User's Manual

A Developer's Guide to Programmable Graphics

Release 1.0
December 2002

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.

翻訳 監修　シリコンスタジオ株式会社

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com

目次

図

表

序文

コンピュータ グラフィックスは、ハードウェアの面でも、またゲーム、対話型アプリケーション、アニメーションなどの画像品質やオーサリング プロセスの面でも大きな転換期にあります。グラフィックス ハードウェアは、数十万ドルもの大型コンピュータ グラフィックス ワークステーションから、単一チップのグラフィックス プロセッシング ユニット（GPU）へと進化してきました。GPU のパフォーマンスと機能は、従来のワークステーションと同等にまで成長し、今ではそれを凌駕しています。現在の GPU で単一フレーム内に提供される処理能力は、以前にオフラインでレンダリングされるアニメーション フレームに費やされていた膨大な計算量に匹敵します。実際、Apple Macintosh に GeForce3 が登場したとき、リアルタイムで対話的に稼働する Pixar 社の Luxo, Jr. のデモが行われましたが、これには説得力がありました。2001 年の SIGGRAPH カンファレンスでは、最近の映画の対話型バージョンである Square Studios 社の「ファイナル ファンタジー」が、同じく GeForce3 上でリアルタイムで稼働するデモが実演されました。

計算能力に関するこれらの功績は驚くべきものですが、他にもまだまだあります。今日の GPU は非常にめまぐるしく進化しています。一般に、製品はわずか 6 か月間で世代交代し、新しい世代の製品が登場するたびにパフォーマンスは 2 倍になっています。グラフィックス プロセッサのパフォーマンスは、マイクロプロセッサの約 3 倍の比率で向上しています。なんとムーアの法則の 3 倍なのです。パフォーマンスの向上だけでなく、新しいハードウェア機能と、それをサポートする新しいアプリケーション プログラミング インタフェース（API）が毎年のように登場しています。このように目もくらむような進化のペースに追いつくのは困難ですが、それでも開発者は追いついていく必要があります。

開発者とユーザは、より高いレンダリング品質と、より現実的なイメージおよび体験を要求しています。ユーザは、細かいことは気にかけません。ただ単に、映画や SFX、そしてアニメーションに少しでも近いゲームなどの対話型アプリケーションを望んでいるだけなのです。開発者は、常により高い能力を求めており、今日および将来の高機能な GPU の制御に関する柔軟性の向上がそれに伴うことを期待しています。API は、GPU の急速な革新に追いついておらず、また追いつくことができません。API および基礎となるテクノロジが変更されると、プログラマ、アーティストおよびソフトウェア パブリッシャは、ハードウェアとソフトウェア プラットフォームの変更と激動に適応すべく奮闘します。

必要なのは、GPU との対話のために抽象レベルを上げることです。開発者がハードウェアに近い部分にばかり目を向けているかぎり、ハードウェアと API の間断ない更新と改良は苦痛以外のなにものでもありません。この問題は、プログラマブル GPU の出現によりさらに悪化しました。古い GPU では、制御可能または

構成可能なレンダリング パスがごく少数でしたが、最新のテクノロジは高度にプログラマブルであり、その傾向はますます強まっています。今では、GPU で実行される短い頂点プログラムとフラグメント プログラムを記述することが可能です。これは高いスキルを必要とし、短いプログラムでのみ実現可能です。

GPU ハードウェアが進化し、数百から数千以上の命令からなるプログラムを作成できるようになれば、アセンブリ コーディングはもはや現実的ではなくなります。各レンダリング ステートを、データのビットごと、バイトごと、そしてワードごとにプログラミングし、低レベルなアセンブリ言語を通じて制御するのではなく、高級言語を使用して、より単純な形式でアイデアを表現しなければなりません。

こうした状況が、Cg（C for Graphics）を必要不可欠なものにしています。プロセッサの特定機能を活かしつつ、その一方でより高度な抽象化を実現するために C 言語が生まれたように、Cg では GPU に対して同じ抽象化を実現できます。Cg によって、プログラマによるプログラミング方法が変わります。Cg では、ハードウェア実装の詳細ではなく、アイデア、コンセプトおよび生成しようとするエフェクトに焦点を当てています。Cg は、ハードウェア実装に固有ではなく機能的であるため、プログラムを特定のハードウェアから分離します。また、Cg は任意のプラットフォーム、オペレーティング システムおよびグラフィックス ハードウェアで実行時にコンパイル可能なため、Cg プログラムは真の意味で移植可能です。最後に、またおそらく最も重要なことは、Cg プログラムが将来にわたっても使用でき、将来の製品上でも適切に稼働するよう適応できることです。コンパイラは、オリジナルの Cg プログラムが作成されたときにはおそらく存在すらしていなかった新しいターゲット GPU に対して、直接的に最適化できるからです。

本書は、Cg を紹介すること、また、プログラマが Cg での開発を開始する際の実用的なハンドブックとなることを意図されています。本書には、言語の説明、標準ライブラリおよびランタイム ライブラリのリファレンスと、役に立つサンプルが満載されています。本書の目的は、新たなユーザが概要およびツールとして利用できること、また、開発者が熟練したときにはリファレンスやリソースとして利用できるようにすることにあります。

Cg の世界へようこそ！

David Kirk
Chief Scientist
NVIDIA Corporation

はじめに

本書の目的は、グラフィックス プログラミング用の新しい高級言語である Cg を紹介することにあります。そのために、本書は次の各章で編成しました。

- 1 ページ、「Cg 言語の概要」
  Cg の最新リリースを簡単に紹介し、Cg を使い始める際に知っておくべきことをすべて記載しています。
- 19 ページ、「Cg 標準ライブラリ関数」
  プログラム開発時間の短縮に役立つ標準ライブラリ関数のリストです。
- 29 ページ、「Cg ランタイム ライブラリの使用」
  Cg ランタイム API の概要です。この API を使用すると、Cg プログラムを簡単にコンパイルし、アプリケーションから Cg プログラムにデータを渡すことができます。
- 89 ページ、「簡単なチュートリアル」
  Cg の実験開始に使用できる単純な Cg プログラムと、Microsoft Visual Studio ワークスペース（どちらも付属の CD で提供されています）についての説明です。
- 97 ページ、「拡張プロファイルのサンプル シェーダ」
  完全なソース コードを併記したサンプルの NV30 シェーダのリストです。
- 133 ページ、「基本プロファイルのサンプル シェーダ」
  完全なソース コードを併記したサンプルの NV2X シェーダのリストです。
- 165 ページ、付録 A「Cg 言語仕様」
  Cg の正式な言語仕様です。
- 197 ページ、付録 B「言語プロファイル」
  現在サポートされている言語プロファイルの機能と制限事項について説明しています。DirectX 8 頂点、DirectX 8 ピクセル、OpenGL ARB 頂点、NV2X OpenGL 頂点、NV30 OpenGL 頂点および NV30 OpenGL フラグメントの各プロファイルがあります。
- 257 ページ、付録 C「Cg のパフォーマンスを向上させる 9 つのステップ」
  Cg コードを最大限に活用するための戦略です。
- 265 ページ、付録 D「Cg コンパイラ オプション」
  Cg コンパイラで使用できる各種コマンドライン オプションのリストです。
- 267 ページ、付録 E「カラー図」
  サンプル シェーダ イメージのカラー バージョンです。

- Cg Developer's CD
  本書に添付されている CD には、Cg リリースのすべてが収録されています。この CD を使用して、Cg をすぐに使い始めることができます。CD 上の readme.txt ファイルでは、リリースの内容を詳細に説明しています。

1 ページの「Cg 言語の概要」を読み、次に 89 ページの「簡単なチュートリアル」を完了することにより、Cg をすぐに使い始めることができます。Cg 言語の基本を理解した後で、97 ページの「拡張プロファイルのサンプル シェーダ」および 133 ページの「基本プロファイルのサンプル シェーダ」をベースとして使用し、独自のエフェクトを構築できます。

リリース ノート

今回のリリースは、DirectX 8、DirectX 9 および OpenGL を含む 14 種類の対象プロファイルをサポートする、Cg の最初の完全リリースです。これは大きな成果であり、これにより開発者は、様々なハードウェア プラットフォームおよびグラフィックス API に対して興味深い Cg プログラムを作成できます。バグや問題点、および NVIDIA へのフィードバックは、cgsupport@nvidia.com 宛てに電子メールでご報告ください。報告された問題については、早急に対処します。

Cg コンパイラは、現在次のプロファイルをサポートしています。

- VertexShader VS 1.1 を対象とした、DirectX 8 および DirectX 9 用の **vs_1_1**
- VertexShader VS 2.0 および Extended VS 2.0 を対象とした、DirectX 9 用の **vs_2_0** および **vs_2_x**（まとめて **vs_2_*** と呼ぶ）
- PixelShader PS 1.1、1.2 および 1.3 を対象とした、DirectX 8 および DirectX 9 用の **ps_1_1**、**ps_1_2** および **ps_1_3**（まとめて **ps_1_*** と呼ぶ）
- PixelShader PS 2.0 および Extended PS 2.0 を対象とした、DirectX 9 用の **ps_2_0** および **ps_2_x**（まとめて **ps_2_*** と呼ぶ）
- ARB_Vertex_Program 1.0 を対象とした、OpenGL 用の **arbvp1**
- ARB_Fragment_Program 1.0 を対象とした、OpenGL 用の **arbfp1**
- NV_Vertex_program 1.0 および NV_Vertex_program 2.0 を対象とした、OpenGL 用の **vp20** および **vp30**
- NV_Fragment_Program 1.0 を対象とした、OpenGL 用の **fp30**
- NV_register_combiners and NV_Texture_shader を対象とした、OpenGL 用の **fp20**

Cg には、次のランタイム ライブラリが含まれます。

- パラメータ管理とプログラムのロードを行うコア ランタイム ライブラリ
- DirectX 8 に基づくアプリケーション用のランタイム ライブラリ
- DirectX 9 に基づくアプリケーション用のランタイム ライブラリ
- OpenGL に基づくアプリケーション用のランタイム ライブラリ

開発者からのフィードバックに応えて、前回のベータ リリースから Cg ランタイム ライブラリを更新しました。より直観的であり、簡潔かつ一貫したインタフェースを備えた新しい API によって、新しい機能が公開されています。以前のランタイム API からの移行を支援するために、このパッケージにはランタイム移行に必要なガイドが含まれています。

既知の問題および実装されていない機能

次に挙げる問題と機能は、将来のリリースで対処されます。

プリプロセッサでまだサポートされていない機能

および ## マクロ演算子はサポートされていません。

言語実装の既知の問題

- エラー レポート
  認識しておく必要のある問題を次に示します。
  - 標準ライブラリ関数を使用した場合に、レポートされる行番号がソースコードの行と一致しません。
  - エラーは、プログラムで出現する順序ではレポートされない場合があります。
  - 型のない定数の場合、定数が範囲外であってもエラーはレポートされません。
  - 一部のエラー メッセージの表現は改善される予定です。

  以上の問題に対する作業は、現在進行中です。
  プログラムではキーワードを識別子として使用しないよう注意する必要もあります。（**in** または **out** を変数名として使用することは、よく起こしやすい誤りです。）このような場合に発行されるエラーは、理解しにくいことがあります。
  書き込まれていない **out** パラメータには未定義の値が格納されるため、アプリケーションのバグの原因となる可能性があります。この場合、Cg コンパイラが警告を発行します。
  条件式（**?:**）と論理式（**&&** および **||**）の副作用は、条件にかかわらず常に評価されます。現在、警告は必ずしも発行されないため、これらのケースに注意する必要があります。

標準ライブラリ関数のオーバーロードは許可されており、コンパイラは警告を発行しません。このため、ユーザ定義関数と標準ライブラリ関数の名前の競合に注意する必要があります。

- staticとして定義されていないかぎり、**const** グローバルでは定数の畳み込みが行われず、コードの効率を低下させることがあります。ただし、static の定数グローバルおよびローカルでは、定数の畳み込みが行われます。効率の悪いコードを回避する 1 つの方法は、可能なかぎり静的な定数変数を使用することです。
- プログラムのエントリ関数に対する **inout** パラメータはサポートされていません。エントリ以外の関数に対する **inout** パラメータは正常に動作します。
- 1つの関数で使用できる **return** ステートメントは1つのみです。到達できないコードがある場合には、エラーが発生します。**if** および **for** ブロック内での **return** ステートメントは、サポートされていません。
- varying 型の配列データでは、セマンティクスはサポートされていません。この問題は、次のリリースで修正される予定です。
- 条件として **==** または **!=** を使用するループはアンロールされません。**<**、**>**、**<=** および **>=** を使用するループはアンロールできます。したがって、
  ```
  while (foo) { ... }
  ```
  のようなループはアンロールされず、動的分岐をサポートしないプロファイルでは動作しません。単一帰納変数を持つループのみがアンロールされます。
- すべての行列は行優先のみです。列優先の行列は現在サポートされていません。
- コンパイラでは、uniform型変数あたり最大1つのバインディング セマンティクスがサポートされます。異なるプロファイル用の複数のバインディング セマンティクスを、1 つの uniform 型変数で使用することはできません。
- **+** や ***** などの 2 項演算子は、行列型では現在サポートされていません。

ランタイムにおける既知の問題

- APIのエントリ ポイント **cgIsParameterReferenced()** では、パラメータが最終的なコンパイル出力で参照されない可能性がある場合でも **true** を返します。
- Cg プログラム内で宣言され、実際には参照されないパラメータを設定すると、ランタイムがエラーを返すことがあります。

ARB フラグメント プログラム プロファイルにおける既知の問題

- OpenGL ステート構造体への Cg の ARB 頂点プログラム プロファイルと同様のアクセスは、まだサポートされていません。この制限は、アプリケーションで OpenGL ステートに明示的な uniform 型パラメータを設定することによって、非効率ながら回避することができます。
- OpenGL で使用できる ARB フラグメント プログラムの実装は非常に限られているため、このプロファイルはまだ開発段階にあります。

ps_2_* プロファイルにおける既知の問題

- 複数のカラー出力への書出しは、まだサポートされていません。
- 依存読取り制限チェックは改善が必要です。

arbfp1、fp30 および ps_2_* プロファイルにおける既知の問題

- 配列要素への条件付き代入および **`if/else`** ブロックでの配列要素への代入は、動作しない場合があります。
- **`%`** 演算子はサポートされていません。整数の除算は完全にはエミュレートされず、浮動小数点除算として実装されています。

fp20 プロファイルにおける既知の問題

このプロファイルでは、FOG varying 型入力セマンティクスがまだサポートされていません。

ps_1_* および fp20 プロファイルにおける既知の問題

- **`fp20`**および**`ps_1_*`**プロファイルに対するハードウェアのサポートは非常に制限されており柔軟性がないため、これらのプロファイルでは、単純に見える Cg プログラムをコンパイルできないことがあります。これらのプロファイルにおける制限の詳細は、次の URL のプレゼンテーションをお読みください。
  http://developer.nvidia.com/view.asp?IO=gdc2001_texture_shaders および
  http://developer.nvidia.com/view.asp?IO=gdc2001_programmable_texture
  詳細は、OpenGL の NV_register_combiners 拡張および NV_texture_shader 拡張、または DirectX ピクセル シェーダ PS 1.1、1.2 および 1.3 の仕様をお読みください。

DirectX 8 頂点プロファイルにおける既知の問題

このプロファイルでは、標準ライブラリの **`noise()`** 関数族がまだサポートされていません。

コンパイラ使用上のその他の注意事項

- すべての varying 型データに対してバインディング セマンティクスを指定する必要があります。この指定をコンパイラに任せ、指定されていないバインディング セマンティクスで varying 型データにリソースを割り当てると、予期しない結果が発生することがあります。混乱の要因の 1 つは、同じバインディング セマンティクスを持つ複数の varying 型入力変数の使用が認められていることです。このため、コンパイラはプログラム作成者の意図どおりにリソースを割り当てるとは限りません。
- 出力への部分的な書込みは、それをサポートするプロファイルでは可能ですが推奨はされません。
- プログラムでは、書き込まれていない **out** パラメータの使用を避けてください。

オンライン アップデート

変更、追加または訂正は、すべて NVIDIA Cg の次の Web サイトに掲載されます。

http://developer.nvidia.com/Cg

Cg 言語に対する最新の変更と追加は、このサイトを頻繁に参照して確認してください。リリースにバグを発見した場合の報告方法に関する情報も、このサイトに記載されています。

Cg 言語の概要

従来、グラフィックス ハードウェアは、非常に低レベルの言語でプログラミングされてきました。固定機能パイプラインは、テクスチャの結合モードなどのステートを設定することによって設定されました。最近では、プログラマがアセンブリ言語レベルのプログラミング インタフェースを使用することにより、プログラマブル パイプラインを設定しました。理論的には、これらの低レベル プログラミング インタフェースによって、高い柔軟性が提供されました。実際には、低レベル プログラミング インタフェースは使いにくく、ハードウェアの効果的な使用に対する重大な障壁となっていました。

過去の低レベル言語ではなく高級言語を使用すると、いくつかの利点があります。

- 高級言語では、シェーダ開発時の改良および実行サイクルがスピードアップします。シェーダの最良のテスト方法は「うまく表示できたか ?」を確認することです。したがって、高品質なエフェクトを迅速に開発するためには、素早くシェーダのプロトタイプを作成し、また変更できることが不可欠です。
- コンパイラは、コードを自動的に最適化し、レジスタ割当てなどの煩雑で誤りやすい低レベルのタスクを実行します。
- 高級言語で作成されたシェーディング コードの方がはるかに読みやすく、簡単に理解できます。以前に作成されたシェーダを変更することにより、新しいシェーダを簡単に作成することもできます。優れたアーティストやプログラマによって作成されたシェーダから学習するよりもよい方法があるでしょうか。
- 高級言語で作成されたシェーダは、アセンブリ コードで作成されたシェーダよりも広い範囲のハードウェア プラットフォームに移植できます。

この章では、GPU のプログラミングに適応する新しい高級言語である Cg（C for Graphics）を紹介します。Cg には前述のすべての利点があり、プログラマは、最終的にGPU本来の能力と、GPUプログラミングを簡単にする言語を結合できます。

Cg 言語

Cg は C をベースにしていますが、拡張と変更を加えて、高度に最適化された GPU コードにコンパイルされるプログラムを簡単に作成できるようにしています。Cg のコードは C のコードとよく似ており、宣言、関数呼出しおよびほとんどのデータ型に C と同じ構文を使用しています。

Cg 言語について詳細に説明する前に、Cg と C の間に差異がいくつか存在する理由を説明します。基本的に、これらの差異は、GPU と CPU のプログラミング モデルの相違に帰結します。

Cg の GPU 用プログラミング モデル

CPU には、通常 1 つのプログラマブル プロセッサしかありません。対照的に、GPU には、少なくとも頂点プロセッサとフラグメント プロセッサの 2 つのプログラマブル プロセッサがあり、加えてその他の非プログラマブルなハードウェア ユニットがあります。プロセッサ、非プログラマブルなユニットおよびアプリケーションは、すべてデータ フローによってリンクされます。図 1 に、Cg による GPU のモデルを示します。

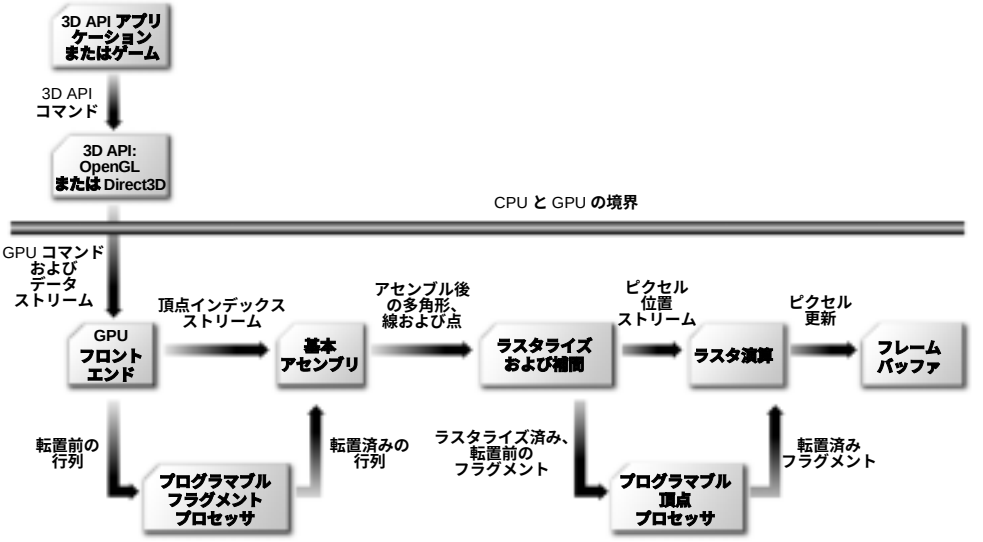


図 1　Cg の GPU のモデル

Cg 言語では、頂点プロセッサとフラグメント プロセッサの両方のプログラムを作成できます。これらのプログラムを、それぞれ頂点プログラムおよびフラグメント プログラムと呼びます。（フラグメント プログラムは、ピクセル プログラムまたはピクセル シェーダとも呼ばれます。このマニュアルでは、これらの用語を同じ意味で使用します。）Cg コードは、実行時にオンデマンドで、または事前に、GPU アセンブリ コードにコンパイルできます。

