevlookup.g$ and $t = prevlookup.b$. Returns a 4-vector <code>result</code> with $result.r = texlookup(s',t').r$, $result.g = texlookup(s',t').g$, $result.b = texlookup(s',t').b$, and $result.a = texlookup(s',t').a$. Can be used to get functionality of <code>texreg2gb</code> in DirectX pixel shaders.</td>
</tr>
<tr>
<td><code>tex2D_dp3x2(sampler2D tex, float4 intermediate_stage_coord, float4 prevlookup)</code></td>
<td>Performs a 2D texture lookup with texture coordinates $s' = intermediate_stage_coord.x \cdot prevlookup.x + intermediate_stage_coord.y \cdot prevlookup.y + intermediate_stage_coord.z \cdot prevlookup.z$ and $t' = s \cdot prevlookup.x + t \cdot prevlookup.y + r \cdot prevlookup.z$ where $(s,t,r)$ are the texture coordinates associated with texture <code>tex</code>. Returns a 4-vector <code>result</code> with $result.r = texlookup(s',t').r$, $result.g = texlookup(s',t').g$, $result.b = texlookup(s',t').b$, and $result.a = texlookup(s',t').a$.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>texRECT_dp3x2(samplerRECT tex, float4 intermediate_stage_coord, float4 prevlookup)</td>
<td>Performs a RECT texture lookup with texture coordinates $s' = intermediate_stage_coord_x<em>prevlookup_x + intermediate_stage_coord_y</em>prevlookup_y + intermediate_stage_coord_z<em>prevlookup_z$ $t' = s</em>prevlookup_x + t<em>prevlookup_y + r</em>prevlookup_z$ where $(s,t,r)$ are the texture coordinates associated with texture tex Returns a 4-vector result with result.r = texlookup($s',t',r'$).r result.g = texlookup($s',t',r'$).g result.b = texlookup($s',t',r'$).b result.a = texlookup($s',t',r'$).a</td>
</tr>
<tr>
<td>tex3D_dp3x3(sampler3D tex, float4 intermediate_stage_coord1, float4 intermediate_stage_coord2, float4 prevlookup)</td>
<td>Performs a 3D texture lookup with texture coordinates $s' = intermediate_stage_coord_1_x<em>prevlookup_x + intermediate_stage_coord_1_y</em>prevlookup_y + intermediate_stage_coord_1_z<em>prevlookup_z$ $t' = intermediate_stage_coord_2_x</em>prevlookup_x + intermediate_stage_coord_2_y<em>prevlookup_y + intermediate_stage_coord_2_z</em>prevlookup_z$ $r' = s<em>prevlookup_x + t</em>prevlookup_y + r*prevlookup_z$ where $(s,t,r)$ are texture coordinates associated with texture tex Returns a 4-vector result with result.r = texlookup($s',t',r'$).r result.g = texlookup($s',t',r'$).g result.b = texlookup($s',t',r'$).b result.a = texlookup($s',t',r'$).a</td>
</tr>
<tr>
<td>texCUBE_dp3x3(samplerCUBE tex, float4 intermediate_stage_coord1, float4 intermediate_stage_coord2, float4 prevlookup)</td>
<td>Performs a CUBE texture lookup with texture coordinates $s' = intermediate_stage_coord_1_x<em>prevlookup_x + intermediate_stage_coord_1_y</em>prevlookup_y + intermediate_stage_coord_1_z<em>prevlookup_z$ $t' = intermediate_stage_coord_2_x</em>prevlookup_x + intermediate_stage_coord_2_y<em>prevlookup_y + intermediate_stage_coord_2_z</em>prevlookup_z$ $r' = s<em>prevlookup_x + t</em>prevlookup_y + r*prevlookup_z$ where $(s,t,r)$ are texture coordinates associated with texture tex Returns a 4-vector result with result.r = texlookup($s',t',r'$).r result.g = texlookup($s',t',r'$).g result.b = texlookup($s',t',r'$).b result.a = texlookup($s',t',r'$).a</td>
</tr>
</tbody>
</table>
Appendix B. Language Profiles

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
</table>
| texCUBE_reflect_dp3x3(samplerCUBE tex, float4 intermediate_stage_coord1, float4 intermediate_stage_coord2, float4 prevlookup) | Performs a CUBE texture lookup with texture coordinates

\[
\begin{align*}
    s' &= 2 \times Ux \times (Ux*Ex + Uy*Ey + Uz*Ez) / (Ux*Ux + Uy*Uy + Uz*Uz) - Ex \\
    t' &= 2 \times Uy \times (Ux*Ex + Uy*Ey + Uz*Ez) / (Ux*Ux + Uy*Uy + Uz*Uz) - Ey \\
    r' &= 2 \times Uz \times (Ux*Ex + Uy*Ey + Uz*Ez) / (Ux*Ux + Uy*Uy + Uz*Uz) - Ez
\end{align*}
\]

where

\[
\begin{align*}
    Ex &= intermediate\_stage\_coord1.w \\
    Ey &= intermediate\_stage\_coord2.w \\
    Ez &= q \\
    Ux &= intermediate\_stage\_coord1.x*prevlookup.x + intermediate\_stage\_coord1.y*prevlookup.y + intermediate\_stage\_coord1.z*prevlookup.z \\
    Uy &= intermediate\_stage\_coord2.x*prevlookup.x + intermediate\_stage\_coord2.y*prevlookup.y + intermediate\_stage\_coord2.z*prevlookup.z \\
    Uz &= s*prevlookup.x + t*prevlookup.y + r*prevlookup.z
\end{align*}
\]

and \((s',t',r,q)\) are the texture coordinates associated with texture \(tex\).

Returns a 4-vector \(result\) with

\[
\begin{align*}
    result.r &= texlookup(s',t',r').r \\
    result.g &= texlookup(s',t',r').g \\
    result.b &= texlookup(s',t',r').b \\
    result.a &= texlookup(s',t',r').a
\end{align*}
\]

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
</table>
| texCUBE_reflect_eye_dp3x3(samplerCUBE tex, float4 intermediate_stage_coord1, float4 intermediate_stage_coord2, float4 prevlookup, float3 eye) | Performs a CUBE texture lookup with texture coordinates

\[
\begin{align*}
    s' &= 2 \times Ux \times (Ux*Ex + Uy*Ey + Uz*Ez) / (Ux*Ux + Uy*Uy + Uz*Uz) - Ex \\
    t' &= 2 \times Uy \times (Ux*Ex + Uy*Ey + Uz*Ez) / (Ux*Ux + Uy*Uy + Uz*Uz) - Ey \\
    r' &= 2 \times Uz \times (Ux*Ex + Uy*Ey + Uz*Ez) / (Ux*Ux + Uy*Uy + Uz*Uz) - Ez
\end{align*}
\]

where

\[
\begin{align*}
    Ex &= eye.x \\
    Ey &= eye.y \\
    Ez &= eye.z \\
    Ux &= intermediate\_stage\_coord1.x*prevlookup.x + intermediate\_stage\_coord1.y*prevlookup.y + intermediate\_stage\_coord1.z*prevlookup.z \\
    Uy &= intermediate\_stage\_coord2.x*prevlookup.x + intermediate\_stage\_coord2.y*prevlookup.y + intermediate\_stage\_coord2.z*prevlookup.z \\
    Uz &= s*prevlookup.x + t*prevlookup.y + r*prevlookup.z
\end{align*}
\]

and \((s',t',r,q)\) are texture coordinates associated with texture \(tex\).