Cg によって、Cg フラグメント プログラムを手書きの頂点プログラムと簡単に結合できます。また、Cg フラグメント プログラムを非プログラマブルな OpenGL または DirectX 頂点パイプラインと結合することもできます。同様に、Cg 頂点プログラムを、手書きのフラグメント プログラム、または非プログラマブルな OpenGL または DirectX フラグメント パイプラインと結合することもできます。

Cg 言語プロファイル

すべての CPU では、基本的に同じ基本機能セットがサポートされるため、C 言語はすべての CPU に対してこの同じセットをサポートします。一方、GPU のプログラマビリティは、これと同じレベルの汎用性にまだ到達していません。たとえば、現世代のプログラマブル頂点プロセッサは、プログラマブル フラグメント プロセッサよりも広い範囲の機能をサポートします。Cg は、言語プロファイルの概念を導入することによって、この問題に対処しています。Cg プロファイルでは、特定のハードウェア プラットフォームまたは API でサポートされる完全な Cg 言語のサブセットを定義します。Cg コンパイラの現在のリリースでは、次の各プロファイルをサポートしています。

- DirectX 9 頂点シェーダ
  ランタイム プロファイル： **CG_PROFILE_VS_2_X**
  CG_PROFILE_VS_2_0
  コンパイラ オプション： **-profile vs_2_x**
  -profile vs_2_0
- DirectX 9 ピクセル シェーダ
  ランタイム プロファイル： **CG_PROFILE_PS_2_X**
  CG_PROFILE_PS_2_0
  コンパイラ オプション： **-profile ps_2_x**
  -profile ps_2_0
- OpenGL ARB 頂点プログラム
  ランタイム プロファイル： **CG_PROFILE_ARBVP1**
  コンパイラ オプション： **-profile arbvp1**
- OpenGL ARB フラグメント プログラム
  ランタイム プロファイル： **CG_PROFILE_ARBFP1**
  コンパイラ オプション： **-profile arbfp1**
- OpenGL NV30 頂点プログラム
  ランタイム プロファイル： **CG_PROFILE_VP30**
  コンパイラ オプション： **-profile vp30**
- OpenGL NV30 フラグメント プログラム
  ランタイム プロファイル： **CG_PROFILE_FP30**
  コンパイラ オプション： **-profile fp30**

- DirectX 8 頂点シェーダ
  ランタイム プロファイル： **CG_PROFILE_VS_1_1**
  コンパイラ オプション： **-profile vs_1_1**
- DirectX 8 ピクセル シェーダ
  ランタイム プロファイル： **CG_PROFILE_PS_1_3**
  CG_PROFILE_PS_1_2
  CG_PROFILE_PS_1_1
  コンパイラ オプション： **-profile ps_1_3**
  -profile ps_1_2
  -profile ps_1_1
- OpenGL NV2X 頂点プログラム
  ランタイム プロファイル： **CG_PROFILE_VP20**
  コンパイラ オプション： **-profile vp20**
- OpenGL NV2X フラグメント プログラム
  ランタイム プロファイル： **CG_PROFILE_FP20**
  コンパイラ オプション： **-profile fp20**

DirectX 9 プロファイル（**vs_2_x** および **ps_2_x**）、OpenGL ARB プロファイル（**arbfp1** および **arbvp1**）および NV30 OpenGL プロファイル（**fp30** および **vp30**）は一般に、長く複雑なプログラムをサポートし、より豊富な機能を開発者に提供します。これらのプロファイルを拡張プロファイルと呼びます。

DirectX 8 プロファイル（**vs_1_1** および **ps_1_3**）および NV2X OpenGL プロファイル（**fp20** および **vp20**）は、特にフラグメント プログラムでプログラムの長さと使用可能な機能に制限があります。これらのプロファイルを基本プロファイルと呼びます。

これらのプロファイルおよび関連プロファイルの詳細な説明は、197 ページの「言語プロファイル」を参照してください。

Cg でのプログラムの宣言

CPU コードは、一般に C の **main()** で指定されている 1 つのプログラムから構成されます。対照的に、Cg プログラムには任意の名前を使用できます。プログラムは、次の構文を使用して定義します。

```
<return-type> <program-name>(<parameters>)[: <semantic-name>]
{ /* ... */ }
```

プログラムの入力と出力

GPU のプログラマブル プロセッサは、データのストリームを操作します。頂点プロセッサは頂点のストリームを操作し、フラグメント プロセッサはフラグメントのストリームを操作します。

CPU 上では、プログラマは main プログラムを 1 回のみ実行されるものとみなすことができます。対照的に、GPU 上では、プログラムは繰り返し実行されます。プログラムはストリームのデータ要素ごとに 1 回ずつ実行されます。頂点プログラムは頂点ごとに 1 回ずつ実行され、フラグメント プログラムはフラグメントごとに 1 回ずつ実行されます。

Cg 言語は、C にいくつかの機能を追加して、このストリームベース プログラミング モデルをサポートしています。この機能に直接対応する C の機能はないため、Cg を初めて使用するプログラマは、この機能を理解するのに時間がかかることがあります。ただし、このマニュアルで後で示すサンプル プログラムでは、Cg プログラムでのこれらの機能が真に使いやすいことが示されています。

2 種類のプログラム入力

Cg プログラムでは、2 つの異なった種類の入力を使用できます。

- varying（可変）型入力は、入力データ ストリームの各要素で指定されるデータに使用されます。たとえば、頂点プログラムへの varying 型入力は、頂点配列で指定されている頂点ごとの値です。フラグメント プログラムでは、varying 型入力はテクスチャ座標などの補間値です。
- uniform（均一）型入力は、入力データのメイン ストリームとは別に指定されている値に使用され、ストリーム要素ごとに変化しません。たとえば、頂点プログラムでは、一般に変換行列を uniform 型入力として要求します。uniform 型入力は、グラフィックスのステートとして考えられることがあります。

頂点プログラムへの varying 型入力

頂点プログラムでは通常、頂点ごとに数種類（varying 型）の入力を使用します。たとえば、プログラムでは、アプリケーションで各頂点の次のような varying 型入力を、通常は頂点の配列として指定する必要があります。

- モデル空間位置座標
- モデル空間法線ベクトル
- テクスチャ座標

固定関数のグラフィックス パイプラインの場合、予想される頂点ごとの入力セットは小さく、事前定義されています。このような事前定義済みの入力セットは , グラフィックス API を介してアプリケーションに公開されます。たとえば、OpenGL 1.4 では複数の法線ベクトルから成る 1 つの頂点配列を指定できます。

プログラマブルなグラフィックス パイプラインでは、事前定義された少数の入力セットだけではなくなります。開発者が、頂点ごとの屈折率を使用する頂点プログラムを記述しても、アプリケーションが各頂点にこの値を提供するかぎり、何の問題もありません。

Cg には、事前定義された名前のセットの形式でこれらの頂点ごとの入力を指定する柔軟なメカニズムが用意されています。各プログラム入力は、このセットに属する名前にバインドされる必要があります。次の構造体では、頂点プログラム

定義は **POSITION**、**NORMAL**、**TANGENT** および **TEXCOORD3** という事前定義済みの名前に、パラメータをバインドしています。アプリケーションでは、これらの事前定義済みの名前に対応する頂点配列データを提供する必要があります。

```
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.
  /* ... */
}
```

この事前定義済みの名前を、バインディング セマンティクスと呼びます。次のバインディング セマンティクスのセットは、Cg のすべての頂点プログラム プロファイルでサポートされます。一部の Cg プロファイルでは、この他のバインディング セマンティクスもサポートされます。

POSITION	**BLENDWEIGHT**
NORMAL	**TANGENT**
BINORMAL	**PSIZE**
BLENDINDICES	**TEXCOORD0–TEXCOORD7**

バインディング セマンティクス **POSITION0** は、バインディング セマンティクス **POSITION** と等価であり、他のバインディング セマンティクスも同様に等価なものがあります。

OpenGL の Cg プロファイルでは、バインディング セマンティクスによって暗黙的に、varying 型入力から特定ハードウェア レジスタへのマッピングが指定されます。一方、DirectX ベースの Cg プロファイルでは、そのような暗黙的なマッピングは行われません。

バインディング セマンティクスは、**struct** 要素の中ではなく、プログラムのパラメータで直接指定できます。したがって、頂点プログラム定義は次のように書くこともできます。

```
outdata foo(float3 myPosition       : POSITION,
            float3 myNormal         : NORMAL,
            float3 myTangent        : TANGENT,
            float  refractive_index : TEXCOORD3) {
  /* ... */
  // Within the program, the parameters are referred to by
  // their variable names: "myPosition", "myNormal",
  // "myTangent", and "refractive_index".
  /* ... */
}
```

頂点プログラムへの、および頂点プログラムからの varying 型出力

頂点プログラムの出力は、ラスタライザを通過し、フラグメント プログラムで varying 型入力として使用されます。頂点プログラムとフラグメント プログラムが相互に機能するためには、両者間で受け渡しされるデータが一致する必要があります。

アプリケーションと頂点プログラムの間におけるデータ フローと同様、Cg ではバインディング セマンティクスを使用して、頂点プログラムとフラグメント プログラムの間でのデータ フローを指定します。

次の例では、頂点プログラムの出力のためのバインディング セマンティクスの使用方法を示します。

```
// Vertex program
struct myvf {
  float4 pout         : POSITION; // Used for rasterization
  float4 diffusecolor : COLOR0;
  float4 uv0          : TEXCOORD0;
  float4 uv1          : TEXCOORD1;
};
myvf foo(/* ... */) {
  myvf outstuff;
  /* ... */
  return outstuff;
}
```

次の例では、同じデータをフラグメント プログラムへの入力として使用する方法を示します。

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

バインディング セマンティクス **POSITION**、**PSIZE**、**FOG**、**COLOR0–COLOR1** および **TEXCOORD0–TEXCOORD7** は、Cg のすべての頂点プロファイルで、頂点プログラムからの出力用に使用できます。

頂点プログラムでは、バインディング セマンティクス **POSITION** を使用するベクトル出力を必ず宣言し、設定する必要があります。この値はラスタライズに必要です。

頂点プログラムとフラグメント プログラムとの相互運用性を確保するために、頂点プログラムの出力とフラグメント プログラムの入力には同じ **struct** を使用する必要があります。次に例を示します。

```
struct myvert2frag {
  float4 pos : POSITION;
  float4 uv0 : TEXCOORD0;
  float4 uv1 : TEXCOORD1;
};

// Vertex program
myvert2frag vertmain(...) {
  myvert2frag outdata;
  /* ... */
  return outdata;
}

// Fragment program
void fragmain(myvert2frag indata ) {
  float4 tcoord = indata.uv0;
  /* ... */
}
```

一部の頂点出力セマンティクスに関連付けられている値は、ラスタライザ用のものであり、ラスタライザにより使用されることに注意してください。これらの値は、入力 **struct** に出現する場合でも、フラグメント プログラムでは実際に使用できません。たとえば、**POSITION** フラグメント セマンティクスに関連付けられている **indata.pos** 値は、**fragmain** シェーダで読み取れません。

フラグメント プログラムからの varying 型出力

フラグメント プログラムの出力では、常にバインディング セマンティクスが必要です。フラグメント プログラムでは、**COLOR** セマンティクスを使用するベクトル出力を必ず宣言し、設定する必要があります。この値は、通常はフラグメントの最終的な色としてハードウェアにより使用されます。一部のフラグメント プロファイルでは、フラグメントの深度値を変更できる **DEPTH** 出力セマンティクスもサポートされています。

頂点プログラムの場合と同様、フラグメント プログラムは、構造体の本体に出力を返すことがあります。ただし、通常は **out** パラメータとして出力を宣言するほうが便利です。

```
void main(/* ... */,
  out float4 color : COLOR, out float depth : DEPTH) {
  /* ...*/
  color = diffuseColor * /* ...*/;
  depth = /*...*/;
}
```

または、次のようにして、セマンティクスをシェーダの戻り値に関連付けます。

```
float4 main(/* ... */) : COLOR {
  /* ... */
  return diffuseCOlor * /* ... */;
}
```

次の例では、ディフューズおよびスペキュラ ライティングを計算する単純な頂点プログラムを示します。varying 型データのための 2 つの構造体が、**appin** および **vertout** として定義されています。プログラムが実行する内容を正確に理解する必要はありません。ここでの目標は、Cg コードがどのようになるかを示すことだけです。89 ページの「簡単なチュートリアル」で、このシェーダを詳細に説明します。

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

// Define outputs from vertex shader.
struct vertout
{
  float4 HPosition    : POSITION;
  float4 Color        : COLOR;
};

vertout main(appin IN,
             uniform float4x4 ModelViewProj,
             uniform float4x4 ModelViewIT,
             uniform float4 LightVec)
{
  vertout OUT;

  // Transform vertex position into homogenous clip-space.
  OUT.HPosition = mul(ModelViewProj, IN.Position);

  // Transform normal from model-space to view-space.
  float3 normalVec = normalize(mul(ModelViewIT,
                                   IN.Normal).xyz);

  // Store normalized light vector.
  float3 lightVec = normalize(LightVec.xyz);

  // Calculate half angle vector.
  float3 eyeVec = float3(0.0, 0.0, 1.0);
```

```
  float3 halfVec = normalize(lightVec + eyeVec);

  // Calculate diffuse component.
  float diffuse = dot(normalVec, lightVec);

  // Calculate specular component.
  float specular = dot(normalVec, halfVec);

  // Use the lit function to compute lighting vector from
  // diffuse and specular values.
  float4 lighting = lit(diffuse, specular, 32);

  // Blue diffuse material
  float3 diffuseMaterial = float3(0.0, 0.0, 1.0);

  // White specular material
  float3 specularMaterial = float3(1.0, 1.0, 1.0);

  // Combine diffuse and specular contributions and
  // output final vertex color.
  OUT.Color.rgb = lighting.y * diffuseMaterial +
                  lighting.z * specularMaterial;
  OUT.Color.a = 1.0;

  return OUT;
}
```

データの操作

C と同様に、Cg ではデータを作成および操作する機能がサポートされています。

- 基本型
- 構造体
- 配列
- 型変換

基本データ型

Cg では、6 つの基本データ型がサポートされます。

- **`float`**

  32 ビットの IEEE 浮動小数点数（s23e8）。1 つの符号ビット、23 ビットの仮数および 8 ビットの指数があります。このデータ型は、すべてのプロファイルでサポートされます。ただし、DirectX 8 ピクセル プロファイルでの実装では、一部の演算で精度と範囲が小さくなります。

- **half**

  16 ビットの IEEE に似た浮動小数点数（s10e5）。

- **int**

  32 ビットの整数。各プロファイルでは、このデータ型のサポートを省略している場合や、**int** を **float** として扱うオプションがある場合があります。

- **fixed**

  12 ビットの固定小数点数（s1.10）。このデータ型は、すべてのフラグメント プロファイルでサポートされます。

- **bool**

  ブール データ。比較によって生成され、**if** および条件演算子（**?:**）コンストラクトで使用されます。このデータ型は、すべてのプロファイルでサポートされます。

- **sampler***

  テクスチャ オブジェクトへのハンドル。**sampler**、**sampler1D**、**sampler2D**、**sampler3D**、**samplerCUBE** および **samplerRECT** の 6 種類があります。これらのデータ型は、1 つの例外を除き、すべてのピクセル プロファイルとフラグメント プロファイルでサポートされます。**samplerRECT** は、DirectX プロファイルではサポートされません。

Cg には、基本データ型に基づく組込みベクトル データ型もあります。これらの組込みベクトル データ型の例を次に示します（ただし、これらに限定されません）。

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

最大 4 × 4 の要素からなる行列用に、追加サポートが用意されています。次に、行列宣言の例を示します。

```
float1x1 matrix1;   // One element matrix
float2x3 matrix2;   // Two-by-three matrix (six elements)
float4x2 matrix3;   // Four-by-two matrix  (eight elements)
float4x4 matrix4;   // Four-by-four matrix (sixteen elements)
```

多次元配列 **float M[4][4]** は、**float4x4 M** 行列と等しい型ではないことに注意してください。

現時点では、Cg に共用体やビット フィールドはありません。

型変換

Cg における型変換は、C の型変換とほぼ同じように機能し、C のキャスト演算子 **(newtype)** を使用して明示的に型変換を指定できます。

型が混在する式では、C と同じく自動的に昇格が行われます。たとえば、式 **floatvar * halfvar** は、**floatvar * (float) halfvar** としてコンパイルされます。

型の昇格規則については、Cg と C で 1 つだけ異なる点があります。明示的に型の接尾辞を持たない定数では、型の昇格が行われません。Cg は、式 **halfvar * 2.0** を **halfvar * (half) 2.0** としてコンパイルします。

C の場合であれば、これは **((double) halfvar) * 2.0** にコンパイルされるところです。Cg が C と異なる規則を採用しているのは、意図しない型の昇格によって、精度は高いが処理速度の低い算術演算が実行されてしまうことを回避するためです。C と同じ動作が必要な場合には、次のように定数を明示的に型指定し、型の昇格を処理させる必要があります。**halfvar * 2.0f** は、**((float) halfvar) * 2.0f** としてコンパイルされます。

Cg の定数に使用される型の接尾辞は、次のとおりです。

- **float** の場合は **f**
- **half** の場合は **h**
- **fixed** の場合は **x**

構造体

Cg では、C と同じ方法で構造体をサポートしています。Cg は、**struct** が宣言されるときにタグ名に基づいて **typedef** を暗黙的に実行する C++ 方式を採用しています。

```
struct mystruct {
  /* ... */ };
mystruct s; // Define "s" as a "mystruct".
```

配列

Cg では配列をサポートしており、C の場合と同様に宣言します。Cg はポインタをサポートしないため、配列は常にポインタ構文ではなく配列構文を使用して定義する必要があります。

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

基本プロファイルでは、配列の宣言と使用法に一定の制限があります。汎用的な配列は、頂点プログラムへの uniform 型パラメータとしてのみ使用できます。これは、アプリケーションがスキニング行列の配列とライティング パラメータの配列を頂点プログラムに渡せるようにするためです。

C との最も重要な違いは、配列が基本型タイプであることです。これは、配列の代入によって実際には配列全体がコピーされ、パラメータとして渡される値が参照ではなく値によって渡される（配列に変更を加える前に配列全体がコピーされる）ことを意味します。

ステートメントと演算子

Cg では、次のタイプのステートメントと演算子をサポートしています。

- 制御フロー
- 関数定義と関数オーバーロード
- C の算術演算子
- 乗法関数
- ベクトル コンストラクタ
- ブール演算子と比較演算子
- スウィズル演算子
- 書込みマスク演算子
- 条件演算子

制御フロー

Cg では、次の C 制御コンストラクトを使用します。

- 関数呼出しと **return** ステートメント
- **if/else**
- **while**
- **for**

これらの制御コンストラクトでは、条件式の型が **bool** である必要があります。**i <= 3** などの Cg 式の型は **bool** であるため、C からのこの変更は通常は明らかではありません。

vs_2_x および **vp30** プロファイルは分岐命令をサポートしているため、これらのプロファイルでは **for** および **while** ループが完全にサポートされます。その他のプロファイルでは、**for** および **while** ループは、コンパイラでこれらを完全にアンロールできる場合（つまり、コンパイラがコンパイル時に反復回数を決定できる場合）にのみ使用できます。同様に、これらのプロファイルでは、**return** は関数の最後のステートメントとしてのみ使用できます。

Cg では、関数の再帰（および副再帰）は禁止されています。

switch、**case** および **default** のキーワードは予約語ですが、現在のリリースの Cg コンパイラでは、どのプロファイルでもサポートされていません。

関数定義と関数オーバーロード

C においては、変更可能なパラメータを関数に渡す場合、プログラマが明示的にポインタを使用する必要があります。C++ には、ポインタを明示的に使用する必要性を回避する参照渡しのメカニズムが組込みで用意されていますが、このメカニズムでも、ハードウェアでポインタがサポートされることが暗黙的に想定されます。GPU の頂点およびフラグメント ハードウェアはポインタの使用をサポートしないため、Cg では別のメカニズムを使用する必要があります。Cg は、参照ではなく値と結果によって、書き換え可能な関数パラメータを渡します。この 2 つの方法の違いは微妙であり、2 つの関数のパラメータが関数呼出しによってエイリアス化される場合にだけ明らかになります。Cg では、2 つのパラメータが関数に別々の記憶域を持ちます。一方、C++ では、2 つのパラメータが記憶域を共有します。

この区別を強調するために、Cg では C++ とは異なる構文を使用して、書き換え可能な関数のパラメータを宣言します。

```
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 (default, as in C)
```

Cg は、オペランド数とオペランド型によって関数オーバーロードをサポートします。関数の選択は、最初のオペランドから開始して一度に 1 つのオペランドを照合することによって行われます。正式な言語仕様では照合ルールの詳細が規定されていますが、オーバーロードは一般に直観的に動作するため、通常はこれらのルールを学習する必要はありません。たとえば、次のコードは 2 つの関数バージョンを宣言しています。一方は 2 つの **bool** オペランドを受け取り、もう一方は 2 つの **float** オペランドを受け取ります。

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

C の算術演算子

Cg には、C のすべての標準算術演算子（**+**、**-**、*****、**/**）が含まれており、これらの演算子をベクトルとスカラの両方で使用できます。ベクトル演算は、常に要素ごとに実行されます。次に例を示します。

float3(a, b, c) * float3(A, B, C) = **float3(a*A, b*B, c*C)**

これらの演算子は、スカラとベクトルが混在したフォームでも使用できます。スカラは、要素ごとの演算の実行に必要なサイズのベクトルを作成するために展開されます。したがって、次のようになります。

a * float3(A, B, C) = **float3(a*A, a*B, a*C)**

組込み算術演算子では、現在は行列オペランドがサポートされません。行列とベクトルの次元が同じであっても、その行列とベクトルは同じではないことに注意する必要があります。

乗法関数

Cg の **mul()** 関数では、行列にベクトルを、また行列に行列を乗算します。

```
// 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);
```

適切なバージョンの **mul()** を使用することが重要です。そうしないと、予期しない結果が返されることがあります。**mul()** 関数の詳細は、19 ページの「Cg 標準ライブラリ関数」を参照してください。

ベクトル コンストラクタ

Cg では、次の表記法を使用してベクトル（最大サイズ 4）を作成できます。

```
y = x * float4(3.0, 2.0, 1.0, -1.0);
```

ベクトル コンストラクタは、式の任意の場所に配置できます。

ブール演算子と比較演算子

Cg には、C の標準ブール演算子のうち 3 つが含まれています。

&& 論理 AND
|| 論理 OR
! 論理否定

C では、これらの演算子には **int** 型の値を使用し、生成される値も **int** 型ですが、Cg では、使用する値も生成される値も **bool** 型です。ブール式の値を保持する変数を宣言する場合を除き、通常はこの違いに注目する必要はありません。Cg は、**bool** 型の値を生成する C の比較演算子もサポートしています。

< より小さい
<= 以下
!= 等しくない
== 等しい
>= 以上
> より大きい

C とは異なり、Cg では、すべてのブール演算子をベクトルに適用できます。この場合、ブール演算は要素ごとに実行されます。このようなブール式の結果は、**bool** 要素のベクトルであり、要素数は 2 つのソース ベクトルと同じです。また、C とは異なり、AND（**&&**）および OR（**||**）論理演算子を使用して短絡評価することはできず、ブール式の値にかかわらず、これらの式の両辺では常に副作用が発生します。

スウィズル演算子

Cg には、スウィズル演算子（**.**）があります。この演算子では、ベクトルの成分を再配置して新しいベクトルを作成できます。新しいベクトルを元のベクトルと同じサイズにする必要はありません。要素は反復または省略できます。文字 **x**、**y**、**z** および **w** は、それぞれ元のベクトルの 1 番目、2 番目、3 番目および 4 番目の成分を表します。文字 **r**、**g**、**b** および **a** も同じ目的に使用できます。スウィズル演算子は、GPU ハードウェアで効率的に実装されており、通常はコストがかかりません。

次に、スウィズルの例を示します。

```
float3(a, b, c).zyx      → float3(c, b, a)
float4(a, b, c, d).xxyy  → float4(a, a, b, b)
float2(a, b).yyxx        → float4(b, b, a, a)
float4(a, b, c, d).w     → d
```

スウィズル演算子を使用して、スカラからベクトルを作成することもできます。

```
a.xxxx → float4(a, a, a, a)
```

スウィズル演算子の優先順位は、配列添字演算子（**[]**）の優先順位と同じです。

書込みマスク演算子

書込みマスク演算子（**.**）は、代入ステートメントの左側に配置されます。書込みマスクを使用すると、ベクトルのコンポーネントを選択的に上書きできます。書込みマスクで特定のコンポーネントを 2 回以上指定することや、宣言の一部として変数を初期化する際に書込みマスクを指定することは不正です。

次に、書込みマスクの例を示します。

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

書込みマスク演算子は、ハードウェアの機能によく対応しているため、効率的なコードを生成できる強力なツールとなります。書込みマスク演算子の優先順位は、スウィズル演算子の優先順位と同じです。

条件演算子

Cg には、C の **if/else** 条件ステートメントと条件演算子（**?:**）が含まれます。条件演算子では、制御変数を **bool** ベクトルにすることができます。この場合、2 番目と 3 番目のオペランドは同じサイズのベクトルである必要があり、選択は要素ごとに実行します。C とは異なり、条件にかかわらず、2 番目と 3 番目のオペランドに関連する副作用は常に発生します。

次の例では、**min()** 関数と **max()** 関数が存在しなかった場合に、ベクトル クランプ関数を非常に効率よく実装する方法を示しています。

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

拡張フラグメント プロファイルにおけるテクスチャ ルックアップ

Cg の拡張フラグメント プロファイルには、様々なテクスチャ ルックアップ関数が用意されています。ハードウェアのピクセル プログラミング機能に制限があるため、Cg では、基本フラグメント プロファイルに対して異なるテクスチャ ルックアップ関数のセットを使用することに注意してください。基本フラグメント プロファイルのルックアップ関数については、この概要の章では説明しません。

拡張フラグメント プロファイルのテクスチャ ルックアップ関数では、少なくとも次の 2 個のパラメータが常に必要です。

- テクスチャ サンプラ

  テクスチャ サンプラは **sampler**、**sampler1D**、**sampler2D**、**sampler3D**、**samplerCUBE** または **samplerRECT** の型を持つ変数で、テクスチャ画像とフィルタ、クランプ、ラップとの組合せ、または類似の構成を表します。テクスチャ サンプラ変数は、Cg 言語内で直接設定することはできず、アプリケーションによって uniform 型パラメータとして、Cg プログラムに指定される必要があります。

- テクスチャ座標

  テクスチャ ルックアップの種類に応じて、座標はスカラ、2 ベクトル、3 ベクトルまたは 4 ベクトルのいずれかになります。

次のフラグメント プログラムでは、**tex2D()** 関数を使用して 2D テクスチャ ルックアップを行い、フラグメントの RGBA カラーを決定しています。

```
void applytex(uniform sampler2D mytexture,
                      float2    uv       : TEXCOORD0,
              out     float4    outcolor : COLOR) {
  outcolor = tex2D(mytexture, uv);
}
```

Cg には、様々なテクスチャ ルックアップ関数が用意されています。次に、そのサンプルを示します。完全なリストは、25 ページの「テクスチャ マッピング関数」を参照してください。

- 標準の非射影テクスチャ ルックアップ

```
tex2D   (sampler2D   tex, float2 s);
texRECT (samplerRECT tex, float2 s);
texCUBE (samplerCUBE tex, float3 s);
```

- 標準の射影テクスチャ ルックアップ

```
tex2Dproj   (sampler2D   tex, float3 sq);
texRECTproj (samplerRECT tex, float3 sq)
texCUBEproj (samplerCUBE tex, float4 sq);
```

- ユーザ定義のフィルタ カーネル サイズを指定する非射影テクスチャ ルックアップ

```
tex2D   (sampler2D tex, float2 s,
         float2 dsdx, float2 dsdy);
texRECT (samplerRECT tex, float2 s,
         float2 dsdx, float2 dsdy);
texCUBE (samplerCUBE tex, float3 s,
         float3 dsdx, float3 dsdy);
```

  フィルタ サイズは、ピクセル座標 ***x***（***dsdx***）および ***y***（***dsdy***）に対するテクスチャ座標の導関数で指定します。詳細は、25 ページの「テクスチャ マッピング関数」を参照してください。

- シャドウマップ ルックアップ :

```
tex2Dproj (sampler2D   tex, float4 szq);
tex2DRECT (samplerRECT tex, float4 szq);
```

  これらの関数では、テクスチャ座標の ***z*** 成分が、シャドウマップと比較する深度値を保持します。シャドウマップ ルックアップでは、深度比較テクスチャリングのためにアプリケーションにより構成された関連テクスチャのユニットが必要です。これがないと、深度比較は実際に実行されません。

詳細情報

この章の目的は、Cg をすぐに使い始めて実地演習を体験できるように、Cg の概要を簡単に示すことです。この章で説明している言語機能の詳細は、165 ページの「Cg 言語仕様」を参照してください。

Cg 標準ライブラリ関数

GPU プログラミングを簡略化するため、Cg には組込み関数と、バインディング セマンティクスを持つ事前定義済みの構造体のセットが用意されています。これらの関数は、C の標準ライブラリと性質が類似しており、よく使用する関数の便利なセットを提供します。多くの場合、関数は単一のネイティブ GPU 命令にマッピングされ、非常に高速に実行されます。これらの関数の中には複数のネイティブ GPU 命令にマッピングされるものもあり、近い将来には、これらの関数が、最も有効でより効率的なものになる予定です。

パフォーマンスまたは精度上の理由から、特定の関数をカスタマイズして記述することもできますが、一般には、可能なかぎり標準ライブラリ関数を使用する方が賢明です。標準ライブラリ関数は、将来の GPU に対しても引き続き最適化されるため、現在作成されているシェーダは、コンパイル時に最新のアーキテクチャ用に自動的に最適化されます。また、標準ライブラリには、頂点プログラムとフラグメント プログラムの便利な統一インタフェースが用意されています。

この章では、次の Cg 標準ライブラリの内容について説明します。

- 数学関数
- 幾何学関数
- テクスチャ マッピング関数
- 導関数
- 定義済みのヘルパー **`struct`** 型

関数は、入力と出力の型が同じ場合にスカラとベクトルのバリエーションをサポートするために適宜オーバーロードされます。

数学関数

表 1 に、Cg 標準ライブラリに用意されている数学関数をリストします。リストには、特に三角関数、指数関数、丸め、ベクトルおよび行列演算に役立つ関数が含まれています。すべての関数は、特に注記しないかぎり、すべてのサイズのスカラとベクトルに対して動作します。

表 1　数学関数

数学関数	
関数	**説明**
abs(x)	*x* の絶対値。
acos(x)	*x* のアークコサイン。値域は [-π/2,π/2]。*x* は [-1,1]。
all(x)	*x* のすべての要素が 0 でない場合は true を返します。 それ以外の場合は false を返します。
any(x)	*x* のいずれかの要素が 0 でない場合は true を返します。 それ以外の場合は false を返します。
asin(x)	*x* のアークサイン。値域は [0,π]。 *x* は [-1,1] である必要があります。
atan(x)	*x* のアークタンジェント。値域は [-π/2,π/2]。
atan2(y, x)	*y/x* のアークタンジェント。値域は [-π,π]。
ceil(x)	*x* 以上の最小の整数。
clamp(x, a, b)	*x* が次のように値域 [*a*, *b*] にクランプされます。 • *x* が *a* 未満の場合は *a* を返します。 • *x* が *b* より大きい場合は *b* を返します。 • それ以外の場合は *x* を返します。
cos(x)	*x* のコサイン。
cosh(x)	*x* のハイパーボリック コサイン。
cross(a, b)	ベクトル *a* と *b* の外積。 *a* および *b* は 3 成分ベクトルである必要があります。
degress(*x*)	ラジアンから度への変換。
determinant(*M*)	行列 *M* の行列式。
dot(*a*, *b*)	ベクトル *a* と *b* の内積。
exp(*x*)	指数関数 e^x。
exp2(*x*)	指数関数 2^x。
floor(*x*)	*x* 以下の最大の整数。
fmod(*x*, *y*)	*x/y* の剰余。符号は *x* と同じです。 *y* が 0 の場合、その結果は実装に依存します。
frac(*x*)	*x* の小数部分。

表 1　数学関数 （続き）

数学関数	
関数	**説明**
`frexp(x, out exp)`	*x* を区間[1/2,1)の正規化した小数部と2のべき乗に分解します。正規化した小数部を返し、2 のべき乗を *exp* に格納します。 *x* が 0 の場合は、結果の両方の部分が 0 になります。
`isfinite(x)`	*x* が有限の場合に `true` を返します。
`isinf(x)`	*x* が無限の場合に `true` を返します。
`isnan(x)`	*x* が NaN（数値以外）の場合に `true` を返します。
`ldexp(x, n)`	$x * 2^n$
`lerp(a, b, f)`	線形補間： `(1-f)*a + b*f`。*a* および *b* は、一致するベクトルまたはスカラ型です。*f* は、*a* および *b* と同じ型のスカラかベクトルのいずれかです。
`lit(ndotl, ndoth, m)`	環境ライティング、ディフューズ ライティングおよびスペキュラ ライティングの寄与を表すライティングの係数を計算します。4 成分ベクトルを次のように返します。 • 結果ベクトルの *x* 成分には、常に 1.0 である環境係数が含まれます。 • *y* 成分には、`(n ● l)` < 0 の場合には 0、それ以外の場合には `(n ● l)` であるディフューズ係数が含まれます。 • *z* 成分には、`(n ● l)` < 0 または `(n ● h)` < 0 の場合には 0、それ以外の場合には $(n \bullet h)^m$ であるスペキュラ係数が含まれます。 • *w* 成分は 1.0 です。 この関数のベクトル化バージョンはありません。
`log(x)`	自然対数 `ln(x)`。 *x* は 0 より大きい必要があります。
`log2(x)`	底 2 の *x* の対数。 *x* は 0 より大きい必要があります。
`log10(x)`	底 10 の *x* の対数。 *x* は 0 より大きい必要があります。
`max(a, b)`	*a* と *b* の最大値。
`min(a, b)`	*a* と *b* の最小値。

表 1　数学関数 （続き）

数学関数	
関数	**説明**
modf(*x*, out *ip*)	***x*** **を、それぞれ** ***x*** **と同じ符号を持つ整数部と小数部に分解します。** **整数部を** ***ip*** **に格納し、小数部を返します。**
mul(*M*, *N*)	**次に示すように、行列** ***M*** **と行列** ***N*** **の行列積。** $\text{mul(M, N)} = \begin{bmatrix} M_{11} & M_{21} & M_{31} & M_{41} \\ M_{12} & M_{22} & M_{32} & M_{42} \\ M_{13} & M_{23} & M_{33} & M_{43} \\ M_{14} & M_{24} & M_{34} & M_{44} \end{bmatrix} \begin{bmatrix} N_{11} & N_{21} & N_{31} & N_{41} \\ N_{12} & N_{22} & N_{32} & N_{42} \\ N_{13} & N_{23} & N_{33} & N_{43} \\ N_{14} & N_{23} & N_{34} & N_{44} \end{bmatrix}$ ***M*****のサイズが** **A** **×** **B** **で、*****N*****のサイズが** **B** **×** **C** **の場合は、サイズ** **A** **×** **C** **の行列を返します。**
mul(*M*, *v*)	**次に示すように、行列** ***M*** **と列ベクトル** ***v*** **の積。** $\text{mul(M, v)} = \begin{bmatrix} M_{11} & M_{21} & M_{31} & M_{41} \\ M_{12} & M_{22} & M_{32} & M_{42} \\ M_{13} & M_{23} & M_{33} & M_{43} \\ M_{14} & M_{24} & M_{34} & M_{44} \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \end{bmatrix}$ ***M*****が** **A** **×** **B** **の行列で、*****v*****が** **B** **×** **1** **のベクトルの場合は、** **A** **×** **1** **のベクトルを返します。**
mul(*v*, *M*)	**次に示すように、行ベクトル** ***v*** **と行列** ***M*** **の積。** $\text{mul(v, M)} = [V_1 \quad V_2 \quad V_3 \quad V_4] \begin{bmatrix} M_{11} & M_{21} & M_{31} & M_{41} \\ M_{12} & M_{22} & M_{32} & M_{42} \\ M_{13} & M_{23} & M_{33} & M_{43} \\ M_{14} & M_{24} & M_{34} & M_{44} \end{bmatrix}$ ***v*****が** **1** **×** **A** **のベクトルで、*****M*****が** **A** **×** **B** **の行列の場合は、** **1** **×** **B** **のベクトルを返します。**
noise(*x*)	**引数の型に応じて、** 1 **次元、** 2 **次元または** 3 **次元のノイズ関数。** **戻り値は** 0 **～** 1 **の範囲で、特定の入力値に対して常に同じになります。**
pow(*x*, *y*)	x^y
radians(*x*)	**度からラジアンへの変換。**
round(*x*)	***x*** **に最も近い整数。**
rsqrt(*x*)	***x*** **の平方根の逆数。** ***x*** **は** 0 **より大きい必要があります。**

表 1　数学関数 （続き）

数学関数	
関数	**説明**
sign(*x*)	*x* > 0 の場合は 1。 *x* < 0 の場合は -1。 それ以外の場合は 0。
sin(*x*)	*x* のサイン。
sincos(float *x*, out *s*, out *c*)	*s* は *x* のサインに設定され、*c* は *x* のコサインに設定されます。 sin(*x*) と cos(*x*) の両方が必要な場合、この関数はそれぞれ個別に計算するよりも効率的です。
sinh(*x*)	*x* のハイパーボリック サイン。
smoothstep(*min*, *max*, *x*)	*x* の値が *min* と *max* の間にある場合は、*x* = *min* のときの 0 から *x* = *max* のときの 1 までの値域で徐々に変化する値を返します。*x* は、値域 [*min*,*max*] にクランプされ、補間式が評価されます。 $-2*((x-min)/(max-min))^3 + 3*((x-min)/(max-min))^2$
step(*a*, *x*)	*x* < *a* の場合は 0。 *x* >= *a* の場合は 1。
sqrt(*x*)	*x* の平方根。 *x* は 0 より大きい必要があります。
tan(*x*)	*x* のタンジェント。
tanh(*x*)	*x* のハイパーボリック タンジェント。
transpose(*M*)	行列 *M* の転置行列。*M* が A × B の行列である場合、*M* の転置は B × A の行列で、その第 1 列は *M* の第 1 行、第 2 列は *M* の第 2 行、第 3 列は *M* の第 3 行のようになります。

幾何学関数

表 2 に、Cg 標準ライブラリに用意されている幾何学関数を示します。

表 2　幾何学関数

幾何学関数	
関数	**説明**
`distance(pt1, pt2)`	ポイント *pt1* と *pt2* の間のユークリッド距離。
`faceforward(N, I, Ng)`	`dot(Ng,I)` < 0 の場合は *N*。 それ以外の場合は *-N*。
`length(v)`	ベクトルのユークリッド長。
`normalize(v)`	ベクトル *v* と同じ方向を指す長さ 1 のベクトルを返します。
`reflect(i, n)`	入射方向 *i* と表面法線 *n* から反射ベクトルを計算します。 3 成分ベクトルに対してのみ有効です。
`refract(i, n, eta)`	与えられた入射方向 *i*、表面法線 *n* および相対屈折率 *eta* に対して、屈折ベクトルを計算します。*i* と *n* の間の角度が、与えられた *eta* に対して大きすぎる場合は、(0,0,0) を返します。 3 成分ベクトルに対してのみ有効です。

テクスチャ マッピング関数

表 3 に、Cg 標準ライブラリに用意されているテクスチャ関数を示します。これらのテクスチャ関数が完全にサポートされるのは **ps_2**、**arbvp1** および **fp30** プロファイルのみですが、将来的にはテクスチャ マッピング機能を持つすべてのプロファイルでサポートされる予定です。表 3 のすべての関数は **float4** 値を返します。

古いハードウェアのピクセル プログラミング機能には制限があるため、**ps_1** および **fp20** プロファイルでは、使用するテクスチャ マッピング関数のセットが異なります。詳細は 197 ページの「言語プロファイル」を参照してください。

表 3　テクスチャ マッピング関数

テクスチャ マッピング関数	
関数	**説明**
tex1D(sampler1D *tex*, float *s*)	**1 次元の非射影**
tex1D(sampler1D *tex*, float *s*, float *dsdx*, float *dsdy*)	**1 次元の非射影、導関数付き**
tex1D(sampler1D *tex*, float2 *sz*)	**1 次元の非射影深度比較**
tex1D(sampler1D *tex*, float2 *sz*, float *dsdx*, float *dsdy*)	**1 次元の非射影深度比較、導関数付き**
tex1Dproj(sampler1D *tex*, float2 *sq*)	**1 次元の射影**
tex1Dproj(sampler1D *tex*, float3 *szq*)	**1 次元の射影深度比較**
tex2D(sampler2D *tex*, float2 *s*)	**2 次元の非射影**
tex2D(sampler2D *tex*, float2 *s*, float2 *dsdx*, float2 *dsdy*)	**2 次元の非射影、導関数付き**
tex2D(sampler2D *tex*, float3 *sz*)	**2 次元の非射影深度比較**

表 3　テクスチャ マッピング関数 （続き）

テクスチャ マッピング関数	
関数	**説明**
tex2D(sampler2D *tex*, float3 *sz*, float2 *dsdx*, float2 *dsdy*)	2 次元の非射影深度比較、導関数付き
tex2Dproj(sampler2D *tex*, float3 *sq*)	2 次元の射影
tex2Dproj(sampler2D *tex*, float4 *szq*)	2 次元の射影深度比較
texRECT(samplerRECT *tex*, float2 *s*)	2 次元の RECT 非射影
texRECT(samplerRECT *tex*, float2 *s*, float2 *dsdx*, float2 *dsdy*)	2 次元の RECT 非射影、導関数付き
texRECT(samplerRECT *tex*, float3 *sz*)	2 次元の RECT 非射影深度比較
texRECT(samplerRECT *tex*, float3 *sz*, float2 *dsdx*, float2 *dsdy*)	2 次元の RECT 非射影深度比較、導関数付き
texRECTproj(samplerRECT *tex*, float3 *sq*)	2 次元の RECT 射影
texRECTproj(samplerRECT *tex*, float3 *szq*)	2 次元の RECT 射影深度比較
tex3D(sampler3D *tex*, float3 *s*)	3 次元の非射影
tex3D(sampler3D *tex*, float3 *s*, float3 *dsdx*, float3 *dsdy*)	3 次元の非射影、導関数付き
tex3Dproj(sampler3D *tex*, float4 *szq*)	3 次元の射影深度比較

表 3　テクスチャ マッピング関数 （続き）

テクスチャ マッピング関数	
関数	**説明**
`texCUBE(samplerCUBE tex, float3 s)`	キューブマップの非射影
`texCUBE(samplerCUBE tex, float3 s, float3 dsdx, float3 dsdy)`	キューブマップの非射影、導関数付き
`texCUBEproj(samplerCUBE tex, float4 sq)`	キューブマップの射影

表では、各関数の第 2 引数の名前は、テクスチャ ルックアップの実行時の値の使用方法を示しています。***s*** は、1、2 または 3 成分のテクスチャ座標を示します。***z*** は、シャドウマップ ルックアップの深度比較値を示します。***q*** はパースペクティブ値を示し、テクスチャ ルックアップが実行される前にテクスチャ座標 ***s*** の除算に使用されます。

便宜を図るため、標準ライブラリでは、**half4** 値を返す **h4** 接頭辞の付いた **h4tex2D()** など、および **fixed4** 値を返す **x4** 接頭辞の付いた **x4tex2D()** などのテクスチャ関数のバージョンも定義しています。

深度比較値を指定できるテクスチャ関数が使用される場合、関連するテクスチャ ユニットは深度比較テクスチャリング用に構成される必要があります。それ以外の場合は、深度比較は実際に実行されません。

導関数

表 4 に、Cg 標準ライブラリでサポートされている導関数を示します。頂点プロファイルは、これらの関数をサポートする必要はありません。

表 4　導関数

導関数	
関数	**説明**
ddx(*a*)	**スクリーン空間の x 座標に対して、*a* の偏導関数を近似する。**
ddy(*a*)	**スクリーン空間の y 座標に対して、*a* の偏導関数を近似する。**

デバッグ関数

表 5 に、Cg 標準ライブラリでサポートされているデバッグ関数を示します。頂点プロファイルは、この関数をサポートする必要はありません。

表 5　デバッグ関数

デバッグ関数	
関数	**説明**
`void debug(float4 x)`	**コンパイラの DEBUG オプションを指定した場合、この関数をコールすると、値 *x* がプログラムの COLOR 出力にコピーされ、プログラムの実行が停止します。** **コンパイラの DEBUG オプションが指定していない場合、この関数は何も行いません。**

デバッグ関数は、プログラムを 2 回コンパイルできるようにすることを目的としています。1 回は **DEBUG** オプションを指定してコンパイルし、もう 1 回はこのオプションを指定せずにコンパイルします。両方のプログラムを実行することにより、一方のフレーム バッファにプログラムの最終出力を表示し、もう一方にデバッグで検証する中間値を表示することができます。

定義済みのフラグメント プログラム出力構造体

標準ライブラリでは、フラグメント プログラムで使用するいくつかのヘルパー構造体が事前定義されています。これらの型の変数を使用し、フラグメント プログラムの出力を保持できます。これらの変数の使用は完全に随意です。

ps_1 および **fp20** プロファイルでは、**fragout** 構造体は次のように定義されています。

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

ps_2、**arbfp1** および **fp30** プロファイルには、2 つのフラグメント出力タイプが定義されています。