Returns a 4-vector \(result\) with

\[
\begin{align*}
    result.r &= texlookup(s',t',r').r \\
    result.g &= texlookup(s',t',r').g \\
    result.b &= texlookup(s',t',r').b \\
    result.a &= texlookup(s',t',r').a
\end{align*}
\]
Texture Shader Operations

All texture shader capabilities of the underlying instruction set are exposed using built-in functions. The function description table in this profile includes a detailed list of specific built-in functions. Texture shader operations that do a lookup take as input a corresponding uniform parameter specifying a texture sampler. In this case, there is an implicit texture coordinate that must be specified in the input connector for the texture stage. This texture coordinate is used by the underlying hardware to sample the texture sampler object. Note that this texture coordinate may not be referred to in Cg code, even though it is required. Texture shader operations that do arithmetic operations on input texture coordinates take texture coordinates from the input connector as input parameters. Thus, the dx8ps profile requires you to specify texture coordinates in the input connector for every texture shader stage that is needed.

If unspecified, the Cg compiler allocates bindings between texture stage numbers and texture coordinates from the input connector. By default, the bindings are based on the order in which the texture coordinates are listed in the input connector. For uniform texture sampler parameters, if you do not specify the binding, the Cg compiler allocates the binding between texture stage number and the sampler object based on the context in which it was used. All binding information is listed in the output header so that the runtime or application using it can find out the bindings that the compiler used during compilation.

Manual Assignment of Bindings

As just described, the Cg compiler can determine bindings between texture stages/units and uniform sampler parameters/TEXn connector members automatically. This automatic assignment is based on the context in which uniform sampler parameters are used and on the order of TEXn interpolants in the connector structure.

If you want to specify bindings/semantics between texture stages/units and uniform parameters/connector members to match their application, you need to specify bindings/semantics for all sampler uniform parameters and TEXn type of members of connector structure that are used in the program. Partially specified bindings/semantics may not work in all cases. Fundamentally, this restriction is due to the close coupling between texture samplers and texture coordinates in DirectX 8 Pixel Shaders, version 1.1.

Arithmetic Operations

Arithmetic operations that are compatible with the underlying instruction set are allowed in Cg using modifiers, binary operators (+, −, and *), and built-in functions (mad(), lerp(), select(), dot3*()). All other arithmetic operators are disallowed. Modifiers and built-ins should be used to get optimal code whenever possible. In arithmetic operators, the types of operands need to match for + and −. Multiplication of scalars and vectors is allowed. All other arithmetic and Boolean operators are not supported by this profile.
Binding Semantics and Input/Output Varying Data

The input varying data for `dx8ps` use the same binding semantics as that of output varying data of the `dx8vs` profile. Binding semantics for varying input data of `dx8ps` consists of `COLOR0`, `COLOR1`, `TEXCOORD0`, `TEXCOORD1`, `TEXCOORD2`, and `TEXCOORD3`. These map to output registers in DirectX 8.1 vertex shaders.

Table 9. `dx8ps` Input Binding Semantics

<table>
<thead>
<tr>
<th>Binding Semantics Name</th>
<th>Corresponding Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLOR0-COLOR1</td>
<td>Input color values: v0, v1</td>
</tr>
<tr>
<td>TEXCOORD0-TEXCOORD3</td>
<td>Input texture coordinates</td>
</tr>
</tbody>
</table>

Additionally, the profile allows `POSITION`, `FOG`, `PSIZE`, `TEXCOORD4`, `TEXCOORD5`, `TEXCOORD6`, and `TEXCOORD7` to be present as semantics on members of a structure being passed in as a varying input structure, provided members with these semantics are not referenced. This allows Cg programs to have the same structure specify the varying output of `dx8vs` profile program and the varying input of a `dx8ps` profile program. The profile implementation imposes the extra restriction that texture coordinates in the vertex-to-fragment connecting structure must be `float4`.

The output varying data for the `dx8ps` profile use `COLOR` for binding semantics to specify the output color.

This profile supports two additional predefined uniform semantics:

- `Cn`, where `n` is a number in the range `[0..7]` that can be used to tell the compiler to allocate the associated uniform variable in constant register of index `n`
- `texunitn`, where `n` is a number in the range `[0..3]` that can be used to tell the compiler to allocate a sampler object in texture unit `n`

Uniform Parameters

The compiler always allocates scalar `float` uniform parameters in the alpha channel or `w` component of a constant register. This is due to the instruction set limitation that scalar arithmetic operations can be done only on the alpha channel or `w` component. The compiler allocates bindings between `float*` uniform parameters and registers if unspecified by the developer. Bindings for uniform parameters are listed in the output header to be used by applications as well as the run-time library.
Other Language Constructs

The \texttt{dx8ps} profile does not allow the following:

- generic structures (only connector structures are allowed)
- arrays
- vector-constructors
- conditional assignments
- \texttt{if} statements
- \texttt{while} statements
- \texttt{do} statements
- \texttt{for} statements

The \texttt{dx8ps} profile allows \texttt{.w/.a, .xyz/.rgb} and \texttt{.xyzw/.rgba} as legal swizzles for vectors. This profile also allows \texttt{.www/.aaa, .www/.aaaa} for vectors of length 4 and \texttt{.xxx/.rrr, .www/.aaa} for scalars. This can be used to get the alpha channel/\texttt{w} component as an input to arithmetic operations that require vectors. Since the underlying instruction set does not allow arithmetic operations on individual \texttt{r}, \texttt{g}, and \texttt{b} channels, other swizzles are not allowed. The DirectX 8 instruction set allows putting the blue channel value into the alpha channel. Currently, this capability is not supported in the \texttt{dx8ps} profile.

DirectX 8 Pixel Shader Examples

The following examples illustrate how a developer can use Cg to achieve DirectX 8’s pixel shader functionality.

Example 1

```cpp
struct myVertexOut {
    float4 COL0 : COLOR0;
    float4 texCoord0 : TEXCOORD0;
    float4 texCoord1 : TEXCOORD1;
};

struct myFragment {
    float4 COL : COLOR;
};

myFragment main( myVertexOut I,
    uniform sampler2D diffuseMap,
    uniform sampler2D normalMap )
{
    myFragment O;

    float4 diffuseTexColor = tex2D( diffuseMap );
    float4 normal = expand( tex2D( normalMap ) );
    float3 light_vector = expand( I.COL0.rgb );
    ```
Example 2