```
struct fragout  {
  half4 col   : COLOR;
  float depth : DEPTH;
};
struct fragout_float  {
  float4 col   : COLOR;
  float  depth : DEPTH;
};
```

Cg ランタイム ライブラリの使用

この章では、Cg ランタイム ライブラリについて説明します。ここでは、Cg 言語についての基礎知識と、アプリケーションで使用する API に応じて OpenGL API または Direct3D API についての基礎知識があることを前提とします。

最初の節（29 ページの「Cg ランタイムの概要」）では、Cg ランタイム ライブラリを使用する利点について説明し、アプリケーションでの使用方法の概要を説明します。次の 2 つの節（35 ページの「コア Cg ランタイム」および 45 ページの「API 固有の Cg ランタイム」）では、Cg ランタイムを構成する API をすべて説明します。

Cg ランタイムの概要

Cg プログラムはシェーディングを記述するコード行ですが、画像の生成にはアプリケーションのサポートが必要です。Cg プログラムとアプリケーションとの間をインタフェースするには、次の 2 つの操作を実行する必要があります。

1. 適切なプロファイルに対してプログラムをコンパイルします。つまり、アプリケーションおよび下層のハードウェアで使用される 3D API と互換性のある形式にプログラムをコンパイルします。
2. プログラムをアプリケーション プログラムにリンクします。これにより、アプリケーションは varying 型データと uniform 型データをプログラムに転送できます。

これらの操作を実行するタイミングには、コンパイル時つまりアプリケーション プログラムを実行可能ファイルにコンパイルする際と、実行時つまりアプリケーションを実際に実行する際の 2 つの選択肢があります。Cg ランタイムは、アプリケーションが Cg プログラムをランタイムにコンパイル、リンクできるようにするためのアプリケーション プログラミング インタフェースです。

Cg ランタイムの利点

将来の互換性

ほとんどのアプリケーションは、いくつかのプロファイルに対して実行する必要があります。アプリケーションが Cg プログラムを事前にコンパイルしてある場合（コンパイル時を選択した場合）は、プロファイルごとに各プログラムのコンパイル済みバージョンを格納する必要があります。これは 1 つのプログラムに対しては妥当ですが、多くのプログラムを使用するアプリケーションでは煩雑で

す。さらに悪いことに、アプリケーションは時間軸の中のある一点に凍結されてしまいます。この方法では、コンパイル時に存在していたプロファイルのみがサポートされ、将来のコンパイラで提供される最適化は利用できません。

これに対し、実行時にアプリケーションによりプログラムをコンパイルすると、次の利点があります。

- 既存のプロファイルに対して将来のコンパイラの最適化を利用できる。
- Cg プログラムの作成時に存在していなかった新しい 3D API またはハードウェアに対応する将来のプロファイルで実行できる。

依存性の制限がない

Cg プログラムをコンパイル時にアプリケーションにリンクする場合は、アプリケーションがコンパイル結果に依存する度合いが高くなります。アプリケーション プログラムは、Cg コンパイラから出力されるハードウェア レジスタ名を使用して Cg プログラムの入力パラメータを参照する必要があるからです。このアプローチは、2 つの理由で不適当です。

- レジスタ名と、Cg プログラム内で対応する意味のある名前との対応を、コンパイラの出力を参照せずに簡単に行うことができない。
- Cg プログラム、Cg コンパイラまたはコンパイル プロファイルが変更されるたびに、レジスタ割当てが変更される可能性がある。このため、アプリケーションも毎回更新しなければならない。

これに対し、実行時に Cg プログラムをアプリケーション プログラムにリンクすると、Cg コンパイラへの依存性が排除されます。ランタイムでは、アプリケーション コードの変更が必要になるのは、Cg の入力パラメータを追加、削除または変更するときのみです。

入力パラメータ管理

Cg ランタイムには、Cg プログラムの入力パラメータを管理するための追加機能も用意されています。特に、配列や行列などのデータ型を簡単に扱えるようになります。追加機能には、必要な 3D API コールをラップする関数が含まれており、これによりコード長は短くなり、プログラマの誤りも減少します。

Cg ランタイムの概要

Cg ランタイム API は、3 つの部分（図 2）から構成されます。

- ランタイムの機能全体をカプセル化する、関数と構造体のコア セット
- コア セットの上に構築される、OpenGL 固有の関数セット
- コア セットの上に構築される、Direct3D 固有の関数セット

アプリケーションの作成を簡単にするために、OpenGL と Direct3D のランタイム ライブラリは、それぞれの API の考え方とデータ構造を取り込んでいます。

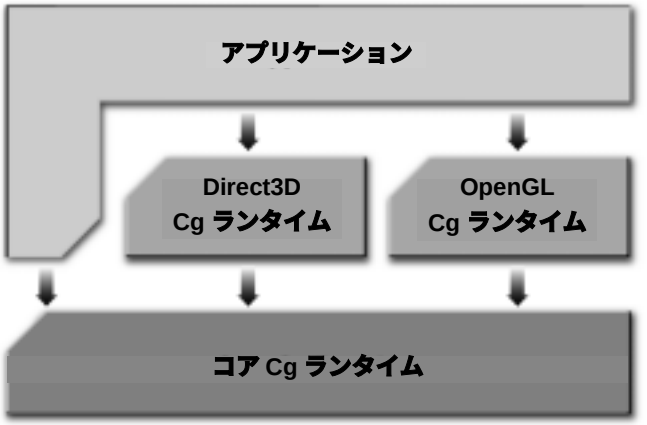


図 2　Cg ランタイム API の各部分

これ以降の部分では、アプリケーションのフレームワークで Cg ランタイムを使用するための説明を記載しています。各ステップには、OpenGL および Direct3D プログラミングのソース コードが含まれます。

純粋な Cg リソース管理にのみ関係する関数はコア ランタイムに属し、**cg** という接頭辞が付いています。この場合、OpenGL と Direct3D で同じコードが使用されます。

OpenGL または Direct3D Cg ランタイムの関数が使用される場合は、API 名が関数名で示されることに注意してください。OpenGL Cg ランタイム ライブラリに属する関数には **cgGL** 接頭辞が付き、Direct3D Cg ランタイム ライブラリの関数には **cgD3D** 接頭辞が付きます。

実際には、2 つの Direct3D Cg ランタイム ライブラリがあります。1 つは Direct3D 8 用で、もう 1 つは Direct3D 9 用です。Direct3D 8 Cg ランタイムに属する関数には **cgD3D8** 接頭辞が付き、Direct3D 9 Cg ランタイムに属する関数には **cgD3D9** 接頭辞が付きます。ほとんどの関数は 2 つのランタイム間で同一であるため、Direct3D 9 Cg ランタイムについては、特に注記しないかぎりその説明が Direct3D 8 Cg ランタイムにも適用されるという前提で説明しています。

関数名に使用されるのと同じ接頭辞規則が、型名、マクロ名および列挙値にも使用されます。

ヘッダー ファイル

次に、コア Cg ランタイム API を C または C++ プログラムにインクルードする方法を示します。

```
#include <Cg/cg.h>
```

次に、OpenGL Cg ランタイム API のインクルード方法を示します。

```
#include <Cg/cgGL.h>
```

次に、Direct3D 9 Cg ランタイム API のインクルード方法を示します。

```
#include <Cg/cgD3D9.h>
```

次に、Direct3D 8 Cg ランタイム API のインクルード方法を示します。

```
#include <Cg/cgD3D8.h>
```

コンテキストの作成

コンテキストは、複数の Cg プログラムのコンテナです。コンテキストは、Cg プログラムとその共有データを保持します。

次に、コンテキストの作成方法を示します。

```
CGcontext context = cgCreateContext();
```

プログラムのコンパイル

Cg プログラムをコンパイルするには、**cgCreateProgram()** を使用してコンテキストに追加します。

```
CGprogram program = cgCreateProgram(context,
                                    CG_SOURCE, myVertexProgramString,
                                    CG_PROFILE_ARBVP1, "main", args);
```

CG_SOURCE は、文字列引数 ***myVertexProgramString*** に、Cg ソースコード（プリコンパイルされたオブジェクト コードではない）が含まれることを示します。実際に Cg ランタイムでは、必要に応じてプリコンパイルされたオブジェクト コードからプログラムを作成することもできます。

CG_PROFILE_ARBVP1 は、プログラムのコンパイル先のプロファイルです。**"main"** パラメータには、プログラムの実行時にメイン エントリ ポイントとして使用される関数の名前を指定します。最後に ***args*** は、コンパイラに引数として渡される null 終了文字列の null 終了リストです。

プログラムのロード

プログラムをコンパイルしたら、生成されるオブジェクト コードを、使用する 3D API に渡します。このために、Cg ランタイムの API 固有関数をコールする必要があります。

Direct3D 固有の関数では、必要な Direct3D コールを行うために Direct3D デバイス構造体が必要です。アプリケーションは、次のコールを使用してこれをランタイムに渡します。

```
cgD3D9SetDevice(Device);
```

このコールは、新しい Direct3D デバイスが作成されるたびに行う必要があります。通常、デバイスはアプリケーションの開始時にのみ作成されます。

これで、次のようにして Cg プログラムを Direct3D 9 Cg ランタイムにロードできるようになります。

```
cgD3D9LoadProgram(program, CG_FALSE, 0);
```

Direct3D 8 Cg ランタイムに対してロードする場合は、次のようにします。

```
cgD3D8LoadProgram(program, CG_FALSE, 0, 0, vertexDeclaration);
```

パラメータ **vertexDeclaration** は、必要な頂点属性が頂点ストリームのどこにあるかを記述する Direct3D 8 の頂点宣言配列です。（**cgD3D8LoadProgram()** および **cgD3D9LoadProgram()** への引数の詳細は、75 ページの「拡張インタフェース プログラムの実行」を参照してください。）

OpenGL では、同等のコールは次のとおりです。

```
cgGLLoadProgram(program);
```

プログラム パラメータの変更

ランタイムには、プログラム パラメータの値を変更するオプションがあります。最初の手順は、パラメータへのハンドルの取得です。

```
CGparameter myParameter = cgGetNamedParameter(
                                    program, "myParameter");
```

変数 ***"myParameter"*** は、プログラム ソース コードに記述されているとおりのパラメータの名前です。

第 2 の手順は、パラメータ値の設定です。使用される関数は、パラメータの型によって決まります。

次に、OpenGL での例を示します。

```
cgGLSetParameter4fv(myParameter, value);
```

次に、Direct3D での同じ例を示します。

```
cgD3D9SetUniform(myParameter, value);
```

これらの関数コールは、配列 ***value*** に含まれる 4 つの浮動小数点値を、**float4** 型として想定されているパラメータ ***myParameter*** に割り当てます。

どちらの API でも、これらのコールには行列、配列、テクスチャおよびテクスチャ ステートを設定するためのバリエーションがあります。

プログラムの実行

OpenGL では、プログラムを実行する前に、対応するプロファイルを有効にする必要があります。

```
cgGLEnableProfile(CG_PROFILE_ARBVP1);
```

Direct3D では、特定のプロファイルを有効にする明示的な操作は不要です。

次に、プログラムを現在のステートにバインドします。つまり、後続の描画コールで、頂点プログラムの場合には各頂点に対して、フラグメント プログラムの場合には各フラグメントに対してプログラムが実行されます。

次に、OpenGL でのプログラムのバインド方法を示します。

```
cgGLBindProgram(program);
```

次に、Direct3D でのプログラムのバインド方法を示します。

```
cgD3D9BindProgram(program);
```

ある特定のプロファイルに対して、一度に 1 つの頂点プログラムと 1 つのフラグメント プログラムのみバインドできます。したがって、別の頂点プログラムがバインドされるまでは、同じ頂点プログラムが実行されます。同様に、別のフラグメント プログラムがバインドされるまでは、同じフラグメント プログラムが実行されます。

OpenGL では、次のコールによりプロファイルを無効にします。

```
cgGLDisableProfile(CG_PROFILE_ARBVP1);
```

プロファイルを無効にすると、対応する頂点プログラムまたはフラグメント プログラムも実行されなくなります。

リソースの解放

アプリケーションを終了する準備ができたら、取得したリソースを解放するのが正しいプログラミング作法です。

Direct3D ランタイムは Direct3D デバイスへの内部参照を保持するため、ランタイムの使用終了時にこの参照を解放するよう指示する必要があります。これは、次のコールで行います。

```
cgD3D9SetDevice(0);
```

プログラムに割り当てられているリソースを解放するには、次の関数をコールします。

```
cgDestroyProgram(program);
```

コンテキストに割り当てられているリソースを解放するには、次の関数を使用します。

```
cgDestroyContext(context);
```

コンテキストを破棄すると、そのコンテキストに含まれるすべてのプログラムも破棄されることに注意してください。

コア Cg ランタイム

コア Cg ランタイムには、アプリケーションから Cg プログラムを管理するために必要なすべての関数が用意されています。コア Cg ランタイムにおいては、アプリケーションがどちらの 3D API を使用しているかは関係ないため、API 固有の Cg ランタイム ライブラリや構成要素に関して考える必要はありません。

コア Cg ランタイムはコンテキスト、プログラムおよびパラメータの 3 つの主要概念によって構築され、これらは **CGcontext**、**CGprogram** および **CGparameter** オブジェクト型で表されます。3 つの概念は階層的な関係にあります。つまりプログラムには複数のパラメータがあり、コンテキストには複数のプログラムが含まれ、アプリケーションでは複数のコンテキストを定義できます。

注意： **将来的には、コンテキストのレベルでもパラメータを定義できるようになり、同じコンテキストのすべてのプログラムでパラメータが共有されるようになります。**

次以降の項では、この 3 つの基本オブジェクト型とその関連機能について検討します。3 つのオブジェクト型には、共通する点がいくつかあります。

- **CGbool** の使用法。これは **CG_TRUE** または **CG_FALSE** と等しい整数型です。
- **CGenum** の使用法。これは、各種列挙値の指定に使用される列挙型です。それぞれの値は、必ずしも関連し合っているとは限りません。
- **CGcontext**、**CGprogram**、**CGparameter** または **const char*** 型の値を返す関数が、0 を返すことでエラーを示すという規則。

コア Cg コンテキスト

Cg には、コンテキストを作成、破棄および問合せする関数が用意されています。

コンテキストの作成と破棄

プログラムは、プログラムのコンテナとして機能するコンテキストの一部としてのみ作成できます。コンテキストを作成するには、次のように **cgCreateContext()** をコールします。

```
CGcontext cgCreateContext();
```

コンテキストを破棄するには、次のように **cgDestroyContext()** をコールします。

```
void cgDestroyContext(CGcontext context);
```

コンテキスト問合せ

コンテキスト ハンドルが有効なコンテキストを参照しているかどうかをチェックするには、次のように **cgIsContext()** を使用します。

```
CGbool cgIsContext(CGcontext context);
```

コア Cg プログラム

プログラムを作成、破棄、反復（iterate）および問合せする Cg 関数があります。

プログラムの作成と破棄

プログラムは、次のように **cgCreateProgram()** をコールすることによって作成されます。

```
CGprogram cgCreateProgram(CGcontext    context,
                          CGenum       programType,
                          const char*  program,
                          CGprofile    profile,
                          const char*  entry,
                          const char** args);
```

または、**cgCreateProgramFromFile()** をコールして作成することもできます。

```
CGprogram cgCreateProgramFromFile(CGcontext    context,
                                  CGenum       programType,
                                  const char*  program,
                                  CGprofile    profile,
                                  const char*  entry,
                                  const char** args);
```

これらの関数は、プログラム オブジェクトを作成し、指定されたコンテキストにそれを追加し、関連付けられているソース コードをコンパイルします。2 つのコールについて、次の点は共通です。

- ***context*** は、有効なコンテキスト ハンドルです。
- ***profile*** は、プログラムをコンパイルする対象プロファイルを指定する列挙です。
- ***entry*** は、コンパイラによりメイン エントリ ポイントとみなされる必要のある関数名です。値が 0 の場合は、**main** という名前が使用されます。
- ***args*** は、コンパイラに引数として渡される null 終了文字列の null 終了配列です。ポインタ自体が null でもかまいません。

2 つの関数の違いは、***program*** の解釈だけです。**cgCreateProgramFromFile()** の場合、***program*** はソース コードが格納されているファイル名を含む文字列であり、**cgCreateProgram()** の場合、***program*** はソース コードを直接含みます。列挙 ***programType*** が **CG_SOURCE** と等しければ、ソース コードは Cg ソース コードです。**CG_OBJECT** と等しければ、ソース コードはプリコンパイルされた、さらにコンパイルする必要のないオブジェクト コードです。

cgCreateProgramFromFile() から返される **CGprogram** ハンドルは、0 以外の場合は有効です。この場合、プログラムが正常に作成され、コンパイルされたことを意味します。プログラムは、次のように **cgDestroyProgram()** にハンドルを渡すことによって破棄されます。

```
void cgDestroyProgram(CGprogram program);
```

注意： 将来的には、**cgCreateProgram()** または **cgCreateProgramFromFile()** により作成されたプログラムを、ランタイムを介して変更できるようになる（たとえば一部のパラメータの可変性またはセマンティクスを変更することによって）ため、再コンパイルが必要になります。

cgIsProgramCompiled() のコールは、プログラムの再コンパイルが必要かどうかを判断します。

```
CGbool cgIsProgramCompiled(CGprogram program);
```

プログラムを再コンパイルするには、次のように **cgCompileProgram()** を使用します。

```
cgCompileProgram(CGprogram program);
```

このコンテキストで役立つ関数は **cgCopyProgram()** です。

```
CGprogram cgCopyProgram(CGprogram program);
```

この関数は、***program*** のコピーである新しいプログラム オブジェクトを作成し、これを同じコンテキストに追加します。このため、同じオリジナル プログラムの複数のバージョンを作成し、それぞれを特定の方法で変更することができます。

プログラムの反復（Iteration）

コンテキスト内のプログラムは順次に並べられており、**cgGetFirstProgram()** および **cgGetNextProgram()** を使用して反復（Iteration）できます。

```
CGprogram cgGetFirstProgram(CGcontext context);
CGprogram cgGetNextProgram(CGprogram program);
```

一連のプログラムのうちの最初のプログラムは、**cgGetFirstProgram()** によって取得されます。コンテキストが無効であるか、プログラムを含んでいない場合、この関数は 0 を返します。プログラムがあった場合、**cgGetNextProgram()** はすぐ次の順番のプログラムを返すか、次のプログラムがない場合は 0 を返します。次の例は、***context*** という名前の有効なコンテキストがある場合に、この 2 つの関数が通常どのように使用されるかを示したものです。

```
CGprogram program = cgGetFirstProgram(context);
while (program != 0) {
  /* Here is the code that handles the program */
  program = cgGetNextProgram(program);
}
```

反復中にプログラムが作成または破棄される場合には、プログラムの順序についても、**cgGetFirstProgram()** および **cgGetNextProgram()** の動作についても、何の保証もありません。

プログラム問合せ

プログラム問合せには、有効性、コンパイル結果および属性が含まれています。

プログラムの有効性

プログラム ハンドルが有効なプログラムを参照しているかどうかをチェックするには、**cgIsProgram()** を使用します。

```
CGbool cgIsProgram(CGprogram program);
```

コンパイル結果

特定のコンテキストについて最後にコールした **cgCreateProgram()** のコンパイル結果を問い合せるには、**cgGetLastListing()** を使用します。

```
const char* cgGetLastListing(CGcontext context);
```

コンテキストに対して **cgCreateProgram()** がコールされていない場合、**cgGetLastListing()** は 0 を返します。コールされている場合、通常はコマンドライン バージョンのコンパイラから取得される出力を含む文字列を返します。

プログラムの属性

プログラムが属するコンテキストを取得するには、**cgGetProgramContext()** を使用します。

```
CGcontext cgGetProgramContext(CGprogram program);
```

プログラムのコンパイル対象のプロファイルの取得は、**cgGetProgramProfile()** で行います。

```
CGprofile cgGetProgramProfile(CGprogram program);
```

cgGetProfile() と **cgGetProfileString()** の関数ペアでは、プロファイル列挙とそれに対応する文字列の間の一致を検索できます。

```
CGprofile   cgGetProfile(const char* profileString);
const char* cgGetProfileString(CGprofile profile);
```

cgGetProfile() に渡された文字列がどのプロファイルとも一致しない場合は、**CG_PROFILE_UNKNOWN** が返されます。

関数 **cgGetProgramString()** は、列挙 ***stringType*** の値に応じて、プログラムに関連する様々な文字列を取得します。

```
const char* cgGetProgramString(CGprogram program,
                               CGenum stringType);
```

変数 ***stringType*** の値は次のいずれかです。

- **CG_PROGRAM_SOURCE**: 元の Cg ソース プログラムが返されます。
- **CG_PROGRAM_ENTRY**: Cg ソース プログラムのメイン エントリ ポイントが返されます。
- **CG_PROGRAM_PROFILE**: プロファイル文字列が返されます。
- **CG_COMPILED_PROGRAM**: コンパイルされたプログラムが返されます。

コア Cg パラメータ

Cg には、パラメータの取得と問合せを行う関数があります。

パラメータの取得

パラメータの取得には、反復取得と直接取得があります。

反復取得

プログラムには、**cgGetFirstParameter()** および **cgGetNextParameter()** を使用して反復できるパラメータがあります。

```
CGparameter cgGetFirstParameter(CGprogram program,
                                CGenum namespace);
CGparameter cgGetNextParameter(CGparameter parameter);
```

cgGetFirstParameter() のコールでは、1 番目のパラメータが返されます。プログラムが無効であるか、パラメータを含んでいない場合、この関数は 0 を返します。パラメータがあった場合、**cgGetNextParameter()** はすぐ次の順番のパラメータを返すか、次のパラメータがない場合は 0 を返します。
cgGetFirstParameter() の ***namespace*** 引数には、この関数および後続の **cgGetNextParameter()** のコールで返されるパラメータのネームスペースを指定します。各パラメータは、そのスコープを定義する特定のネームスペースに属します。現在のところ、パラメータのスコープは、それが属するプログラムに限定されているため、***namespace*** に対して有効な値は **CG_PROGRAM** のみです。

注意： **将来的には、コンテキストなどの他のネームスペースを定義できるようになります。その場合、cgGetFirstParameter() および cgGetNextParameter() では、コンテキストのスコープ内にあるプログラムのすべてのパラメータを反復できます。**

次の例は、**program** という名前の有効なプログラムがある場合に、この 2 つの関数が通常どのように使用されるかを示したものです。

```
CGparameter parameter = cgGetFirstParameter(program,
                                            CG_PROGRAM);
while (parameter != 0) {
  /* Here is the code that handles the parameter */
  parameter = cgGetNextParameter(parameter);
}
```

これらの関数では、構造体パラメータ（**CG_STRUCT** 型）のフィールドや、配列パラメータ（**CG_ARRAY** 型）の要素にはアクセスできません。

構造体のフィールドにアクセスするには、**cgGetFirstStructParameter()** を **cgGetNextParameter()** とともに使用します。

```
CGparameter cgGetFirstStructParameter(
              CGparameter parameter);
```

parameter の型が **CG_STRUCT** ではない場合、**cgGetFirstStructParameter()** は 0 を返します。

配列の要素にアクセスするには、**cgGetArrayDimension()**、**cgGetArraySize()**、**cgGetArrayParameter()** および **cgGetNextParameter()** を使用します。

```
int cgGetArrayDimension(CGparameter parameter);
int cgGetArraySize(CGparameter parameter, int dimension);
CGparameter cgGetArrayParameter(CGparameter parameter,
                                int index);
```

この 3 つの関数は、***parameter*** の型が **CG_ARRAY** ではない場合に 0 を返します。関数 **cgGetArrayDimension()** は配列の次元を返します。戻り値は、**float4 array[10]** の場合は 1、**float4 array[10][100]** の場合は 2 などのようになります。次に、**cgGetArraySize()** は各次元のサイズを返します。たとえば、**float4 array[10][100]** の場合、**cgGetArraySize(array,0)** とすれば 10 を返し、**cgGetArraySize(array,1)** とすれば 100 を返します。配列 **anArray** には **cgGetArraySize(anArray,0)** 要素があります。次元が 1 より大きい場合は、これらの要素自体が配列です。

次の例は、**program** という名前の有効なプログラムがある場合に、これらすべての反復関数が通常どのように使用されるかを示したものです。

```
void IterateProgramParameters(CGprogram program) {
  RecurseProgramParameters(cgGetFirstParameter(program,
                                               CG_PROGRAM));
}

void RecurseProgramParameters(CGparameter parameter) {
  if (parameter == 0)
    return;
  do {
    switch(cgGetParameterType(parameter)) {
      case CG_STRUCT:
        RecurseProgramParameters(
          cgGetFirstStructParameter(parameter));
        break;
      case CG_ARRAY:
        int arraySize = cgGetArraySize(parameter, 0);
        for (int i = 0; i < arraySize; ++i)
          RecurseProgramParameters(
            cgGetArrayParameter(parameter, i));
        break;
      default:
```

```
        /* Here is the code that handles the parameter */
        break;
    }
  } while((parameter = cgGetNextParameter(parameter))!= 0);
}
```

構造体と配列に関してパラメータがどのように編成されているかを知る必要がない場合は、**cgGetFirstLeafParameter()** および **cgGetNextLeafParameter()** を使用してこれらすべてを反復することもできます。

```
CGparameter cgGetFirstLeafParameter(CGprogram program,
                                    CGenum namespace);
CGparameter cgGetNextLeafParameter(CGparameter parameter);
```

これらの関数は、プログラムへの入力である単純なパラメータ、構造体のフィールドおよび配列の要素をすべて反復します。パラメータの順序については何も保証されません。

直接取得

プログラムのパラメータは、**cgGetNamedParameter()** でパラメータの名前を使用することによって、直接取得することができます。

```
CGparameter cgGetNamedParameter(CGprogram program,
                                const char* name);
```

name に対応するパラメータがプログラムにない場合、**cgGetNamedParameter()** は 0 を返します。

構造体のフィールドまたは配列の要素の取得には Cg 構文が使用されます。例として、次のコードを検討してみます。

```
struct FooStruct {
  float4 A;
  float4 B;
};
struct BarStruct {
  FooStruct Foo[2];
};
void main(BarStruct Bar[3]) {
  // ...
}
```

対応するパラメータを取得するための有効な名前は、次のようになります。

```
"Bar"
"Bar[1]"
"Bar[1].Foo"
"Bar[1].Foo[0]"
"Bar[1].Foo[0].B"
```

パラメータの問合せ

パラメータの問合せには、有効性、参照および属性が含まれます。

パラメータの有効性

パラメータ ハンドルが有効なパラメータを参照しているかどうかをチェックするには、関数 **cgIsParameter()** を使用します。

```
CGbool cgIsParameter(CGparameter parameter);
```

パラメータ ハンドルは、対応するプログラムまたはプログラムのコンテキストが破棄されると無効になります。

パラメータの参照

元の Cg ソース コードで参照されるパラメータは、コンパイラによってコンパイルされたプログラムに含まれないように最適化できます。この場合、アプリケーションは単純にこのパラメータを無視し、値を設定しません。パラメータが最終的なコンパイル済みプログラムで実際に使用されているかどうかをチェックするには、**cgIsParameterReferenced()** をコールします。

```
CGbool cgIsParameterReferenced(CGparameter parameter);
```

参照されていないパラメータの値を設定した場合でも、エラーは生成されません。

パラメータの属性

パラメータに対応するプログラムを取得するには、**cgGetParameterProgram()** を使用します。

```
CGprogram cgGetParameterProgram(CGparameter parameter);
```

パラメータが varying 型、uniform 型または定数のいずれであるかを判断するには、**cgGetParameterVariability()** を使用します。

```
CGenum cgGetParameterVariability(CGparameter parameter);
```

このコールでは、パラメータが varying 型であれば **CG_VARYING**、uniform 型であれば **CG_UNIFORM**、定数であれば **CG_CONSTANT** を返します。定数パラメータは、コンパイルされたプログラムの存続中は値が変更されないパラメータであるため、その値を変更するにはプログラムを再コンパイルする必要があります。一部のプロファイルでは、リテラル定数値に対応するものをコンパイラがコードに追加する必要があります。

パラメータの方向を取得するには、**cgGetParameterDirection()** を使用します。

```
CGenum cgGetParameterDirection(CGparameter parameter);
```

この関数は、パラメータが入力パラメータであれば **CG_IN**、出力パラメータであれば **CG_OUT**、入力および出力パラメータであれば **CG_INOUT** を返します。

パラメータの型は **cgGetParameterType()** によって取得されます。

```
CGtype cgGetParameterType(CGparameter parameter);
```

5 つの型のいずれかが返されます。(1)パラメータが構造体の場合は **CG_STRUCT**、(2) パラメータが配列の場合は **CG_ARRAY**、(3) パラメータが **half** ベース型の場合は **CG_HALF***、(4) パラメータが **float** ベース型の場合は **CG_FLOAT***、(5) パラメータが **sampler** ベース型の場合は **CG_SAMPLER*** を返します。

cgGetType() と **cgGetTypeString()** の関数ペアは、型の列挙とそれに対応する文字列の間の一致を示します。

```
CGtype cgGetType(const char* typeString);
const char* cgGetTypeString(CGtype type);
```

cgGetType() に渡された文字列がどの型とも一致しない場合は、**CG_UNKNOWN_TYPE** が返されます。

関数 **cgGetParameterName()** はパラメータ名を取得します。

```
const char* cgGetParameterName(CGparameter parameter);
```

パラメータ セマンティクス文字列を取得するには**cgGetParameterSemantic()**を使用します。

```
const char* cgGetParameterSemantic(CGparameter parameter);
```

パラメータにセマンティクスがない場合は、空の文字列が返されます。

定義済みセマンティクス（**POSITION**、**COLOR** など）とハードウェア リソース（レジスタ、テクスチャ ユニットなど）の間には、1 対 1 の対応があります。Cg ランタイムでは、ハードウェア リソースは **CGresource** 型で表され、**cgGetParameterResource()** はパラメータに割り当てられているリソースを取得します。

```
CGresource cgGetParameterResource(CGparameter parameter);
```

パラメータにリソースが関連付けられていない場合、**cgGetParameterResource()** は **CG_UNDEFINED** を返します。

cgGetResource() と **cgGetResourceString()** の 2 つの関数では、リソース列挙とそれに対応する文字列の間の一致を判断できます。

```
CGresource  cgGetResource(const char* resourceString);
const char* cgGetResourceString(CGresource resource);
```

cgGetResource() に渡された文字列がどのリソースとも一致しない場合は、**CG_UNDEFINED** が返されます。

cgGetParameterBaseResource() を使用すると、Cg プログラムのパラメータのベース リソースを取得できます。

```
CGresource cgGetParameterBaseResource(
              CGparameter parameter);
```

ベース リソースは、連続したリソースのセット内の最初のリソースです。たとえば、ある特定のパラメータに **CG_TEXCOORD7** と等しいリソースがある場合、そ

のベース リソースは **CG_TEXCOORD0** です。ベース リソースを持つのは、名前が数値で終わるリソースを持つパラメータのみです。その他すべてのパラメータは、**cgGetParameterBaseResource()** がコールされたときに **CG_UNDEFINED** を返します。

リソースの数値部分は、**cgGetParameterResourceIndex()** で取得できます。

```
unsigned long cgGetParameterResourceIndex(
                    CGparameter parameter);
```

たとえば、ある特定のパラメータのリソースが **CG_TEXCOORD7** の場合、**cgGetParameterResourceIndex()** は 7 を返します。

cgGetParameterValues() 関数は、uniform 型パラメータのデフォルトまたは定数値を取得します。

```
const double* cgGetParameterValues(CGparameter parameter,
                CGenum valueType, int* numberOfValuesReturned);
```

valueTypeが**CG_DEFAULT**と等しい場合はデフォルト値を取得し、***valueType***が**CG_CONSTANT** と等しい場合は定数値を取得します。値の構成要素は、**double** 型の要素を含む配列へのポインタとして、行優先の順序で返されます。**cgGetParameterValues()** のコール後、***numberOfValuesReturned*** は配列で有効な構成要素数を指します。

コア Cg エラー

コア Cg ランタイムは、エラー コードを含むグローバル変数を設定することによってエラーをレポートします。このエラーおよび対応するエラー文字列は、次のようにして問い合せます。

```
CGerror error = cgGetError();
const char* errorString = cgGetErrorString(error);
```

エラーが発生するたびに、コア Cg ランタイムは、アプリケーションでオプションとして用意されているコールバック関数もコールします。通常、このコールバック関数は **cgGetError()** をコールします。

```
void MyErrorCallback() {
  const char* errorString = cgGetErrorString(cgGetError());
}
cgSetErrorCallback(MyErrorCallback);
```

次に、コア Cg ランタイムに固有のすべての **CGerror** エラーのリストを示します。

- **CG_NO_ERROR**: エラーが発生していない場合に返されます。
- **CG_COMPILER_ERROR**: コンパイラがエラーを生成した場合に返されます。**cgGetLastListing()** をコールし、実際のコンパイラ エラーの詳細を取得する必要があります。
- **CG_INVALID_PARAMETER_ERROR**: 使用されているパラメータが無効な場合に返されます。

- **CG_INVALID_PROFILE_ERROR**: プロファイルがサポートされていない場合に返されます。
- **CG_INVALID_VALUE_TYPE_ERROR**: 不明な値の型がパラメータに代入されている場合に返されます。
- **CG_NOT_MATRIX_PARAM_ERROR**: パラメータが行列型ではない場合に返されます。
- **CG_INVALID_ENUMERANT_ERROR**: 列挙パラメータの値が無効な場合に返されます。
- **CG_NOT_4x4_MATRIX_ERROR**: パラメータが4×4の行列型である必要がある場合に返されます。
- **CG_FILE_READ_ERROR**: ファイルを読み取れない場合に返されます。
- **CG_FILE_WRITE_ERROR**: ファイルに書き込めない場合に返されます。
- **CG_MEMORY_ALLOC_ERROR**: メモリ割当てに失敗した場合に返されます。
- **CG_INVALID_CONTEXT_HANDLE_ERROR**: 無効なコンテキスト ハンドルが使用されている場合に返されます。
- **CG_INVALID_PROGRAM_HANDLE_ERROR**: 無効なプログラム ハンドルが使用されている場合に返されます。
- **CG_INVALID_PARAM_HANDLE_ERROR**: 無効なパラメータ ハンドルが使用されている場合に返されます。
- **CG_UNKNOWN_PROFILE_ERROR**: 指定されたプロファイルが不明な場合に返されます。
- **CG_VAR_ARG_ERROR**: 変数引数が誤って指定されている場合に返されます。
- **CG_INVALID_DIMENSION_ERROR**: 次元値が無効な場合に返されます。
- **CG_ARRAY_PARAM_ERROR**: パラメータが配列である必要がある場合に返されます。
- **CG_OUT_OF_ARRAY_BOUNDS_ERROR**: 配列へのインデックスが範囲外にある場合に返されます。

API 固有の Cg ランタイム

各 API に固有の Cg ランタイムには、その API 上でのアプリケーションへの Cg の統合を簡単にするために、コア Cg ランタイムの上に追加の関数セットが用意されています。これらの関数セットは、基本的に、次の機能を提供するためにコア ランタイム データ構造体とAPIデータ構造体との間をインタフェースします。

- パラメータ値の設定： テクスチャ、行列、配列、ベクトルおよびスカラ型は API ごとに異なる方法で処理され、異なるデータ構造体を持つため、区別して扱われます。

- プログラムの実行： プログラムの実行は、プログラムのロード（Cg コンパイラの結果を API に渡す）と、プログラムのバインド（後続の描画コールで実行されるようにプログラムを設定）に分かれます。これは、この 2 つの操作が通常は異なる時点で実行されるためです。プログラムは、再コンパイルされるたびにロードされ、特定の描画コールに対して実行する必要があるたびにバインドされます。

パラメータ シャドーイング

uniform 型パラメータの値が OpenGL Cg ランタイムの関数によって設定される場合、この値は実際には Cg または OpenGL ランタイムにより内部的に格納（シャドーイング）されるため、プログラムを実行するたびに再設定する必要はありません。この動作をパラメータ シャドーイングと呼びます。

Direct3D Cg ランタイムの拡張インタフェース（69 ページの「Direct3D 拡張インタフェース」を参照）が使用されている場合、パラメータ シャドーイングはプログラムごとに有効 / 無効を設定できます。パラメータ シャドーイングが特定のプログラムに対して無効にされ、そのプログラムの uniform 型パラメータの値が Direct3D Cg ランタイムの関数により設定される場合、その値は GPU の定数メモリ（すべての uniform 型パラメータの値を含むメモリ）に即時にダウンロードされます。パラメータ シャドーイングが有効になっている場合は、値がシャドーイングされ、その設定時に Direct3D コールは行われません。プログラムがバインドされるときにのみ、すべてのパラメータが定数メモリに実際にダウンロードされます。したがって、プログラムのバインド後に設定されたパラメータ値は、次にプログラムがバインドされるまではプログラムの実行中に使用されません。パラメータ シャドーイングは、テクスチャ ステート ステージとテクスチャ モードを含むすべてのパラメータ設定に適用されます。

パラメータ シャドーイングを無効にすると、ランタイムが消費するメモリは減少しますが、アプリケーションは、プログラムをアクティブにするたびに、定数メモリに正しい値がすべて含まれているかどうかの確認作業を行うことになります。

OpenGL Cg ランタイム

ここでは、OpenGL Cg ランタイムのパラメータの設定とプログラムの実行について説明します。

OpenGL でのパラメータの設定

OpenGL の規則に従い、次に説明する関数の多くには、**`float`** 値を操作するバージョン（**`f`** でマーク）と、**`double`** 値を操作するバージョン（**`d`** でマーク）の 2 つのバージョンがあります。

uniform 型スカラ パラメータと uniform 型ベクトル パラメータの設定

スカラ パラメータまたはベクトル パラメータの値を設定するには、**cgGLSetParameter** 関数を使用します。