```c
struct myVertexOut {
    float4 texCoord0 : TEXCOORD0;
    float4 texCoord1 : TEXCOORD1;
    float4 texCoord2 : TEXCOORD2;
    float4 texCoord3 : TEXCOORD3;
};

struct myFragment {
    float4 COL : COLOR;
};

myFragment main( myVertexOut I,
    uniform sampler2D normalMap,
    uniform sampler2D intensityMap,
    uniform sampler2D colorMap )
{
    myFragment O;

    float4 normal = tex2D( normalMap );
    float4 intensity = tex2D_dp3x2(intensityMap,
        I.texCoord1,
        expand( normal ) );
    float4 color = tex2D( colorMap );
    O.COL = color * intensity;
    return O;
}
```
The vp20 Vertex Program Profile

The vp20 Vertex Program Profile is used to compile Cg source code to vertex programs for use by the NV_vertex_program OpenGL extension. The profile is called vp20 and it is invoked with the compiler option -profile vp20. The vp20 profile limits Cg to match the capabilities of the NV_vertex_program extension. NV_vertex_program has the same capabilities as DirectX 8 vertex shaders, so the limitations that this profile places on the Cg source code written by the programmer is the same as the DirectX 8 Vertex Shader profile. See The DirectX 8 Vertex Shader Profile on page 137 for a full explanation of the data types, statements, and operators supported by this profile. To understand the capabilities of NV_vertex_program and the code produced by the compiler using the vp20 profile, see the GL_NV_vertex_program extension documentation.

Aside from the syntax of the compiler output, the only difference between the vp20 Vertex Shader profile and the DirectX 8 Vertex Shader profile is that the vp20 profile supports two more outputs. The vertex-to-fragment connector can have the additional fields BFC0 (for back-facing primary color) and BFC1 (for back-facing secondary color).

Binding Semantics

The set of binding semantics for varying input data to vp20 consists of POSITION, BLENDWEIGHT, NORMAL, COLOR0, COLOR1, TESSFACTOR, PSIZE, BLENDINDICES, and TEXCOORD0–TEXCOORD7. One can also use TANGENT and BINORMAL instead of TEXCOORD6 and TEXCOORD7. A second set of binding semantics, ATTR0–ATTR15, can also be used. The two sets act as aliases to each other.

Table 10. vp20 Input Binding Semantics

<table>
<thead>
<tr>
<th>Binding Semantics Name</th>
<th>Corresponding Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>POSITION, ATTR0</td>
<td>Input Vertex, Generic Attribute 0</td>
</tr>
<tr>
<td>BLENDWEIGHT, ATTR1</td>
<td>Input vertex weight, Generic Attribute 1</td>
</tr>
<tr>
<td>NORMAL, ATTR2</td>
<td>Input normal, Generic Attribute 2</td>
</tr>
<tr>
<td>DIFFUSE, COLOR0, ATTR3</td>
<td>Input primary color, Generic Attribute 3</td>
</tr>
<tr>
<td>SPECULAR, COLOR1, ATTR4</td>
<td>Input secondary color, Generic Attribute 4</td>
</tr>
<tr>
<td>TESSFACTOR, FOGCOORD, ATTR5</td>
<td>Input fog coordinate, Generic Attribute 5</td>
</tr>
<tr>
<td>PSIZE, ATTR6</td>
<td>Input point size, Generic Attribute 6</td>
</tr>
<tr>
<td>BLENDINDICES, ATTR7</td>
<td>Generic Attribute 7</td>
</tr>
<tr>
<td>TEXCOORD0–TEXCOORD7, ATTR8–ATTR15</td>
<td>Input texture coordinates (texcoord0-texcoord7), Generic Attributes 8-15</td>
</tr>
<tr>
<td>TANGENT, ATTR14</td>
<td>Generic Attribute 14</td>
</tr>
<tr>
<td>BINORMAL, ATTR15</td>
<td>Generic Attribute 15</td>
</tr>
</tbody>
</table>
The set of binding semantics for varying output data of \texttt{vp20} consists of \texttt{POSITION}, \texttt{FOG}, \texttt{COLOR0}, \texttt{COLOR1}, \texttt{PSIZE}, \texttt{TEXCOORD0}, \texttt{TEXCOORD2}, and \texttt{TEXCOORD3}. Additionally, a set of binding semantics \texttt{HPOS}, \texttt{COL0}, \texttt{COL1}, \texttt{BCOL0}, \texttt{BCOL1}, \texttt{TEX0–TEX7}, \texttt{FOGC}, \texttt{PSIZ} can be used. These binding semantics map to \texttt{NV_vertex_program} output registers. The two sets act as aliases to each other.

Table 11. \texttt{vp20} Output Binding Semantics

<table>
<thead>
<tr>
<th>Binding Semantics Name</th>
<th>Corresponding Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{POSITION}, \texttt{HPOS}</td>
<td>Output position</td>
</tr>
<tr>
<td>\texttt{PSIZE}, \texttt{PSIZ}</td>
<td>Output point size</td>
</tr>
<tr>
<td>\texttt{FOG}, \texttt{FOGC}</td>
<td>Output fog coordinate</td>
</tr>
<tr>
<td>\texttt{COLOR0}, \texttt{COL0}</td>
<td>Output primary color</td>
</tr>
<tr>
<td>\texttt{COLOR1}, \texttt{COL1}</td>
<td>Output secondary color</td>
</tr>
<tr>
<td>\texttt{BCOL0}</td>
<td>Output backface primary color</td>
</tr>
<tr>
<td>\texttt{BCOL1}</td>
<td>Output backface secondary color</td>
</tr>
<tr>
<td>\texttt{TEXCOORD0–TEXCOORD3}, \texttt{TEX0–TEX3}</td>
<td>Output texture coordinates</td>
</tr>
</tbody>
</table>

The profile also allows \texttt{WPOS} to be present as binding semantics on a member of a structure of a varying output data structure, provided the member with this binding semantics is not referenced. This allows Cg programs to have the same structure specify the varying output of a \texttt{vp20} profile program and the varying input of an \texttt{fp30} profile program.

This profile supports the additional predefined uniform semantics \texttt{Cn}, where \texttt{n} is a number in the range \([0..95]\) that can be used to tell the compiler to allocate the associated uniform variable in constant register of index \texttt{n}. 
The **vp30** Vertex Program Profile

The **vp30** Vertex Program Profile is used to compile Cg source code to vertex programs for use by the NV_vertex_program2 OpenGL extension. The profile is called **vp30** and it is invoked with the compiler option `-profile vp30`. The **vp30** profile limits Cg to match the capabilities of the NV_vertex_program2 extension.

This profile is a superset of the **vp20** profile. Any program that compiles for the **vp20** profile should also compile for the **vp30** profile, although the converse is not true.

The additional capabilities of the **vp30** profile, beyond those of **vp20** are:
- `for` and `while` loops
- Full support for `if/else`

**Binding Semantics**

The set of binding semantics for varying input data to **vp30** consists of `POSITION, BLENDEIGHT, NORMAL, COLOR0, COLOR1, TESSFACTOR, PSIZE, BLENDINDICES,` and `TEXCOORD0–TEXCOORD7`. One can also use `TANGENT` and `BINORMAL` instead of `TEXCOORD6` and `TEXCOORD7`. Additionally, a set of binding semantics `ATTR0–ATTR14, and ATTR15` can be used. These binding semantics map to NV_vertex_program2 input attribute parameters. The two sets act as aliases to each other.

Table 12. **vp30** Input Binding Semantics

<table>
<thead>
<tr>
<th>Binding Semantics Name</th>
<th>Corresponding Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>POSITION, ATTR0</td>
<td>Input Vertex, Generic Attribute 0</td>
</tr>
<tr>
<td>BLENDWEIGHT, ATTR1</td>
<td>Input vertex weight, Generic Attribute 1</td>
</tr>
<tr>
<td>NORMAL, ATTR2</td>
<td>Input normal, Generic Attribute 2</td>
</tr>
<tr>
<td>DIFFUSE, COLOR0, ATTR3</td>
<td>Input primary color, Generic Attribute 3</td>
</tr>
<tr>
<td>SPECULAR, COLOR1, ATTR4</td>
<td>Input secondary color, Generic Attribute 4</td>
</tr>
<tr>
<td>TESSFACTOR, FOCOORD,</td>
<td>Input fog coordinate, Generic Attribute 5</td>
</tr>
<tr>
<td>ATTR5</td>
<td></td>
</tr>
<tr>
<td>PSIZE, ATTR6</td>
<td>Input point size, Generic Attribute 6</td>
</tr>
<tr>
<td>BLENDINDICES, ATTR7</td>
<td>Generic Attribute 7</td>
</tr>
<tr>
<td>TEXCOORD0–TEXCOORD7,</td>
<td>Input texture coordinates (texcoord0-texcoord7), Generic Attributes 8-15</td>
</tr>
<tr>
<td>ATTR8–ATTR15</td>
<td></td>
</tr>
<tr>
<td>TANGENT, ATTR14</td>
<td>Generic Attribute 14</td>
</tr>
<tr>
<td>BINORMAL, ATTR15</td>
<td>Generic Attribute 15</td>
</tr>
</tbody>
</table>

The set of binding semantics for varying output data of **vp30** consists of `POSITION, FOG, COLOR0, COLOR1, PSIZE, TEXCOORD0–TEXCOORD6,` and `TEXCOORD7`.  

Cg Language Toolkit
Additionally, a set of binding semantics HPOS, COL0, COL1, BCOL0, BCOL1, TEX0–TEX7, FOGC, PSIZ, CLP0, CLP1, CLP2, CLP3, CLP4, CLP5 can be used. These binding semantics map to NV_vertex_program2 output registers. The two sets act as aliases to each other.

Table 13. \textit{vp30} Output Binding Semantics

<table>
<thead>
<tr>
<th>Binding Semantics Name</th>
<th>Corresponding Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>POSITION, HPOS</td>
<td>Output position</td>
</tr>
<tr>
<td>PSIZE, PSIZ</td>
<td>Output point size</td>
</tr>
<tr>
<td>FOG, FOGC</td>
<td>Output fog coordinate</td>
</tr>
<tr>
<td>COLOR0, COL0</td>
<td>Output primary color</td>
</tr>
<tr>
<td>COLOR1, COL1</td>
<td>Output secondary color</td>
</tr>
<tr>
<td>BCOL0</td>
<td>Output backface primary color</td>
</tr>
<tr>
<td>BCOL1</td>
<td>Output backface secondary color</td>
</tr>
<tr>
<td>TEXCOORD0–TEXCOORD7, TEX0–TEX7</td>
<td>Output texture coordinates</td>
</tr>
<tr>
<td>CLP0–CL5</td>
<td>Output Clip distances</td>
</tr>
</tbody>
</table>

The profile allows \textit{WPOS} to be present as binding semantics on a member of a structure of a varying output data structure, provided the member with this binding semantics is not referenced. This allows Cg programs to have same structure specify the varying output of a \textit{vp30} profile program and the varying input of an \textit{fp30} profile program.

This profile supports the additional predefined uniform semantics \textit{Cn}, where \textit{n} is a number in the range \([0..255]\) that can be used to tell the compiler to allocate the associated uniform variable in constant register of index \textit{n}. 
The fp30 Fragment Program Profile

The fp30 Fragment Program Profile is used to compile Cg source code to fragment programs for use by the NV_fragment_program OpenGL extension. The profile is called fp30 and it is invoked with the compiler option -profile fp30.

The key capabilities and limitations of the fp30 profile are

- full support for if/else
- no for and while loops, unless they can be unrolled by the compiler
  [This is not yet implemented.]
- support for fixed type (s1.10 fixed point)
- support for half type (s10e5 floating-point)
- support for flexible texture mapping
- support for screen-space derivative functions
- no support for indexible uniform arrays

It is recommended you use fixed, half, and float in that order. This ordering provides maximum performance. Reversing this ordering provides maximum precision. You are encouraged to use the fastest type that meets your needs for precision.

Binding Semantics

The set of binding semantics for varying input data of fp30 consists of COLOR0, COLOR1, and TEXCOORD0–TEXCOORD7. Additionally, the set of binding semantics COL0, COL1, TEX0–TEX7, and WPOS can also be used. These binding semantics map to NV_fragment_program input registers. The two sets act as aliases to each other. The profile also allows POSITION, FOG, PSIZE, HPOS, FOGC, PSIZ, BCOL0, BCOL1, and CLP0–CLP5 to be present as binding semantics on a member of a structure of a varying input data structure, provided the member with this binding semantics is not referenced. This allows Cg programs to have same structure specify the varying output of a vp30 profile program and the varying input of an fp30 profile program.

Table 14. fp30 Input Binding Semantics

<table>
<thead>
<tr>
<th>Binding Semantics Name</th>
<th>Corresponding Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLOR0, COL0</td>
<td>Input color0</td>
</tr>
<tr>
<td>COLOR1, COL1</td>
<td>Input color1</td>
</tr>
<tr>
<td>TEXCOORD0–TEXCOORD7,</td>
<td>Input texture coordinates</td>
</tr>
<tr>
<td>TEX0–TEX7</td>
<td></td>
</tr>
<tr>
<td>WPOS</td>
<td>Window Position Coordinates</td>
</tr>
</tbody>
</table>
Appendix B. Language Profiles

The set of binding semantics for varying output data of \texttt{fp30} consists of \texttt{COLOR} and \texttt{DEPTH}. Additionally, a set of binding semantics of \texttt{COL} and \texttt{DEPR} can be used. These binding semantics map to \texttt{NV\_fragment\_program} input registers. The two sets act as aliases to each other.

This profile supports the additional predefined uniform semantics \texttt{texunitn}, where \( n \) is a number in the range \([0..7]\) that can be used to tell the compiler to allocate the associated uniform sampler variable in texture unit \( n \).
The ARB Vertex Program Profile

The OpenGL ARB Vertex Program Profile is used to compile Cg source code to vertex programs compatible with version 1.0 of the GL_ARB_vertex program. The profile is called arbvp1 and it is invoked with the compiler option -profile arbvp1. It is the same as the vp20 profile except for the format of its output and its capability of accessing OpenGL state easily. ARB_vertex_program has the same capabilities as NV_vertex_program and DirectX 8 vertex shaders, so the limitations that this profile places on the Cg source code written by the programmer is the same as the NV_vertex_program profile. See The DirectX 8 Vertex Shader Profile for a full explanation of the data types, statements, and operators supported by this profile.

Accessing OpenGL State

The arbvp1 profile allows Cg programs to refer to the OpenGL state directly, unlike the vp20 profile. However, if you want to write Cg programs that are compatible with vp20 and dx8vs profiles, you should use the alternate mechanism of setting uniform variables with the necessary state using the Cg run time. The compiler relies on the feature of ARB vertex assembly programs that enables parts of the OpenGL state to be written automatically to program parameter registers as the state changes. The OpenGL driver handles this state-tracking feature. A special variable called glstate, defined as a structure, can be used to refer to every part of the OpenGL state that ARB vertex programs can reference. Following this paragraph are three lists of the glstate fields that can be accessed. The array indexes are shown as 0, but an array can be accessed using any positive integer that is less than the limit of the array. For example, to access the diffuse component of the second light use glstate.light[1].diffuse, assuming that GL_MAX_LIGHTS is at least 2.

These are the glstate fields of type float4x4 that can be accessed:

- glstate.matrix.modelview[0]
- glstate.matrix.mvp
- glstate.matrix.palet[0]
- glstate.matrix.inverse.modelview[0]
- glstate.matrix.inverse.mvp
- glstate.matrix.inverse.palet[0]
- glstate.matrix.transp[0]
- glstate.matrix.transp[0]
- glstate.matrix.invt[0]
- glstate.matrix.invt[0]
- glstate.matrix.projection
- glstate.matrix.texture[0]
- glstate.matrix.inver[0]
- glstate.matrix.inver[0]
- glstate.matrix.invtrans[0]
- glstate.matrix.invtrans[0]
These are the `glstate` fields of type `float4` that can be accessed:

```cpp
glstate.material.ambient glstate.material.diffuse
glstate.material.specular glstate.material.emission
glstate.material.shininess glstate.material.front.ambient
glstate.material.front.diffuse glstate.material.front.specular
glstate.material.front.emission glstate.material.front.shininess
glstate.material.back.ambient glstate.material.back.diffuse
glstate.material.back.specular glstate.material.back.emission
glstate.material.back.shininess glstate.light[0].ambient
glstate.light[0].diffuse glstate.light[0].specular
glstate.light[0].position glstate.light[0].attenuation
glstate.light[0].spot.direction glstate.light[0].half
glstate.lightmodel.ambient glstate.lightmodel.scenecolor
glstate.lightmodel.front.scenecolor glstate.lightmodel.back.scenecolor
glstate.lightprod[0].ambient glstate.lightprod[0].diffuse
glstate.lightprod[0].specular glstate.lightprod[0].front.ambient
glstate.lightprod[0].front.diffuse glstate.lightprod[0].front.specular
glstate.lightprod[0].back.ambient glstate.lightprod[0].back.diffuse
glstate.lightprod[0].back.specular glstate.texgen[0].eye.s
glstate.texgen[0].eye.t glstate.texgen[0].eye.r
glstate.texgen[0].eye.q glstate.texgen[0].object.s
glstate.texgen[0].object.t glstate.texgen[0].object.r
glstate.texgen[0].object.q glstate.fog.color
glstate.fog.params glstate.clip[0].plane
```

These are the `glstate` fields of type `float` that can be accessed:

```cpp
glstate.point.size glstate.point.attenuation
```

## Binding Semantics

The set of binding semantics for varying input data to `arbvp1` consists of `POSITION`, `BLENDWEIGHT`, `NORMAL`, `COLOR0`, `COLOR1`, `TESSFACTOR`, `PSIZE`, `BLENDINDICES`, and `TEXCOORD0–TEXCOORD7`. One can also use `TANGENT` and `BINORMAL` instead of `TEXCOORD6` and `TEXCOORD7`. Additionally, a set of binding semantics of `ATTR0–ATTR15` can be used. The mapping of these semantics to corresponding setting command is listed in table below.

### Table 15. `arbvp1` Input Binding Semantics

<table>
<thead>
<tr>
<th>Binding Semantics Name</th>
<th>Corresponding Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>POSITION</td>
<td>Input Vertex, through Vertex command</td>
</tr>
<tr>
<td>BLENDWEIGHT</td>
<td>Input vertex weight through WeightARB, VertexWeightEXT command</td>
</tr>
<tr>
<td>NORMAL</td>
<td>Input normal through Normal command</td>
</tr>
</tbody>
</table>
Cg Language Toolkit

<table>
<thead>
<tr>
<th>Binding Semantics Name</th>
<th>Corresponding Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIFFUSE, COLOR0</td>
<td>Input primary color through Color command</td>
</tr>
<tr>
<td>SPECULAR, COLOR1</td>
<td>Input secondary color through SecondaryColorEXT command.</td>
</tr>
<tr>
<td>FOGCOORD</td>
<td>Input fog coordinate through FogCoordEXT command</td>
</tr>
<tr>
<td>TEXCOORD0–TEXCOORD7</td>
<td>Input texture coordinates (texcoord0-texcoord7) through MultiTexCoord command</td>
</tr>
<tr>
<td>ATTR0–ATTR15</td>
<td>Generic Attribute 0-15 through VertexAttrib command</td>
</tr>
<tr>
<td>PSIZE, ATTR6</td>
<td>Generic Attribute 6</td>
</tr>
</tbody>
</table>

The set of binding semantics for varying output data of arbvp1 consists of POSITION, FOG, COLOR0, COLOR1, PSIZE, and TEXCOORD0–TEXCOORD7. Additionally, a set of binding semantics of HPOS, COL0, COL1, BCOL0, BCOL1, TEX0–TEX7, FOGC, and PSIZ can be used. These binding semantics map to ARB_vertex_program output registers. The two sets act as aliases to each other.

Table 16. arbvp1 Output Binding Semantics

<table>
<thead>
<tr>
<th>Binding Semantics Name</th>
<th>Corresponding Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>POSITION, HPOS</td>
<td>Output position</td>
</tr>
<tr>
<td>PSIZE, PSIZ</td>
<td>Output point size</td>
</tr>
<tr>
<td>FOG, FOGC</td>
<td>Output fog coordinate</td>
</tr>
<tr>
<td>COLOR0, COL0</td>
<td>Output primary color</td>
</tr>
<tr>
<td>COLOR1, COL1</td>
<td>Output secondary color</td>
</tr>
<tr>
<td>BCOL0</td>
<td>Output backface primary color</td>
</tr>
<tr>
<td>BCOL1</td>
<td>Output backface secondary color</td>
</tr>
<tr>
<td>TEXCOORD0–TEXCOORD7,</td>
<td>Output texture coordinates</td>
</tr>
<tr>
<td>TEX0–TEX7</td>
<td></td>
</tr>
</tbody>
</table>

The profile also allows WPOS to be present as binding semantics on a member of a structure of a varying output data structure, provided the member with this binding semantics is not referenced. This allows Cg programs to have the same structure specify the varying output of an arbvp1 profile program and the varying input of an fp30 profile program.

This profile supports the additional predefined uniform semantics Cn, where n is a number in the range [0..255] that can be used to tell the compiler to use a local parameter with index n. In the future, there may be additional mechanisms for using environment parameters and program constants.
Compatibility with the \texttt{vp20} Vertex Program Profile

Programs that work with the \texttt{vp20} profile are compatible with the \texttt{arbvp1} profile as long as they use the Cg run time to manage all uniform parameters, including OpenGL state. That is, \texttt{arbvp1} and \texttt{vp20} profiles can be used interchangeably without changing the Cg source code or the application program except for specifying a different profile. However, if any of the \texttt{glProgramParameterxxNV()} routines are used the application program needs to be changed to use the corresponding ARB functions.

Since there is no ARB function corresponding to \texttt{glTrackMatrixNV()}, an application using \texttt{glTrackMatrixNV()} and the \texttt{arbvp1} profile needs to be modified. One solution is to change the Cg source code to refer to the matrix using the \texttt{glstate} structure so that the matrix is automatically tracked by the OpenGL driver as part of its \texttt{GL_ARB_vertex} support. Another solution is for the application to use the Cg run-time routine \texttt{cgBindUniformStateMatrix()} to load the appropriate matrix or matrices when necessary.

Another potential incompatibility between the \texttt{arbvp1} and \texttt{vp20} profiles is the way that input varying semantics are handled. In the \texttt{vp20} profile, semantic names such as \texttt{POSITION} and \texttt{ATTR0} are aliases of each other the same way \texttt{NV_vertex_program} aliases \texttt{Vertex} and \texttt{Attribute 0} (see Table 10). In the \texttt{arbvp1} profile, the semantic names are not aliased because \texttt{ARB_vertex_program} allows the conventional attributes (such as vertex position) to be separate from the generic attributes (such as Attribute 0). For this reason it is important to follow the conventions given in Table 15 so that \texttt{arbvp1} programs work for all implementations of \texttt{ARB_vertex_program}. The \texttt{arbvp1} conventions are compatible with the \texttt{vp20} and \texttt{vp30} profiles.

Loading Constants

Applications that do not use the Cg run time are no longer required to load constant values into program parameters registers as indicated by the \texttt{#const} expressions in the Cg compiler output. The compiler produces output that causes the OpenGL driver to load them. However, uniform variables that have a default definition still require constant values to be loaded into the appropriate program parameter registers, as ARB vertex programs do not support this feature. Application programs either have to use the Cg run time, parse, and handle the \texttt{#default} commands, or have to avoid initializing uniform variables in the Cg source code.
Appendix C. Cg Compiler Options

This appendix describes the command-line options for the Cg compiler.

Compiler Options

Following are the command-line options for the Cg compiler, `cgc.exe`:

- `-profile prof`  Compile for the `prof` profile
- `-entry fname`  Specify the main function name as `fname`
- `-o fname`  Write the output to file `fname`
- `-quiet`  Suppress printing the header to `stdout`
- `-nocode`  Compile but do not generate any code
- `-nostdlib`  Do not include the `stdlib.h` header file before compilation
- `-longprogs`  Allow code generation that's longer than a profile's limit
- `-v`  Print the compiler's version to `stdout`
- `-h`  Print a short help message
Appendix D.
Color Plates

Color Illustrations

This appendix contains the color versions of the illustrations provided in the following shader examples throughout this book.

- The simple.cg shader
- Anisotropic Lighting
- Bump Dot3x2 Diffuse and Specular
- Bump reflection mapping
- Fresnel
- Grass
- Refraction
- Shadow Mapping
- Shadow Volume Extrusion
- Sine Wave
- Matrix Palette Skinning

Note: The figure numbers reflected in this Appendix match the actual figure numbers of the images in the text. A cross reference to the page of the original figure is provided so you can see the code associated with the illustration.
Example of *simple.cg* Shader

Figure 20. The *simple.cg* Shader  
(See page 46)

Example of Anisotropic Lighting

Figure 21. Example of Anisotropic Lighting Effect  
(See page 84)
Example of Dot3x2 Diffuse and Specular

Figure 22. Example of Bump Dot3x2 Diffuse and Specular
(See page 86)

Example of Bump Reflection Mapping

Figure 23. Example of Bump Reflection Mapping
(See page 90)
Example of Fresnel

Figure 24. Example of Fresnel
(See page 94)
Example of Grass

Figure 25. Example of Grass
(See page 96)
Example of Refraction

Figure 26. Example of Refraction
(See page 99)

Example of Shadow Mapping

Figure 27. Example of Shadow Mapping
(See page 102)
Example of Shadow Volume Extrusion

Figure 28. Example of Shadow Volume Extrusion
(See page 105)
Example of Sine Wave

Figure 29. Example of Sine Wave
(See page 108)

Example of Matrix Palette Skinning

Figure 30. Example of Matrix Palette Skinning
(See page 111)
To help you convert from using the `#pragma bind` syntax to using the binding semantics in this release of Cg, this appendix presents five tables that show the correspondence between binding semantics names and `#pragma bind` registers.

Table 17. Vertex Program Varying Inputs

<table>
<thead>
<tr>
<th>Binding Semantics Name</th>
<th>#pragma bind Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>POSITION (0)</td>
<td>ATTR0</td>
</tr>
<tr>
<td>BLENDWEIGHT (0)</td>
<td>ATTR1</td>