```
void cgGLSetParameter1f(CGparameter parameter,  float x);
void cgGLSetParameter1fv(CGparameter parameter,
                         const float* array);
void cgGLSetParameter1d(CGparameter parameter,  double x);
void cgGLSetParameter1dv(CGparameter parameter,
                         const double* array);

void cgGLSetParameter2f(CGparameter parameter,  float x,
                        float y);
void cgGLSetParameter2fv(CGparameter parameter,
                         const float* array);
void cgGLSetParameter2d(CGparameter parameter,  double x,
                        double y);
void cgGLSetParameter2dv(CGparameter parameter,
                         const double* array);

void cgGLSetParameter3f(CGparameter parameter, float x,
                        float y, float z);
void cgGLSetParameter3fv(CGparameter parameter,
                         const float* array);
void cgGLSetParameter3d(CGparameter parameter, double x,
                        double y, double z);
void cgGLSetParameter3dv(CGparameter parameter,
                         const double* array);

void cgGLSetParameter4f(CGparameter parameter, float x,
                        float y, float z, float w);
void cgGLSetParameter4fv(CGparameter parameter,
                         const float* array);
void cgGLSetParameter4d(CGparameter parameter, double x,
                        double y, double z, double w);
void cgGLSetParameter4dv(CGparameter parameter,
                         const double* array);
```

これらの関数の名前にある数字は、関数で設定されるスカラ値の個数を示しています。**v** 接尾辞は、個々の引数ではなく値の配列を操作する関数に付いています。

パラメータで要求されている個数よりも多くの値が設定されている場合、余分な値は無視されます。パラメータで要求されている個数よりも少ない値が設定されている場合は、最後の値が展開されます。**cgGLSetParameter** 関数は、uniform 型パラメータまたは varying 型パラメータに対してコールできます。varying 型パラメータに対してコールされた場合は、適切な即時モードの OpenGL エントリポイントがコールされます。

対応するパラメータ値取得関数は次のとおりです。

```
cgGLGetParameter1f(CGparameter parameter, float*  array);
cgGLGetParameter1d(CGparameter parameter, double* array);
cgGLGetParameter2f(CGparameter parameter, float*  array);
cgGLGetParameter2d(CGparameter parameter, double* array);
cgGLGetParameter3f(CGparameter parameter, float*  array);
cgGLGetParameter3d(CGparameter parameter, double* array);
cgGLGetParameter4f(CGparameter parameter, double* array);
cgGLGetParameter4d(CGparameter parameter, type*   array);
```

uniform 型行列パラメータの設定

cgGLSetMatrixParameter 関数は、4 × 4 の行列のみを設定します。

```
void cgGLSetMatrixParameterfr(CGparameter parameter,
                              const float* matrix);
void cgGLSetMatrixParameterfc(CGparameter parameter,
                              const float* matrix);
void cgGLSetMatrixParameterdr(CGparameter parameter,
                              const double* matrix);
void cgGLSetMatrixParameterdc(CGparameter parameter,
                              const double* matrix);
```

r 接尾辞は、行列が行優先の順序で配置されていることを想定する関数に付けられ、**c** 接尾辞は、行列が列優先の順序で配置されていることを想定する関数に付けられます。

対応するパラメータ値取得関数は次のとおりです。

```
void cgGLGetMatrixParameterfr(CGparameter parameter,
                              float* matrix);
void cgGLGetMatrixParameterfc(CGparameter parameter,
                              float* matrix);
void cgGLGetMatrixParameterdr(CGparameter parameter,
                              double* matrix);
void cgGLGetMatrixParameterdc(CGparameter parameter,
                              double* matrix);
```

OpenGL 4×4ステートの行列を設定するには**cgGLSetStateMatrixParameter()** を使用します。

```
void cgGLSetStateMatrixParameter(CGparameter parameter,
       GLenum stateMatrixType, GLenum transform);
```

変数 ***stateMatrixType*** は、パラメータの設定に使用するステート行列を次のように指定する列挙型です。

- 現在のモデルビュー行列を指定する場合は、**CG_GL_MODELVIEW_MATRIX**
- 現在の射影行列を指定する場合は、**CG_GL_PROJECTION_MATRIX**
- 現在のテクスチャ行列を指定する場合は、**CG_GL_TEXTURE_MATRIX**

- 連結したモデルビュー行列と射影行列を指定する場合は、**CG_GL_MODELVIEW_PROJECTION_MATRIX**

変数 ***transform*** は、パラメータ値の設定に使用する前にステートの行列に適用する変換を、次のように指定する列挙型です。

- 変換を適用しないことを指定する場合は、**CG_GL_MATRIX_IDENTITY**
- 行列を転置する場合は、**CG_GL_MATRIX_TRANSPOSE**
- 行列を逆にする場合は、**CG_GL_MATRIX_INVERSE**
- 行列に逆と転置の両方を行う場合は、**CG_GL_MATRIX_INVERSE_TRANSPOSE**

uniform 型スカラ、ベクトルおよび行列パラメータの配列の設定

uniform 型スカラまたはベクトル パラメータの配列の値を設定するには、**cgGLSetParameterArray** 関数を使用します。

```
void cgGLSetParameterArray1f(CGparameter parameter,
       long startIndex, long numberOfElements,
       const float* array);
void cgGLSetParameterArray1d(CGparameter parameter,
       long startIndex, long numberOfElements,
       const double* array);
void cgGLSetParameterArray2f(CGparameter parameter,
       long startIndex, long numberOfElements,
       const float* array);
void cgGLSetParameterArray2d(CGparameter parameter,
       long startIndex, long numberOfElements,
       const double* array);
void cgGLSetParameterArray3f(CGparameter parameter,
       long startIndex, long numberOfElements,
       const float* array);
void cgGLSetParameterArray3d(CGparameter parameter,
       long startIndex, long numberOfElements,
       const double* array);
void cgGLSetParameterArray4f(CGparameter parameter,
       long startIndex, long numberOfElements,
       const float* array);
void cgGLSetParameterArray4d(CGparameter parameter,
       long startIndex, long numberOfElements,
       const double* array);
```

これらの関数の名前にある数字は、**1** が **float1** の配列、**2** が **float2** の配列などのように、パラメータ配列要素の型を示します。変数 ***startIndex*** および ***numberOfElements*** では、配列パラメータのどの要素を設定するかを指定します。***startIndex*** から ***startIndex+numberOfElements*** までの、***numberOfElements*** 個の要素を指定したことになります。***numberOfElements*** に 0 の値を渡すと、インデックス ***startIndex*** から配列の最後の有効なインデックス（つまり **cgGetArraySize(parameter,0)-1**）までのすべての値を設定します。これは、

numberOfElements を **cgGetArraySize(parameter,0)-startIndex** に設定するのと等価です。パラメータ ***array*** は、スカラ値の配列です。**cgGLSetParameterArray1** 関数に対しては ***numberOfElements***、**cgGLSetParameterArray2** 関数に対しては **2*numberOfElements** の長さが必要、というようになります。

対応するパラメータ値取得関数は次のとおりです。

```
void cgGLGetParameterArray1f(CGparameter parameter,
      long startIndex, long numberOfElements, float* array);
void cgGLGetParameterArray1d(CGparameter parameter,
      long startIndex, long numberOfElements, double* array);
void cgGLGetParameterArray2f(CGparameter parameter,
      long startIndex, long numberOfElements, float* array);
void cgGLGetParameterArray2d(CGparameter parameter,
      long startIndex, long numberOfElements, double* array);
void cgGLGetParameterArray3f(CGparameter parameter,
      long startIndex, long numberOfElements, float* array);
void cgGLGetParameterArray3d(CGparameter parameter,
      long startIndex, long numberOfElements, double* array);
void cgGLGetParameterArray4f(CGparameter parameter,
      long startIndex, long numberOfElements, float* array);
void cgGLGetParameterArray4d(CGparameter parameter,
      long startIndex, long numberOfElements, double* array);
```

uniform 型行列パラメータの配列の値を設定する、類似の関数もあります。

```
void cgGLSetMatrixParameterArrayfr(CGparameter parameter,
       long startIndex, long numberOfElements,
       const float* array);
void cgGLSetMatrixParameterArrayfc(CGparameter parameter,
       long startIndex, long numberOfElements,
       const float* array);
void cgGLSetMatrixParameterArraydc(CGparameter parameter,
       long startIndex, long numberOfElements,
       const double* array);
void cgGLSetMatrixParameterArraydc(CGparameter parameter,
       long startIndex, long numberOfElements,
       const double* array);
```

これらの値を問い合せる関数は次のとおりです。

```
void cgGLGetMatrixParameterArrayfr(CGparameter parameter,
      long startIndex, long numberOfElements, float* array);
void cgGLGetMatrixParameterArrayfc(CGparameter parameter,
      long startIndex, long numberOfElements, float* array);
void cgGLGetMatrixParameterArraydc(CGparameter parameter,
      long startIndex, long numberOfElements, double* array);
```

```
void cgGLGetMatrixParameterArraydc(CGparameter parameter,
      long startIndex, long numberOfElements, double* array);
```

c および **r** 接尾辞の意味は、**cgGLSetMatrixParameter** 関数の場合と同じです。

varying 型パラメータの設定

フラグメント プログラムの varying 型パラメータの値は、GPU により実行される三角形上での値の補間の結果として設定されるため、アプリケーションで設定されるのは頂点プログラムの varying 型パラメータの値のみです。

頂点プログラムの varying 型パラメータの設定には、2 つの手順が必要です。

最初の手順では、各頂点の値を含む配列へのポインタを渡します。この操作は、**cgGLSetParameterPointer()** を使用して行います。

```
void cgGLSetParameterPointer(CGparameter parameter,
       GLint size, GLenum type, GLsizei stride,
       GLvoid* array);
```

変数 ***size*** は、***array*** に格納されている頂点ごとの値の個数を示します。これは 1、2、3 または 4 のいずれかです。設定されている値の個数がパラメータで要求されている個数よりも少ない場合、指定されていない値はデフォルトの 0（**x**、**y** および **z** の場合）または 1（**w** の場合）に設定されます。

列挙型 ***type*** には、***array*** に格納されている値のデータ型（**GL_SHORT**、**GL_INT**、**GL_FLOAT** または **GL_DOUBLE**）を指定します。

パラメータ***stride***は、連続する2つの頂点の間のバイト オフセットです。***stride*** に 0 の値を渡すことは、***size*** に ***type*** のサイズ（バイト数）を乗算したバイト オフセットを渡すことと同じです。つまり、連続する 2 つの頂点値の間にギャップがないことを意味します。***array*** の最小サイズは、描画される三角形で指定されている最大の頂点インデックスによって暗黙的に定義されることに注意してください。

2 番目の手順では、特定の描画コールに対して varying 型パラメータを有効にします。

```
void cgGLEnableClientState(CGparameter parameter);
```

対応する無効化の関数は次のとおりです。

```
void cgGLDisableClientState(CGparameter parameter);
```

頂点の varying 型パラメータを設定する別の方法は、**cgGLSetParameter** 関数を使用することです。varying 型パラメータに対して **cgGLSetParameter** 関数がコールされた場合は、適切な即時モードの OpenGL エントリ ポイントがコールされます。**cgGLGetParameter** 関数は、varying 型パラメータに適用されません。

sampler パラメータの設定

sampler パラメータの設定には、2 つの手順が必要です。

最初の手順では、使用する sampler パラメータに OpenGL テクスチャ オブジェクトを割り当てます。

```
void cgGLSetTextureParameter(CGparameter parameter,
                             GLuint textureName);
```

textureName は OpenGL テクスチャ名です。

2 番目の手順では、特定の描画コールに対して sampler パラメータを有効にします。

```
void cgGLEnableTextureParameter(CGparameter parameter);
```

関数 **cgGLEnableTextureParameter()** は、**cgGLSetTextureParameter()** の後、実際の描画コールの前にコールする必要があります。

対応する無効化の関数は次のとおりです。

```
void cgGLDisableTextureParameter(CGparameter parameter);
```

sampler パラメータに割り当てられているテクスチャ オブジェクトは、次の関数を使用して取得できます。

```
GLuint cgGLGetTextureParameter(CGparameter parameter);
```

samplerパラメータに関連付けられているテクスチャ ユニットのOpenGL列挙は、次の関数を使用して取得できます。

```
GLenum cgGLGetTextureEnum(CGparameter parameter);
```

返される列挙の形式は **GL_TEXTURE#_ARB** で、**#** はテクスチャ ユニット インデックスです。

OpenGL プロファイル サポート

使用できる OpenGL 拡張機能に応じて、頂点プログラムまたはフラグメント プログラムに可能なかぎり高度なプロファイルを提供する便利な関数が用意されています。

```
CGprofile cgGLGetLatestProfile(CGGLenum profileType);
```

パラメータ ***profileType*** は、**CG_GL_VERTEX** または **CG_GL_FRAGMENT** になります。**cgCreateProgram()** または **cgCreateProgramFromFile()** と連携して関数 **cgGLGetLatestProfile()** を使用すると、可能なかぎり高度な頂点プロファイルおよびフラグメント プロファイルがコンパイルに使用されるようになります。これによって、これらのプロファイルがアプリケーションの作成時に存在していなかった場合でも、Cg プログラムは実行時に使用可能な最適プロファイルに対して自動的にコンパイルされるため、アプリケーションを将来の使用に備えることができます。最適なコンパイルを可能にするもう 1 つの関数として、**cgGLSetOptimalOptions()** もあります。この関数は、**cgCreateProgram()** または **cgCreateProgramFromFile()** に渡される引数リストに追加される暗黙のコンパイラ引数を設定します。

```
void cgGLSetOptimalOptions(CGprofile profile);
```

OpenGL プログラムの実行

すべてのプログラムは、バインドする前にロードする必要があります。プログラムのロードには、**cgGLLoadProgram()** を使用します。

```
void cgGLLoadProgram(CGprogram program);
```

プログラムのバインドは、プロファイルが有効になっている場合にのみ動作します。この処理は、プログラム プロファイルを指定して **cgGLEnableProfile()** をコールすることにより行います。

```
void cgGLEnableProfile(CGprofile profile);
```

バインド自体は、**cgGLBindProgram()** を使用して行います。

```
void cgGLBindProgram(CGprogram program);
```

ある特定の時点でバインドできるのは1つの頂点プログラムと1つのフラグメント プログラムのみであるため、プログラムをバインドすると、そのタイプの他のプログラムは暗黙的にバインド解除されます。

プロファイルを無効にするには、**cgGLDisableProfile()** を使用します。

```
void cgGLDisableProfile(CGprofile profile);
```

一部のプロファイルは、一部のシステムでサポートされないことがあります。たとえば、ある特定のプロファイルは、必要な OpenGL 拡張機能が使用可能ではない場合にはサポートされません。プロファイルがサポートされているかどうかは、**cgGLIsProfileSupported()** を使用して確認できます。

```
CGbool cgGLIsProfileSupported(CGprofile profile);
```

この関数は、***profile*** がサポートされている場合に **CG_TRUE** を返し、サポートされていない場合に **CG_FALSE** を返します。

OpenGL プログラムの例

ここで示すコードでは、OpenGL Cg インタフェースの関数を使用して、Cg プログラムを OpenGL で動作させる方法を示します。次に示す頂点プログラムとフラグメント プログラムは、54 ページの「OpenGL アプリケーション」で使用されています。

OpenGL 頂点プログラム

次のCgコードは、**VertexProgram.cg**というファイル内にあると想定しています。

```
void VertexProgram(
  in float4 position   : POSITION,
  in float4 color      : COLOR0,
  in float4 texCoord   : TEXCOORD0,
  out float4 positionO : POSITION,
  out float4 colorO    : COLOR0,
  out float4 texCoordO : TEXCOORD0,
  const uniform float4x4 ModelViewMatrix )
```

```
{
  positionO = mul(position, ModelViewMatrix);
  colorO = color;
  texCoordO = texCoord;
}
```

OpenGL フラグメント プログラム

次の Cg コードは、**FragmentProgram.cg** というファイル内にあると想定しています。

```
void FragmentProgram(
  in float4 color     : COLOR0,
  in float4 texCoord  : TEXCOORD0,
  out float4 colorO   : COLOR0,
  const uniform sampler2D BaseTexture,
  const uniform float4 SomeColor)
{
  colorO = color * tex2D(BaseTexture, texCoord) + SomeColor;
}
```

OpenGL アプリケーション

このCコードは、前述の頂点プログラムとフラグメント プログラムをアプリケーションにリンクします。

```
#include <cg/cg.h>
#include <cg/cgGL.h>

float* vertexPositions;  // Initialized somewhere else
float* vertexColors;     // Initialized somewhere else
float* vertexTexCoords;  // Initialized somewhere else
GLuint texture;          // Initialized somewhere else
float constantColor[];   // Initialized somewhere else
CGcontext context;
CGprogram vertexProgram, fragmentProgram;
CGprofile vertexProfile, fragmentProfile;
CGparameter position, color, texCoord, baseTexture, someColor,
            modelViewMatrix;

// Called at initialization
void CgGLInit()
{
  // Create context
  context = cgCreateContext();

  // Initialize profiles and compiler options
  vertexProfile = cgGLGetLatestProfile(CG_GL_VERTEX);
```

```
  cgGLSetOptimalOptions(vertexProfile);
  fragmentProfile = cgGLGetLatestProfile(CG_GL_FRAGMENT);
  cgGLSetOptimalOptions(fragmentProfile);

  // Create the vertex program
  vertexProgram = cgCreateProgramFromFile(
                    context, CG_SOURCE, "VertexProgram.cg",
                    vertexProfile, "VertexProgram", 0);

  // Load the program
  cgGLLoadProgram(vertexProgram);

  // Create the fragment program
  fragmentProgram = cgCreateProgramFromFile(
                    context, CG_SOURCE, "FragmentProgram.cg",
                    pixelProfile, "FragmentProgram", 0);

  // Load the program
  cgGLLoadProgram(fragmentProgram);

  // Grab some parameters.
  position = cgGetNamedParameter(vertexProgram, "position");
  color = cgGetNamedParameter(vertexProgram, "color");
  texCoord = cgGetNamedParameter(vertexProgram, "texCoord");
  modelViewMatrix = cgGetNamedParameter(vertexProgram,
                                        "ModelViewMatrix");
  baseTexture = cgGetNamedParameter(fragmentProgram,
                                    "BaseTexture");
  someColor = cgGetNamedParameter(fragmentProgram,
                                  "SomeColor");

  // Set parameters that don't change:
  // They can be set only once because of parameter shadowing.
  cgGLSetTextureParameter(baseTexture, texture);
  cgGLSetParameter4fv(someColor, constantColor);
}

// Called to render the scene
void Display()
{
  // Set the varying parameters
  cgGLEnableClientState(position);
  cgGLSetParameterPointer(position, 3, GL_FLOAT, 0,
                          vertexPositions);
  cgGLEnableClientState(color);
```

```
  cgGLSetParameterPointer(color, 1, GL_FLOAT, 0,
                          vertexColors);
  cgGLEnableClientState(texCoord);
  cgGLSetParameterPointer(texCoord, 2, GL_FLOAT, 0,
                          vertexTexCoords);

  // Set the uniform parameters that change every frame
  cgGLSetStateMatrixParameter(modelViewMatrix,
                            CG_GL_MODELVIEW_PROJECTION_MATRIX,
                            CG_GL_MATRIX_IDENTITY);

  // Enable the profiles
  cgGLEnableProfile(vertexProfile);
  cgGLEnableProfile(fragmentProfile);

  // Bind the programs
  cgGLBindProgram(vertexProgram);
  cgGLBindProgram(fragmentProgram);

  // Enable texture
  cgGLEnableTextureParameter(baseTexture);

  // Draw scene
  // ...