</tr>
<tr>
<td>NORMAL (0)</td>
<td>ATTR2</td>
</tr>
<tr>
<td>DIFFUSE (0)</td>
<td>ATTR3</td>
</tr>
<tr>
<td>SPECULAR (0)</td>
<td>ATTR4</td>
</tr>
<tr>
<td>TESSFACTOR (0)</td>
<td>ATTR5</td>
</tr>
<tr>
<td>PSIZE (0)</td>
<td>ATTR6</td>
</tr>
<tr>
<td>BLENDINDICES (0)</td>
<td>ATTR7</td>
</tr>
<tr>
<td>TEXCOORD (0-7)</td>
<td>ATTR8–15</td>
</tr>
<tr>
<td>TANGENT (0)</td>
<td>ATTR14</td>
</tr>
<tr>
<td>BINORMAL (0)</td>
<td>ATTR15</td>
</tr>
</tbody>
</table>

Table 18. Vertex Program Varying Outputs

<table>
<thead>
<tr>
<th>Binding Semantics Name</th>
<th>#pragma bind Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>POSITION</td>
<td>HPOS</td>
</tr>
<tr>
<td>PSIZE</td>
<td>PSIZ</td>
</tr>
<tr>
<td>FOG</td>
<td>ATTR2</td>
</tr>
<tr>
<td>COLOR (0–1)</td>
<td>COL (0–1)</td>
</tr>
<tr>
<td>TEXCOORD (0–7)</td>
<td>TEX (0–7)</td>
</tr>
<tr>
<td>N/A</td>
<td>CLP (0–5)</td>
</tr>
<tr>
<td>N/A</td>
<td>BCO (0–1)</td>
</tr>
</tbody>
</table>
Table 19. Fragment Program Varying Inputs

<table>
<thead>
<tr>
<th>Binding Semantics Name</th>
<th>#pragma bind Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLOR(0-1)</td>
<td>COL(0-1)</td>
</tr>
<tr>
<td>TEXCOORD(0-7)</td>
<td>TEX(0-7)</td>
</tr>
<tr>
<td>N/A</td>
<td>WPOS</td>
</tr>
</tbody>
</table>

Table 20. Fragment Program Varying Outputs

<table>
<thead>
<tr>
<th>Binding Semantics Name</th>
<th>#pragma bind Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLOR(0-1)</td>
<td>COL(0-1)</td>
</tr>
<tr>
<td>DEPTH</td>
<td>DEPR</td>
</tr>
</tbody>
</table>

Table 21. Uniform Parameters

<table>
<thead>
<tr>
<th>Binding Semantics Name</th>
<th>#pragma bind Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cn</td>
<td>c n</td>
</tr>
<tr>
<td>texunit$n$</td>
<td>texunit $n$</td>
</tr>
</tbody>
</table>

In Table 21, $n$ represents a positive integer.
2D texture lookup ................................................144
aliasing of semantics ..........................................129
animation of geometry ......................................96
anisotropic lighting effect ...................................84
ANSI C ................................................................115
APIs, Cg run-time library ................................29
application to a vertex program .....................19
ARB Vertex Program Profile ........................158
arbvp1 profile ................................................158
Arithmetic operator ........................................134
arithmetic operators ......................................14
arrays ................................................................12
Arrays and Subscripting ................................125
Assignment of an expression..........................124
bind iterator ....................................................41
binding semantic ............................................116
binding semantics, converting to ....................175
binding varying parameters .............................38
bindings ........................................................31
Blinn-Phong Bump-Mapping ..........................77
bool ..................................................................11
Boolean operators .........................................15, 134
built-in functions .........................................19
built-in vector data types ...............................11
bump reflection mapping ...............................90
C arithmetic operators ..................................14
C control constructs .....................................13
C for Graphics ...............................................xiv
C++ ..................................................................115
Cg arrays .......................................................12
Boolean operators ..................................... See
brief tutorial ..................................................45
compiler, using ..............................................29
data types .......................................................10
incompatibilities .............................................115
introduction .....................................................1
meaning of .....................................................1
Model of the GPU ..........................................2
programming model ......................................2
release notes ..................................................xvi
run-time library, using ...................................29
similar differences from ANSI-C .....................116
similar operations ..........................................115
standard library functions .............................19
Cg_Simple file ..............................................45
CgD3D ........................................................29, 40
cgGetBindValueType () .................................37
CgGL ..........................................................29
cgGLBindMatrix{cr}{fd} () ...........................37
cgGLBindUniform4{fd}[v] () ...........................37
command-line options, Cg compiler ..........163
comparison operators ................................15
Comparison operators ..................................134
compilation ..................................................29
compilation profile .......................................117
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>compiler-determined bindings</td>
<td>128</td>
</tr>
<tr>
<td>concrete type category</td>
<td>122</td>
</tr>
<tr>
<td>conditional operator</td>
<td>16, 134</td>
</tr>
<tr>
<td>connector definitions</td>
<td>48</td>
</tr>
<tr>
<td>connectors</td>
<td>19</td>
</tr>
<tr>
<td>av20</td>
<td>19</td>
</tr>
<tr>
<td>fragout</td>
<td>21</td>
</tr>
<tr>
<td>vf20</td>
<td>20</td>
</tr>
<tr>
<td>constant</td>
<td>122</td>
</tr>
<tr>
<td>construction operator</td>
<td>131</td>
</tr>
<tr>
<td>constructs</td>
<td>11</td>
</tr>
<tr>
<td>control constructs, C</td>
<td>13</td>
</tr>
<tr>
<td>C-preprocessor</td>
<td>128</td>
</tr>
<tr>
<td>creating a context</td>
<td>33</td>
</tr>
<tr>
<td>data types</td>
<td>10</td>
</tr>
<tr>
<td>Declaration</td>
<td>117</td>
</tr>
<tr>
<td>Definition</td>
<td>117</td>
</tr>
<tr>
<td>Direct3D</td>
<td></td>
</tr>
<tr>
<td>activating programs</td>
<td>42</td>
</tr>
<tr>
<td>managing parameters</td>
<td>41</td>
</tr>
<tr>
<td>managing programs</td>
<td>40</td>
</tr>
<tr>
<td>DirectX 8 Pixel Shader Profile</td>
<td></td>
</tr>
<tr>
<td>arithmetic operations</td>
<td>148</td>
</tr>
<tr>
<td>connectors</td>
<td>149</td>
</tr>
<tr>
<td>data types</td>
<td>140</td>
</tr>
<tr>
<td>examples</td>
<td>150</td>
</tr>
<tr>
<td>modifiers</td>
<td>150</td>
</tr>
<tr>
<td>other language constructs</td>
<td>150</td>
</tr>
<tr>
<td>texture shader operations</td>
<td>148</td>
</tr>
<tr>
<td>DirectX 8 Pixel Shader Profile</td>
<td>140</td>
</tr>
<tr>
<td>DirectX 8 Vertex Shader Profile</td>
<td>137</td>
</tr>
<tr>
<td>connectors</td>
<td>137, 152, 154, 156, 159</td>
</tr>
<tr>
<td>memory</td>
<td>138</td>
</tr>
<tr>
<td>using arrays</td>
<td>139</td>
</tr>
<tr>
<td>using conditionals</td>
<td>139</td>
</tr>
<tr>
<td>using constants</td>
<td>139</td>
</tr>
<tr>
<td>variable types</td>
<td>139</td>
</tr>
<tr>
<td>DirectX 8/ NV2X fragment program functions</td>
<td>142</td>
</tr>
<tr>
<td>DirectX 8 pixel profile</td>
<td></td>
</tr>
<tr>
<td>features not supported</td>
<td>xvii</td>
</tr>
<tr>
<td>dx8ps</td>
<td>140</td>
</tr>
<tr>
<td>dx8vs</td>
<td>137</td>
</tr>
<tr>
<td>expand(val)</td>
<td>142</td>
</tr>
<tr>
<td>expand_inv(val)</td>
<td>142</td>
</tr>
<tr>
<td>Expanded Operators</td>
<td>133</td>
</tr>
<tr>
<td>fixed-point data types</td>
<td>133</td>
</tr>
<tr>
<td>float</td>
<td>10</td>
</tr>
<tr>
<td>floating type category</td>
<td>122</td>
</tr>
<tr>
<td>foobar()</td>
<td>119</td>
</tr>
<tr>
<td>fragment programs</td>
<td>2</td>
</tr>
<tr>
<td>fragment-to-frame buffer connector</td>
<td>21</td>
</tr>
<tr>
<td>fresnel</td>
<td>94</td>
</tr>
<tr>
<td>fully-conformant profiles</td>
<td>126</td>
</tr>
<tr>
<td>function</td>
<td></td>
</tr>
<tr>
<td>arithmetic</td>
<td>14</td>
</tr>
<tr>
<td>calls</td>
<td>119</td>
</tr>
<tr>
<td>declarations</td>
<td>119</td>
</tr>
<tr>
<td>definitions</td>
<td>14</td>
</tr>
<tr>
<td>fragment program</td>
<td>142</td>
</tr>
<tr>
<td>geometric</td>
<td>25, 27</td>
</tr>
<tr>
<td>LoadShaderConstants()</td>
<td>42</td>
</tr>
<tr>
<td>matching rules</td>
<td>127</td>
</tr>
<tr>
<td>mathematical</td>
<td>22</td>
</tr>
<tr>
<td>multiplication</td>
<td>14</td>
</tr>
<tr>
<td>overloading</td>
<td>14, 127</td>
</tr>
<tr>
<td>Parameters</td>
<td>14</td>
</tr>
<tr>
<td>------------</td>
<td>----</td>
</tr>
<tr>
<td>Recursion</td>
<td>13</td>
</tr>
<tr>
<td>SetShaderActive()</td>
<td>42</td>
</tr>
<tr>
<td>Functions</td>
<td></td>
</tr>
<tr>
<td>Standard Library</td>
<td>19</td>
</tr>
<tr>