  // Disable texture
  cgGLDisableTextureParameter(baseTexture);

  // Disable the profiles
  cgGLDisableProfile(vertexProfile);
  cgGLDisableProfile(fragmentProfile);

  // Set the varying parameters
  cgGLDisableClientState(position);
  cgGLDisableClientState(color);
  cgGLDisableClientState(texCoord);
}

// Called before application shuts down
void CgShutdown()
{
  // This frees any runtime resource.
  cgDestroyContext(context);
}
```

OpenGL エラー レポート

次に、OpenGL Cg ランタイムに固有の **CGerror** エラーのリストを示します。

- **CG_PROGRAM_LOAD_ERROR**: プログラムをロードできない場合に返されます。
- **CG_PROGRAM_BIND_ERROR**: プログラムをバインドできない場合に返されます。
- **CG_PROGRAM_NOT_LOADED_ERROR**: 操作を行う前にプログラムをロードする必要がある場合に返されます。
- **CG_UNSUPPORTED_GL_EXTENSION_ERROR**: 操作を実行するために、サポートされていない Open GL 拡張機能が必要な場合に返されます。

任意の OpenGL Cg ランタイム関数は、Cg 固有のエラーに加えて OpenGL エラーを生成することがあります。これらのエラーは、OpenGL アプリケーションの場合と同様、Cg で **glGetError()** を使用してチェックされます。

Direct3D Cg ランタイム

Direct3D Cg ランタイムは、2 つのインタフェースから構成されます。

- 最小インタフェース: このインタフェースは、Direct3D コールを内部では行いません。アプリケーション側で Direct3D コードを保持することが望ましい場合に使用する必要があります。
- 拡張インタフェース : このインタフェースは、プログラムとパラメータの高度な管理を行い、必要な Direct3D コールを内部的に行います。Cg ランタイムで Direct3D シェーダを管理することが望ましい場合に使用する必要があります。

Direct3D 最小インタフェース

最小インタフェースでは、コア ランタイムで提供された情報を Direct3D 固有の情報に変換するために便利な関数が提供されます。

頂点宣言

Direct3D では、頂点シェーダ入力レジスタと、アプリケーションによりデータ ストリームとして提供されるデータとの間にマッピングを確立する頂点宣言を指定する必要があります。Direct3D 9 では、この頂点宣言は頂点シェーダと同じ方法で現在のステートにバインドされます（詳細な説明は、**IDirect3DDevice9::CreateVertexDeclaration()** および **IDirect3DDevice9::SetVertexDeclaration()** に関する Direct3D 9 のドキュメントを参照してください）。Direct3D 8 では、頂点宣言は、頂点シェーダの作成時に要求されます（詳細は、**IDirect3DDevice8::CreateVertexShader()** に関する Direct3D 8 のドキュメントを参照してください）。

データ ストリームは、基本的にデータ構造体の配列です。各構造体は、ストリームの頂点フォーマットと呼ばれる特定の型です。次に、Direct3D 9 の頂点宣言の例を示します。

```
const D3DVERTEXELEMENT9 declaration[] = {
  { 0, 0 * sizeof(float),
    D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
    D3DDECLUSAGE_POSITION, 0 }, // Position
  { 0, 3 * sizeof(float),
    D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
    D3DDECLUSAGE_NORMAL, 0 }, // Normal
  { 0, 8 * sizeof(float),
    D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT,
    D3DDECLUSAGE_TEXCOORD, 0 }, // Base texture
  { 1, 0 * sizeof(float),
    D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
    D3DDECLUSAGE_TEXCOORD, 1 }, // Tangent
  D3DD3CL_END()
};
```

次に、Direct3D 8 の頂点宣言の例を示します。

```
const DWORD declaration[] = {
  D3DVSD_STREAM(0),
  D3DVSD_REG(D3DVSDE_POSITION, D3DVSDT_FLOAT3), // Position
  D3DVSD_REG(D3DVSDE_NORMAL, D3DVSDT_FLOAT3),   // Normal
  D3DVSD_SKIP(2),      // Skip the diffuse and specular color
  D3DVSD_REG(D3DVSDE_TEXCOORD0,
             D3DVSDT_FLOAT2), // Base texture
  D3DVSD_STREAM(1),    // Tangent basis stream
  D3DVSD_REG(D3DVSDE_TEXCOORD1, D3DVSDT_FLOAT3),// Tangent
  D3DVSD_END()
};
```

どちらの宣言でも、Direct3D ランタイムに対して、(1) ストリーム **0** の頂点フォーマットの最初の 3 個の浮動小数点数が頂点の位置座標であるということ、(2) 法線は、ストリーム **0** の最初の 3 個の浮動小数点数に続く次の 3 個の浮動小数点数であるということ、(3) テクスチャ座標は、ストリーム **0** の法線データの終わりから **DWORD** 2 個分の大きさだけ後ろにあることを表しています。接ベクトルは、ストリーム**1**の頂点フォーマットの最初の3個の浮動小数点として与えられます。

Direct3D 9 Cg ランタイム用の Cg 頂点プログラムから頂点宣言を取得するには、**cgD3D9GetVertexDeclaration()** を使用します。

```
CGbool cgD3D9GetVertexDeclaration(CGprogram program,
        D3DVERTEXELEMENT9 declaration[MAXD3DDECLLENGTH]);
```

MAXD3DDECLLENGTH は、Direct3D 9 頂点宣言の最大長を与える Direct3D 9 定数です。プログラムから宣言を導出できない場合、**cgD3D9GetVertexDeclaration()** は失敗し、**CG_FALSE** を返します。

Direct3D 8 Cg ランタイム用の Cg 頂点プログラムから頂点宣言を取得するには、**cgD3D8GetVertexDeclaration()** を使用します。

```
CGbool cgD3D8GetVertexDeclaration(CGprogram program,
        DWORD declaration[MAX_FVF_DECL_SIZE]);
```

MAX_FVF_DECL_SIZE は、Direct3D 頂点宣言の最大長を与える Direct3D 定数です。プログラムから宣言を導出できない場合、**cgD3D8GetVertexDeclaration()** は失敗し、**CG_FALSE** を返します。

cgD3D9GetVertexDeclaration() または **cgD3D8GetVertexDeclaration()** で返される宣言は単一ストリーム用であるため、次のプログラムは、

```
void main(in  float4 position : POSITION,
          in  float4 color    : COLOR0,
          in  float4 texCoord : TEXCOORD0,
          out float4 hpos     : POSITION)
{ }
```

次のプログラムと等価です。

```
const D3DVERTEXELEMENT9 declaration[] = {
  { 0, 0 * sizeof(float),
    D3DDECLTYPE_FLOAT4, D3DDECLMETHOD_DEFAULT,
    D3DDECLUSAGE_POSITION, 0 },
  { 0, 4 * sizeof(float),
    D3DDECLTYPE_FLOAT4, D3DDECLMETHOD_DEFAULT,
    D3DDECLUSAGE_COLOR, 0 },
  { 0, 8 * sizeof(float),
    D3DDECLTYPE_FLOAT4, D3DDECLMETHOD_DEFAULT,
    D3DDECLUSAGE_TEXCOORD, 0 },
  D3DD3CL_END()
};
```

これは Direct3D 9 Cg ランタイム用です。また、次のプログラムとも等価です。

```
const DWORD declaration[] = {
  D3DVSD_STREAM(0),
  D3DVSD_REG(D3DVSDE_POSITION, D3DVSDT_FLOAT4),
  D3DVSD_REG(D3DVSDE_DIFFUSE, D3DVSDT_FLOAT4),
  D3DVSD_REG(D3DVSDE_TEXCOORD0, D3DVSDT_FLOAT4),
  D3DVSD_END()
};
```

これは Direct3D 8 Cg ランタイム用です。

通常、頂点プログラムは、複数のストリームからなる、または特定の頂点フォーマットを持つ幾何学データに適用します。この場合、頂点宣言は、頂点プログラムではなく頂点フォーマットに基づき、頂点宣言を用意します。プログラムと互換性があるかどうかを確認するには、**cgD3D9ValidateVertexDeclaration()** を使用します。

```
CGbool cgD3D9ValidateVertexDeclaration(CGprogram program,
        const D3DVERTEXELEMENT9* declaration);
```

これは Direct3D 9 Cg ランタイム用です。または、**cgD3D8ValidateVertexDeclaration()** を使用します。

```
CGbool cgD3D8ValidateVertexDeclaration(CGprogram program,
        const DWORD* declaration);
```

これは Direct3D 8 Cg ランタイム用です。

頂点宣言がプログラムと互換性がある場合は、**cgD3D9ValidateVertexDeclaration()** または **cgD3D8ValidateVertexDeclaration()** のコールで **CG_TRUE** が返されます。頂点プログラムで使用されるすべての varying 型の入力パラメータに対し、適合するエントリを頂点宣言が持つ場合、Direct3D 9 の頂点宣言は頂点プログラムと互換性があります。

頂点プログラムで使用されるすべての varying 型の入力パラメータに対し、適合する **D3DVSD_REG()** マクロ呼出しを頂点宣言が持つ場合、Direct3D 8 の頂点宣言は頂点プログラムへと互換性があります。

```
void main(float4 position : POSITION,
          float4 color : COLOR0,
          float4 texCoord : TEXCOORD0)
{ }
```

次の Direct3D 9 頂点宣言は有効です。

```
const D3DVERTEXELEMENT9 declaration[] = {
  { 0, 0 * sizeof(float),
    D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
    D3DDECLUSAGE_POSITION, 0 },
  { 0, 3 * sizeof(float),
    D3DDECLTYPE_D3DCOLOR, D3DDECLMETHOD_DEFAULT,
    D3DDECLUSAGE_COLOR, 0 },
  { 1, 4 * sizeof(float),
    D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT,
    D3DDECLUSAGE_TEXCOORD, 0 },
  D3DD3CL_END()
};
```

次の Direct3D 8 頂点宣言は有効です。

```
DWORD declaration[] = {
  D3DVSD_STREAM(0),
  D3DVSD_REG(D3DVSDE_POSITION, D3DVSDT_FLOAT3),
  D3DVSD_REG(D3DVSDE_DIFFUSE, D3DVSDT_D3DCOLOR),
  D3DVSD_STREAM(1),
  D3DVSD_SKIP(4),
  D3DVSD_REG(D3DVSDE_TEXCOORD0, D3DVSDT_FLOAT2),
  D3DVSD_END()
};
```

これらが有効なのは、**D3DDECLUSAGE_POSITION** および **D3DVSDE_POSITION** が、定義済みセマンティクス **POSITION** に関連付けられているハードウェア レジスタと一致し、**D3DDECLUSAGE_DIFFUSE** および **D3DVSDE_DIFFUSE** が、**COLOR0** に関連付けられているレジスタと一致し、**D3DDECLUSAGE_TEXCOORD0** および **D3DVSDE_TEXCOORD0** が、**TEXCOORD0** に関連付けられているレジスタと一致するためです。

前述の宣言は、**cgD3D9ResourceToDeclUsage()** または **cgD3D8ResourceToInputRegister()** を使用して次の方法で作成することもできます。

```
const D3DVERTEXELEMENT9 declaration[] = {
  { 0, 0 * sizeof(float),
    D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
    cgD3D9ResourceToDeclUsage(CG_POSITION), 0 },
  { 0, 3 * sizeof(float),
    D3DDECLTYPE_D3DCOLOR, D3DDECLMETHOD_DEFAULT,
    cgD3D9ResourceToDeclUsage(CG_COLOR0), 0 },
  { 1, 4 * sizeof(float),
    D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT,
    cgD3D9ResourceToDeclUsage(CG_TEXCOORD0), 0 },
  D3DD3CL_END()
};
```

```
DWORD declaration[] = {
  D3DVSD_STREAM(0),
  D3DVSD_REG(cgD3D8ResourceToInputRegister(CG_POSITION),
                                           D3DVSDT_FLOAT3),
  D3DVSD_REG(cgD3D8ResourceToInputRegister(CG_COLOR0),
                                           D3DVSDT_D3DCOLOR),
  D3DVSD_STREAM(1),
  D3DVSD_SKIP(4),
  D3DVSD_REG(cgD3D8ResourceToInputRegister(CG_TEXCOORD0),
                                           D3DVSDT_FLOAT2),
  D3DVSD_END()
};
```

可能な場合、**cgD3D9ResourceToDeclUsage()** および **cgD3D8ResourceToInputRegister()** は、**CGresource** 列挙型を Direct3D 頂点シェーダ入力レジスタに変換します。

```
BYTE  cgD3D9ResourceToDeclUsage(CGresource resource);
DWORD cgD3D8ResourceToInputRegister(CGresource resource);
```

リソースが頂点シェーダの入力リソースでない場合、**cgD3D9ResourceToDeclUsage()** のコールは **CGD3D9_INVALID_REG** を返し、**cgD3D8ResourceToInputRegister()** のコールは **CGD3D8_INVALID_REG** を返します。

プログラム パラメータに基づいて前述の頂点宣言を記述し、セマンティクスの参照を排除するには、**cgD3D9ResourceToDeclUsage()** または **cgD3D8ResourceToInputRegister()** を使用します。

```
CGparameter position =
               cgGetNamedParameter(program, "position");
CGparameter color =
               cgGetNamedParameter(program, "color");
CGparameter texCoord =
               cgGetNamedParameter(program, "texCoord");
```

```
const D3DVERTEXELEMENT9 declaration[] = {
  { 0, 0 * sizeof(float),
    D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
    cgD3D9ResourceToDeclUsage(
      cgGetParameterResource(position)),
    cgGetParameterResourceIndex(position) },
  { 0, 3 * sizeof(float),
    D3DDECLTYPE_D3DCOLOR, D3DDECLMETHOD_DEFAULT,
    cgD3D9ResourceToDeclUsage(cgGetParameterResource(color)),
    cgGetParameterResourceIndex(color) },
  { 1, 4 * sizeof(float),
    D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT,
    cgD3D9ResourceToDeclUsage(
      cgGetParameterResource(texCoord)),
    cgGetParameterResourceIndex(texCoord) },
  D3DD3CL_END()
};
```

```
DWORD declaration[] = {
  D3DVSD_STREAM(0),
  D3DVSD_REG(cgD3D8ResourceToInputRegister(
           cgGetParameterResource(position)), D3DVSDT_FLOAT3),
  D3DVSD_REG(cgD3D8ResourceToInputRegister(
           cgGetParameterResource(color)), D3DVSDT_D3DCOLOR),
  D3DVSD_STREAM(1),
  D3DVSD_SKIP(4),
  D3DVSD_REG(cgD3D8ResourceToInputRegister(
           cgGetParameterResource(texCoord)), D3DVSDT_FLOAT2),
  D3DVSD_END()
};
```

Direct3D 8 宣言の **D3DVSD_REG()** マクロ呼出しの 2 番目の引数として指定するサイズが、対応するパラメータのサイズと一致していなくとも、頂点宣言は有効です。これらのサイズは、シェーダ コードの型チェックを実行するためではなく、ストリーム内のデータの配置方法を記述するために指定されます。**D3DVSD_REG()** マクロ呼出しにより参照されるデータは、対応するハードウェア レジスタの 4 つの浮動小数点値に展開され、欠落値は 0（**x**、**y** および **z** の場合）および 1（**w** の場合）に設定されます。

最小インタフェースの型の取得

浮動小数点数換算での **CGtype** で列挙された型のサイズを取得するには **cgD3D9TypeToSize()** を使用します。

```
DWORD cgD3D9TypeToSize(CGtype type);
```

より正確には、これは ***type*** 型のパラメータの格納に必要な浮動小数点値の個数です。この関数は、**sampler** 型などの一部の型には適用されず、その場合は 0 を返します。特定のパラメータの値を設定する際に指定が必要な浮動小数点値の個数をアプリケーションが判断できるため、これは便利な関数です。

最小インタフェースのプログラム例

ここで示すコード サンプルでは、最小インタフェースの関数を使用して、Cg プログラムを Direct3D で動作させるための方法を示します。コードのわかりやすさを強調するために、サンプルではエラー チェックをほとんど行っていませんが、実働アプリケーションではすべての Cg 関数の戻り値をチェックする必要があります。次に示す頂点プログラムとフラグメント プログラムは、64 ページの「Direct3D 9 アプリケーション」および 67 ページの「Direct3D 8 アプリケーション」で参照されています。

頂点プログラム

次のCgコードは、**VertexProgram.cg**というファイル内にあると想定しています。

```
void VertexProgram(
  in  float4 position  : POSITION,
  in  float4 color     : COLOR0,
  in  float4 texCoord  : TEXCOORD0,
  out float4 positionO : POSITION,
  out float4 colorO    : COLOR0,
  out float4 texCoordO : TEXCOORD0,
  const uniform float4x4 ModelViewMatrix)
{
  positionO = mul(position, ModelViewMatrix);
  colorO = color;
  texCoordO = texCoord;
}
```

フラグメント プログラム

次のCgコードは、**FragmentProgram.cg**というファイル内にあると想定します。

```
void FragmentProgram(
  in  float4 color    : COLOR0,
  in  float4 texCoord : TEXCOORD0,
  out float4 colorO   : COLOR0,
  const uniform sampler2D BaseTexture,
  const uniform float4 SomeColor)
{
  colorO = color * tex2D(BaseTexture, texCoord) + SomeColor;
}
```

Direct3D 9 アプリケーション

次の C コードは、前述の頂点プログラムとフラグメント プログラムを Direct3D 9 アプリケーションにリンクします。

```
#include <cg/cg.h>
#include <cg/cgD3D9.h>

IDirect3DDevice9*  device;  // Initialized somewhere else
IDirect3DTexture9* texture; // Initialized somewhere else
D3DXMATRIX matrix;          // Initialized somewhere else
D3DXCOLOR constantColor;    // Initialized somewhere else
CGcontext context;
CGprogram vertexProgram, fragmentProgram;
IDirect3DVertexDeclaration9* vertexDeclaration;
IDirect3DVertexShader9* vertexShader;
IDirect3DPixelShader9* pixelShader;
CGparameter baseTexture, someColor, modelViewMatrix;

// Called at application startup
void OnStartup()
{
  // Create context
  context = cgCreateContext();
}

// Called whenever the Direct3D device needs to be created
void OnCreateDevice()
{
  // Create the vertex shader
  vertexProgram = cgCreateProgramFromFile(context, CG_SOURCE,
   "VertexProgram.cg", CG_PROFILE_VS_2_0, "VertexProgram", 0);
  CComPtr<ID3DXBuffer> byteCode;
  const char* progSrc = cgGetProgramString(vertexProgram,
```

```
                          CG_COMPILED_PROGRAM);
  D3DXAssembleShader(progSrc, strlen(progSrc), 0, 0, 0,
                     &byteCode, 0);
  // If your program uses explicit binding semantics (like
  // this one), you can create a vertex declaration
  // using those semantics.
  const D3DVERTEXELEMENT9 declaration[] = {
    { 0, 0 * sizeof(float),
      D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
      D3DDECLUSAGE_POSITION, 0 },
    { 0, 3 * sizeof(float),
      D3DDECLTYPE_D3DCOLOR, D3DDECLMETHOD_DEFAULT,
      D3DDECLUSAGE_COLOR, 0 },
    { 0, 4 * sizeof(float),
      D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT,
      D3DDECLUSAGE_TEXCOORD, 0 },
    D3DD3CL_END()
  };
  // Make sure the resulting declaration is compatible with
  // the shader. This is really just a sanity check.
  assert(cgD3D9ValidateVertexDeclaration(vertexProgram,
                                         declaration));
  device->CreateVertexDeclaration(
            declaration, &vertexDeclaration);
  device->CreateVertexShader(
            byteCode->GetBufferPointer(), &vertexShader);

  // Create the pixel shader.
  fragmentProgram = cgCreateProgramFromFile(context,
                     CG_SOURCE, "FragmentProgram.cg",
                     CG_PROFILE_PS_2_0, "FragmentProgram", 0);
{
  CComPtr<ID3DXBuffer> byteCode;
  const char* progSrc = cgGetProgramString(fragmentProgram,
                          CG_COMPILED_PROGRAM);
  D3DXAssembleShader(progSrc, strlen(progSrc), 0, 0, 0,
                     &byteCode, 0);
  device->CreatePixelShader(byteCode->GetBufferPointer(),
                            &pixelShader)
}

  // Grab some parameters.
  modelViewMatrix = cgGetNamedParameter(vertexProgram,
                                        "ModelViewMatrix");
```

```
  baseTexture = cgGetNamedParameter(fragmentProgram,
                                    "BaseTexture");
  someColor = cgGetNamedParameter(fragmentProgram,
                                  "SomeColor");

  // Sanity check that parameters have the expected size
  assert(cgD3D9TypeToSize(cgGetParameterType(
                                     modelViewMatrix)) == 16);
  assert(cgD3D9TypeToSize(cgGetParameterType(someColor))
         == 4);
}

// Called to render the scene
void OnRender()
{
  // Get the Direct3D resource locations for parameters
  // This can be done earlier and saved
  DWORD modelViewMatrixRegister =
                cgGetParameterResourceIndex(modelViewMatrix);
  DWORD baseTextureUnit =
          cgGetParameterResourceIndex(baseTexture);
  DWORD someColorRegister =
          cgGetParameterResourceIndex(someColor);

  // Set the Direct3D state.
  device->SetVertexShaderConstantF(modelViewMatrixRegister,
                                   &matrix, 4);
  device->SetPixelShaderConstantF(someColorRegister,
                                  &constantColor, 1);
  device->SetVertexDeclaration(vertexDeclaration);
  device->SetTexture(baseTextureUnit, texture);
  device->SetVertexShader(vertexShader);
  device->SetPixelShader(pixelShader);

  // Draw scene.
  // ...
}

// Called before the device changes or is destroyed
void OnDestroyDevice() {
  vertexShader->Release();
  pixelShader->Release();
  vertexDeclaration->Release();
}
```

```
// Called before application shuts down
void OnShutdown() {
  // This frees any core runtime resources.
  // The minimal interface has no dynamic storage to free.
  cgDestroyContext(context);
}
```

Direct3D 8 アプリケーション

次の C コードは、前述の頂点プログラムとフラグメント プログラムを Direct3D 8 アプリケーションにリンクします。

```
#include <cg/cg.h>
#include <cg/cgD3D8.h>

IDirect3DDevice8*  device;  // Initialized somewhere else
IDirect3DTexture8* texture; // Initialized somewhere else
D3DXMATRIX matrix;          // Initialized somewhere else
D3DXCOLOR constantColor;    // Initialized somewhere else
CGcontext context;
CGprogram vertexProgram, fragmentProgram;
DWORD vertexShader, pixelShader;
CGparameter baseTexture, someColor, modelViewMatrix;

// Called at application startup
void OnStartup()
{
  // Create context
  context = cgCreateContext();
}

// Called whenever the Direct3D device needs to be created
void OnCreateDevice()
{
  // Create the vertex shader
  vertexProgram = cgCreateProgramFromFile(context, CG_SOURCE,
   "VertexProgram.cg", CG_PROFILE_VS_1_1, "VertexProgram", 0);
  CComPtr<ID3DXBuffer> byteCode;
  const char* progSrc = cgGetProgramString(vertexProgram,
                        CG_COMPILED_PROGRAM);
  // Normally, you also grab the constants and prepend them
  // to your vertex declaration. Not shown here for brevity.
  D3DXAssembleShader(progSrc, strlen(progSrc), 0, 0, 0,
                     &byteCode, 0);
  // If your program uses explicit binding semantics (like
  // this one), you can create a vertex declaration
  // using those semantics.
```