<td>GeForce3</td>
<td>xiii</td>
</tr>
<tr>
<td>Geometrical Functions</td>
<td>25</td>
</tr>
<tr>
<td>GL_ARB_vertex</td>
<td>158</td>
</tr>
<tr>
<td>glActiveClientTexture()</td>
<td>39</td>
</tr>
<tr>
<td>glActiveTexture()</td>
<td>39</td>
</tr>
<tr>
<td>glBegin()</td>
<td>38</td>
</tr>
<tr>
<td>glEnd()</td>
<td>38</td>
</tr>
<tr>
<td>Global Variables</td>
<td>127</td>
</tr>
<tr>
<td>GPU</td>
<td>2</td>
</tr>
<tr>
<td>Graphics, Evolution of</td>
<td>xiii</td>
</tr>
<tr>
<td>halfbias(val)</td>
<td>143</td>
</tr>
<tr>
<td>halfbias_inv(val)</td>
<td>143</td>
</tr>
<tr>
<td>Improved Water</td>
<td>55</td>
</tr>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>Varying</td>
<td>4</td>
</tr>
<tr>
<td>Int Type</td>
<td>11</td>
</tr>
<tr>
<td>Integral Type Category</td>
<td>122</td>
</tr>
<tr>
<td>Java</td>
<td>115</td>
</tr>
<tr>
<td>Kill(float4 texcoords)</td>
<td>143</td>
</tr>
<tr>
<td>Linked List</td>
<td>41</td>
</tr>
<tr>
<td>LoadCGProgramFromFile()</td>
<td>40</td>
</tr>
<tr>
<td>Main()</td>
<td>3</td>
</tr>
<tr>
<td>Mathematical Functions</td>
<td>22</td>
</tr>
<tr>
<td>Matrix Declarations</td>
<td>11</td>
</tr>
<tr>
<td>Matrix Palette Skinning</td>
<td>111</td>
</tr>
<tr>
<td>Melting Paint</td>
<td>59</td>
</tr>
<tr>
<td>Modifiable Function Parameter</td>
<td>13</td>
</tr>
<tr>
<td>MultiPaint</td>
<td>64</td>
</tr>
<tr>
<td>Namespaces</td>
<td>125</td>
</tr>
<tr>
<td>Non-Static Function</td>
<td>131</td>
</tr>
<tr>
<td>Non-Uniform Binding Semantics</td>
<td>129</td>
</tr>
<tr>
<td>Non-Uniform Variables</td>
<td>130</td>
</tr>
<tr>
<td>Numeric Type Category</td>
<td>122</td>
</tr>
<tr>
<td>NV30 Sample Shaders</td>
<td>51</td>
</tr>
<tr>
<td>Object</td>
<td>117</td>
</tr>
<tr>
<td>Offline Compilation</td>
<td>29</td>
</tr>
<tr>
<td>OffsetTex2D Fragment Program Functions</td>
<td>144</td>
</tr>
<tr>
<td>OpenGL ARB Vertex Program Profile</td>
<td>158</td>
</tr>
<tr>
<td>OpenGL Run-Time Library</td>
<td>35</td>
</tr>
<tr>
<td>Open-Profile Function</td>
<td>119</td>
</tr>
<tr>
<td>Operators</td>
<td></td>
</tr>
<tr>
<td>Conditional</td>
<td>16</td>
</tr>
<tr>
<td>Swizzle</td>
<td>15</td>
</tr>
<tr>
<td>Write-Mask</td>
<td>16</td>
</tr>
<tr>
<td>Operator Precedence</td>
<td>133</td>
</tr>
<tr>
<td>Operators</td>
<td></td>
</tr>
<tr>
<td>Boolean</td>
<td>15</td>
</tr>
<tr>
<td>Comparison</td>
<td>15</td>
</tr>
<tr>
<td>Operators</td>
<td></td>
</tr>
<tr>
<td>Arithmetic</td>
<td>134</td>
</tr>
<tr>
<td>Boolean</td>
<td>134</td>
</tr>
<tr>
<td>Comparison</td>
<td>134</td>
</tr>
<tr>
<td>Conditional</td>
<td>134</td>
</tr>
<tr>
<td>Typecast</td>
<td>135</td>
</tr>
<tr>
<td>Overloading of Functions</td>
<td>119</td>
</tr>
<tr>
<td>Passthrough(float4 texcoords)</td>
<td>143</td>
</tr>
<tr>
<td>Pixel Programs</td>
<td>2</td>
</tr>
<tr>
<td>Profile</td>
<td>117</td>
</tr>
<tr>
<td>Profile dx8ps</td>
<td>140</td>
</tr>
<tr>
<td>Profile dx8vs</td>
<td>137</td>
</tr>
<tr>
<td>Program</td>
<td>30</td>
</tr>
</tbody>
</table>
defining............................................................ 3
kinds of inputs................................................ 4
program, receive and return data...............131
qualifiers, type ..................................................123
Ray-Traced Refraction................................. 70
reflection vector................................................ 94
release notes ..................................................... xvi
RenderMan ...................................................... 115
Reserved Words.............................................. 135
run-time API
  capabilities..................................................... 30
  contexts ......................................................... 32
  key concepts.................................................. 30
  parameters..................................................... 31
  programs........................................................ 30
  setup................................................................... 32
run-time compilation........................................ See
  sample shaders ............................................... 51, 83
  sampler............................................................. 11
  scalar type category................................. 122
  semantics......................................................... 129
  SetShaderActive() ............................................ 42
shader examples
  anisotropic lighting effects .... 52, 55, 59, 64, 70, 77, 84
  bump dot3x2 diffuse and specular......... 86
  grass.............................................................. 96
  matrix palette skinning ......................... 111
  shadow mapping............................................. 102
  shadow volume extrusion.......................... 105
  Sine Wave....................................................... 108
shader profile
  DirectX 8 pixel shader ......................... 140
  DirectX 8 vertex shader ............. 137
  vp20 vertex shader ......................... 152, 154, 156
  shader, simple.cg example.................. 46
  shaders, samples ........................................... 51, 83
  shadow mapping ........................................... 102
  shadow mapping ........................................... 105
  simple shaders ............................................. 51, 83
  simple.cg
    basic transformations.............................. 49
    connector definitions.............................. 48
    passing arguments......................................... 48
    program listing........................................... 46
    Sine function...................................... 96, 108
    sinh(x)....................................................... 24
Skinning
  Improved ....................................................... 52
  smearing....................................................... 125
  standard C arithmetic operators ................. 133
  Stanford shading language...................... 115
  statements...................................................... 131
  statements and operators....................... 13
  stream-processing architectures............... 128
  structure elements....................................... 130
  structures...................................................... 12
  swizzle operator ........................................... 15, 131
  swizzling a constant scalar...................... 131
  tex1D(texobj1D tex)............................. 143
  tex2D(texobj2D tex)............................. 143
  tex3D(texobj3D tex)............................. 143
  texCUBE(texobjCUBE tex).......................... 143
  texRECT(texobjRECT tex).......................... 143
  texture unit.................................................. 44
  texture unit state......................................... 39
type cast.................................................................123
typedef cast operator .................................................. 135
Type-promotion Rules................................................... 124
Types................................................................. 120
uclamp (val) .......................................................... 143
uclamp_inv (val) ...................................................... 143
uniform inputs ......................................................... 4
Uniform Modifier.......................................................... 118
uniform parameters ...................................................... 36
uninitialized variable.................................................... 127
varying inputs........................................................... 4
vector
    constructor.......................................................... 15
vector data types
built-in........................................................................... 11
vector operators ........................................................... 131
vertex buffer format ..................................................... 42
vertex color ............................................................... 48
vertex position ........................................................... 48
vertex programs .......................................................... 2
vertex-to-fragment connector......................................... 20, 48
vp20 Vertex Shader Profile........................................... 152, 154, 156
web site ......................................................................... xvii
workspace screen .......................................................... 45
workspace, loading......................................................... 45
write-mask operator ...................................................... 132
write-mask operator ...................................................... 16