```
  DWORD declaration[] = {
    D3DVSD_STREAM(0),
    D3DVSD_REG(D3DVSDE_POSITION, D3DVSDT_FLOAT3),
    D3DVSD_REG(D3DVSDE_DIFFUSE, D3DVSDT_D3DCOLOR),
    D3DVSD_REG(D3DVSDE_TEXCOORD0, D3DVSDT_FLOAT2),
    D3DVSD_END()
  }
  // Make sure the resulting declaration is compatible with
  // the shader. This is really just a sanity check.
  assert(cgD3D8ValidateVertexDeclaration(vertexProgram,
                                         declaration));
  // Create the shader handle using the declaration.
  device->CreateVertexShader(declaration,
              byteCode->GetBufferPointer(), &vertexShader, 0);

  // Create the pixel shader.
  fragmentProgram = cgCreateProgramFromFile(context,
                    CG_SOURCE, "FragmentProgram.cg",
                    CG_PROFILE_PS_1_1, "FragmentProgram", 0);
{
  CComPtr<ID3DXBuffer> byteCode;
  const char* progSrc = cgGetProgramString(fragmentProgram,
  CG_COMPILED_PROGRAM);
  D3DXAssembleShader(progSrc, strlen(progSrc), 0, 0, 0,
                    &byteCode, 0);
  device->CreatePixelShader(byteCode->GetBufferPointer(),
                            &pixelShader);
}

  // Grab some parameters.
  modelViewMatrix = cgGetNamedParameter(vertexProgram,
                                        "ModelViewMatrix");
  baseTexture = cgGetNamedParameter(fragmentProgram,
                                    "BaseTexture");
  someColor = cgGetNamedParameter(fragmentProgram,
                                  "SomeColor");

  // Sanity check that parameters have the expected size
  assert(cgD3D8TypeToSize(cgGetParameterType(
                                     modelViewMatrix)) == 16);
  assert(cgD3D8TypeToSize(cgGetParameterType(someColor))
         == 4);
}
```

```
// Called to render the scene
void OnRender()
{
  // Get the Direct3D resource locations for parameters
  // This can be done earlier and saved
  DWORD modelViewMatrixRegister =
                cgGetParameterResourceIndex(modelViewMatrix);
  DWORD baseTextureUnit =
          cgGetParameterResourceIndex(baseTexture);
  DWORD someColorRegister =
          cgGetParameterResourceIndex(someColor);

  // Set the Direct3D state.
  device->SetVertexShaderConstant(modelViewMatrixRegister,
                                  &matrix, 4);
  device->SetPixelShaderConstant(someColorRegister,
                                 &constantColor, 1);
  device->SetTexture(baseTextureUnit, texture);
  device->SetVertexShader(vertexShader);
  device->SetPixelShader(pixelShader);

  // Draw scene.
  // ...
}

// Called before the device changes or is destroyed
void OnDestroyDevice() {
  device->DeleteVertexShader(vertexShader);
  device->DeletePixelShader(pixelShader);
}

// Called before application shuts down
void OnShutdown() {
  // This frees any core runtime resources.
  // The minimal interface has no dynamic storage to free.
  cgDestroyContext(context);
}
```

Direct3D 拡張インタフェース

不適切な不整合を回避するためにプログラムに拡張インタフェースを使用する場合は、シェーダ設定、シェーダのアクティブ化、パラメータ設定（テクスチャ ステージ ステートの設定を含む）などの、インタフェースの関数を介して行うことのできるシェーダ関連のあらゆる操作に一貫して拡張インタフェースを使用することをお薦めします。

Direct3D デバイスの設定

拡張インタフェースは、プログラムとパラメータの管理を簡単にするために、最小インタフェースよりも多くの機能をカプセル化しています。これは、適切なタイミングで適切なDirect3Dコールを実行することにより行われます。これらのコールのいくつかは Direct3D デバイスを必要とするため、Cg ランタイムは Direct3D デバイスと通信する必要があります。

```
HRESULT cgD3D9SetDevice(IDirect3DDevice9* device);
```

ランタイムに現在関連付けられている Direct3D デバイスは、**cgD3D9GetDevice()** を使用して取得できます。

```
IDirect3DDevice9* cgD3D9GetDevice();
```

cgD3D9SetDevice() が 0 を入力としてコールされると、拡張インタフェースで使用されているすべての Direct3D リソースが解放されます。Direct3D デバイスは、そのデバイスに対する参照がすべて削除された場合にのみ破棄されるので、アプリケーションでは、Direct3D デバイスを使い終わったときに 0 を入力として **cgD3D9SetDevice()** をコールし、アプリケーションの停止時にそのデバイスが破棄されるようにする必要があります。そうしないと、Direct3D は正しく停止せず、デバッグ コンソールにメモリ リークをレポートします。

0 を入力として **cgD3D9SetDevice()** をコールしても、Cg コア ランタイムのリソースには影響しないことに注意してください。すべての関連コア ランタイム ハンドル（**CGprogram** 型、**CGparameter** 型などのハンドル）は有効なままです。

2 回目に別のデバイスを指定して **cgD3D9SetDevice()** をコールした場合、それ以前のデバイスにより管理されていたすべてのプログラムは、新しいデバイスを使用して再ビルドされます。

消失した Direct3D デバイスへの応答

拡張インタフェースは、デバイスの消失に対応して再作成が必要な Direct3D リソースへの参照を保持することがあります。具体的には、Direct3D デバイスを消失ステートからリセットする前に、特定の sampler パラメータを解放することが必要な場合があります。拡張インタフェースは、テクスチャが次の両方に該当する場合に、デバイスの消失に対応してリセットが必要なテクスチャへの参照を保持しています。

- **D3DPOOL_DEFAULT** プールに作成されている
- パラメータ シャドーイングが有効になっているプログラムの sampler パラメータに（**cgD3D9SetTexture()** を使用して）バインドされている

この場合、パラメータを 0 に設定（**cgD3D9SetTexture()** を使用）して、拡張インタフェースからそのテクスチャへの参照を削除し、テクスチャを破棄して Direct3D デバイスを消失ステートからリセットできるようにする必要があります。その後、Direct3D デバイスをリセットし、テクスチャを再作成してから、sampler パラメータに再バインドする必要があります。次に例を示します。

```
IDirect3DDevice9* device; // Initialized elsewhere
IDirect3DTexture9* myDefaultPoolTexture;
CGprogram program;

void OneTimeLoadScene()
{
  // Load the program with cgD3D9LoadProgram and
  // enable parameter shadowing
  /* ... */
  cgD3D9LoadProgram(program, TRUE, 0, 0, 0);
  /* ... */
  // Bind sampler parameter
  GCparameter parameter;
  parameter = cgGetParameterByName(program, "MySampler");
  cgD3D9SetTexture(parameter, myDefaultPoolTexture);
}

void OnLostDevice()
{
  // First release all necessary resources
  PrepareForReset();
  // Next actually reset the D3D device
  device->Reset( /* ... */ );
  // Finally recreate all those resource
  OnReset();
}

void PrepareForReset()
{
  /* ... */
  // Release expanded interface reference
  cgD3D9SetTexture(mySampler, 0);
  // Release local reference
  // and any other references to the texture
  myDefaultPoolTexture->Release();
  /* ... */
}

void OnReset()
{
  // Recreate myDefaultPoolTexture in D3DPOOL_DEFAULT
  /* ... */
  // Since the texture was just recreated,
  // it must be re-bound to the parameter
  GCparameter parameter;
```

```
  parameter = cgGetParameterByName(prog, "MySampler");
  cgD3D9SetTexture(mySampler, myDefaultPoolTexture);
  /* ... */
}
```

消失デバイスの詳細な説明と、これらを適切に処理する方法については、Direct3D のドキュメントを参照してください。

拡張インタフェース パラメータの設定

ここでは、uniform 型スカラ、uniform 型ベクトル、uniform 型行列、この 3 つの型の uniform 型配列、sampler など、拡張インタフェースの様々な型のパラメータの設定について説明します。

uniform 型スカラ、ベクトルおよび行列パラメータの設定

関数 **cgD3D9SetUniform()** は、**float3** や **float4x3** などの浮動小数点パラメータを設定します。

```
HRESULT cgD3D9SetUniform(CGparameter parameter,
                         const void* value);
```

必要なデータ量は ***parameter*** の型によって決まりますが、常に 1 つ以上の浮動小数点値の配列として指定されます。型は **void*** であるため、互換性のあるユーザ定義構造体は型をキャストせずに渡すことができます。次のコードは、**float3** 型の ***vectorParam***、**float2x3** 型の ***matrixParam*** および **float2x2[3]** 型の ***arrayParam***を設定するための**cgD3D9SetUniform()**の使用法を示したものです。

```
D3DXVECTOR3 vectorData(1,2,3);
float matrixData[2][3] = {{1, 2, 3}, {4, 5, 6}};
float arrayData[3][2][2] =
      {{{1, 2}, {3, 4}},{{5, 6},{7,8}}, {{9, 10}, {11, 12}}};
cgD3D9SetUniform(vectorParam, &vectorData);
cgD3D9SetUniform(matrixParam, matrixData);
cgD3D9SetUniform(arrayParam, arrayData);
```

前述のように、**cgD3D9TypeToSize()** を使用し、特定の型のパラメータを設定するために必要な値の個数を判断できます。

便宜を図るため、**D3DMATRIX** 型の 4 × 4 の行列からパラメータを設定する関数もあります。

```
HRESULT cgD3D9SetUniformMatrix(CGparameter parameter,
                               const D3DMATRIX* matrix);
```

入力パラメータのサイズに合せて行列の左上部分が抽出されるため、***matrixParam*** もこの方法で設定できます。

```
D3DXMATRIX matrix(
  1, 1, 1, 0,
  1, 1, 1, 0,
  0, 0, 0, 0,
  0, 0, 0, 0,
);
cgD3D9SetUniformMatrix(matrixParam, &matrix);
```

前述の例では、***matrixParam*** のすべての要素が 1 に設定されます。

uniform 型スカラ、ベクトルおよび行列パラメータの配列の設定

配列パラメータを設定するには、**cgD3D9SetUniformArray()** を使用します。

```
HRESULT cgD3D9SetUniformArray(CGparameter parameter,
          DWORD startIndex, DWORD numberOfElements,
          const void* array);
```

パラメータ ***startIndex*** および ***numberOfElements*** は、設定される配列パラメータの要素を指定します。***startIndex*** から ***startIndex+numberOfElements-1*** までの、***numberOfElements*** 個の要素を指定したことになります。***array*** は、これらすべての要素を設定するのに十分な値が必要です。**cgD3D9SetUniform()** と同様、**cgD3D9TypeToSize()** を使用して必要な値の個数を判断でき、型は **void*** であるため、互換性のあるユーザ定義構造体は型をキャストせずに渡すことができます。

cgD3D9SetUniformMatrix() と同等の便利な関数もあります。

```
HRESULT cgD3D9SetUniformMatrixArray(CGparameter parameter,
          DWORD startIndex, DWORD numberOfElements,
          const D3DMATRIX* matrices);
```

パラメータ ***startIndex*** および ***numberOfElements*** は、**cgD3D9SetUniformMatrix()** と同じ意味を持ちます。

配列パラメータ ***parameter*** の要素のサイズに合せて、配列 ***matrices*** の各行列の左上部分が抽出されます。配列 ***matrices*** には、***numberOfElements*** 個の要素が必要です。

sampler パラメータの設定

sampler パラメータには、次の関数を使用して Direct3D テクスチャを割り当てます。

```
HRESULT cgD3D9SetTexture(CGparameter parameter,
          IDirect3DBaseTexture9* texture);
```

Direct3D 9 Cg ランタイムで sampler ステートを設定するには、次の関数を使用します。

```
HRESULT cgD3D9SetSamplerState(CGparameter parameter,
          D3DSAMPLERSTATETYPE type, DWORD value);
```

パラメータ ***type*** は **D3DSAMPLERSTATETYPE** 列挙のいずれかであり、パラメータ ***value***の適切な値は、***type***によって異なります。この関数の使用例を次に示します。

```
cgD3D9SetSamplerState(parameter, D3DSAMP_MAGFILTER,
                      D3DTEXF_LINEAR);
```

Direct3D 8 Cg ランタイムでテクスチャ ステージ ステートを設定するには、次の関数を使用します。

```
HRESULT cgD3D8SetTextureStageState(CGparameter parameter,
          D3DTEXTURESTAGESTATETYPE type, DWORD value);
```

パラメータ ***type*** は、次の値のいずれかです。

D3DTSS_ADDRESSU	**D3DTSS_ADDRESSV**
D3DTSS_ADDRESSW	**D3DTSS_BORDERCOLOR**
D3DTSS_MAGFILTER	**D3DTSS_MINFILTER**
D3DTSS_MIPFILTER	**D3DTSS_MIPMAPLODBIAS**
D3DTSS_MAXMIPLEVEL	**D3DTSS_MAXANISOTROPY**

パラメータ ***value*** の適切な値は、***type*** によって異なります。この関数の使用例を次に示します。

```
cgD3D8SetTextureStageState(parameter, D3DTSS_MAGFILTER,
                           D3DTEXF_LINEAR);
```

テクスチャ ラップ モードは、次の関数を使用して設定します。

```
HRESULT cgD3D9SetTextureWrapMode(CGparameter parameter,
          DWORD value);
```

入力 ***value*** は 0 か、あるいは **D3DWRAP_U**、**D3DWRAP_V** および **D3DWRAP_W** の組合せです。この関数の使用例を次に示します。

```
cgD3D9SetTextureWrapMode(parameter, D3DWRAP_U | D3DWRAP_V);
```

パラメータ シャドーイング

パラメータ シャドーイングは、プログラムごとに次の時点で有効 / 無効を設定できます。

- プログラムをロードする時点で（75 ページの「拡張インタフェース プログラムの実行」を参照）
- 次の関数を使用して任意の時点で

  ```
  HRESULT cgD3D9EnableParameterShadowing(
            CGprogram program, CGbool enable);
  ```

  この場合、パラメータ シャドーイングを有効にするには ***enable*** を **CG_TRUE** に設定し、無効にするには **CG_FALSE** に設定します。

パラメータ シャドーイングが特定のプログラムに対して有効になっているかどうかを調べるには、次の関数を使用します。

```
CGbool cgD3D9IsParameterShadowingEnabled(CGprogam program);
```

この関数は、***program*** に対してパラメータ シャドーイングが有効になっている場合に **CG_TRUE** を返します。

拡張インタフェース プログラムの実行

Direct3D 9でプログラムをロードするには、**cgD3D9LoadProgram()** を使用します。

```
HRESULT cgD3D9LoadProgram(CGprogram program,
          CG_BOOL parameterShadowingEnabled,
          DWORD assembleFlags);
```

この関数は、***assembleFlags*** に **D3DXASM** フラグを設定し、**D3DXAssembleShader()** を使用して ***program*** のコンパイル結果をアセンブルします。プログラムのプロファイルに応じて、**IDirect3DDevice9::CreateVertexShader()** を使用して Direct3D 9 頂点シェーダを作成するか、あるいは **IDirect3DDevice9::CreatePixelShader()** を使用して Direct3D 9 ピクセル シェーダを作成します。

この関数の典型的な使用法を次に示します。

```
HRESULT hresult = cgD3D9LoadProgram(vertexProgram, TRUE,
                                    D3DXASM_DEBUG);
HRESULT hresult = cgD3D9LoadProgram(fragmentProgram, TRUE, 0);
```

Direct3D 8でプログラムをロードするには、**cgD3D8LoadProgram()** を使用します。

```
HRESULT cgD3D8LoadProgram(CGprogram program,
          BOOL parameterShadowingEnabled, DWORD assembleFlags,
          DWORD vertexShaderUsage, const DWORD* declaration);
```

この関数は、***assembleFlags*** に **D3DXASM** フラグを設定し、**D3DXAssembleShader()** を使用して ***program*** のコンパイル結果をアセンブルします。次に、プログラムのプロファイルに応じ、**IDirect3DDevice8::CreateVertexShader()** を使用して Direct3D 頂点シェーダを作成するか、または **IDirect3DDevice8::CreatePixelShader()** を使用して Direct3D ピクセル シェーダを作成します。前者のとき、***declaration*** は頂点宣言、***vertexShaderUsage*** は使用情報の制御情報です。

プログラムのパラメータ シャドーイングを有効にするには、***parameterShadowingEnabled*** の値を **TRUE** に設定する必要があります。この動作は、**cgD3DEnableParameterShadowing()** をコールしてプログラムが作成された後で変更できます。この関数の典型的な使用法を次に示します。

```
HRESULT hresult = cgD3D8LoadProgram(vertexProgram, TRUE,
          D3DXASM_DEBUG, D3DUSAGE_SOFTWAREVERTEXPROCESSING,
          declaration);
HRESULT hresult = cgD3D8LoadProgram(fragmentProgram, TRUE,
                                    0, 0, 0);
```

それぞれ異なるレイアウトを持つ複数のジオメトリ データ セットに対して同じ頂点プログラムを適用する場合は、Direct3D 8 では、異なる頂点宣言を持つプログラムをロードする必要があります。これを行うには、これらの各頂点宣言について、**cgCopyProgram()** を使用して頂点プログラムを複製します。次のコード例は、この操作を示したものです。

```
CGprogam program1, program2;
program1 = cgCreateProgramFromFile(context, CG_SOURCE,
             "VertexProgram.cg", CG_PROFILE_VS_1_1, 0, 0);
const DWORD declaration1 =
                          cgD3D8GetVertexDeclaration(program1);
cgD3D8LoadProgram(program1, TRUE, 0, 0, declaration1);
program2 = cgCopyProgram(program1);
const DWORD declaration2[] = {
  //... Custom declaration ...
};
if (cgD3D8ValidateVertexDeclaration(program2, declaration2))
  cgD3D8LoadProgram(program2, TRUE, 0, 0, declaration2);
```

Direct3D 9 と Direct3D 8 ではロード関数のみ異なります。アンロード関数とバインド関数は同じです。

Direct3D シェーダ オブジェクトやシャドーイングされたパラメータなど、**cgD3D9LoadProgram()** によって割り当てられた Direct3D リソースを解放するには、次の関数を使用します。

```
HRESULT cgD3D9UnloadProgam(CGprogram program);
```

cgD3D9UnloadProgam() では、***program*** やそのパラメータ ハンドルなどのコア ランタイム リソースは解放されないことに注意してください。一方、**cgDestroyProgram()** または **cgDestroyContext()** を使用してプログラムを破棄した場合は、**cgD3D9UnloadProgam()** の間接的なコールによりすべての Direct3D リソースが解放されます。

関数 **cgD3D9IsProgramLoaded()** は、***program*** がロードされている場合に **CG_TRUE** を返します。

```
CGbool cgD3D9IsProgramLoaded(CGprogram program);
```

すべてのプログラムは、バインドする前にロードする必要があります。プログラムをバインドするには、**cgD3D9BindProgram()** をコールします。

```
HRESULT cgD3D9BindProgram(CGprogram program);
```

この関数は、基本的に、プログラムのプロファイルに応じて **IDirect3DDevice9::SetVertexShader()** または **IDirect3DDevice9::SetPixelShader()** をコールすることにより、***program*** に対応する Direct3D シェーダをアクティブにします。パラメータ シャドーイングが***program***に対して有効になっている場合は、シャドーイングされたパラメータおよびそれに関連付けられている Direct3D ステート（**sampler** パラメータに対するテクスチャ ステージ ステートなど）をすべて設定します。値やステートの追跡はランタイムにより実行されないため、この設定は、これらのパラメータ

やそのステートの現在値にかかわらず行われます。シャドーイングされたパラメータが、**cgD3D9BindProgram()** のコール時までに設定されていない場合は、このパラメータに対していかなる種類の Direct3D コールも発行されません。

ある特定の時点でバインドできるのは1つの頂点プログラムと1つのフラグメント プログラムのみであるため、あるタイプのプログラムをバインドすると、同じタイプの他のプログラムは暗黙的にバインド解除されます。

拡張インタフェース プロファイル サポート

デバイスでサポートされている最新バージョンの頂点シェーダおよびピクセル シェーダを返す 2 つの便利な関数が用意されています。

```
CGprofile cgD3D9GetLatestVertexProfile();
CGprofile cgD3D9GetLatestPixelProfile();
```

これにより、これらのプロファイルがアプリケーションの作成時に存在していなかった場合でも、Cg プログラムは実行時に使用可能な最適プロファイルに対して自動的にコンパイルされるため、アプリケーションを将来の使用に備えることができます。最適なコンパイルを可能にするもう 1 つの関数として、**cgD3D9GetOptimalOptions()** もあります。この関数は、特定のプロファイルに対して、コンパイラ オプションの最適なセットを表す文字列を返します。

```
char const* cgD3D9GetOptimalOptions(CGprofile profile);
```

この文字列は、**cgCreateProgram()** への引数の一部として使用されます。アプリケーションで破棄する必要はありません。ただし、別の Direct3D デバイスの同じプロファイルに対して **cgD3D9GetOptimalOptions()** が再びコールされた場合は、その内容が変化することがあります。

拡張インタフェースのプログラム例

ここで示すコード サンプルでは、拡張インタフェースの関数を使用して、Cg プログラムを Direct3D で動作させるための方法を示します。コードのわかりやすさを強調するために、サンプルではエラー チェックをほとんど行っていませんが、実働アプリケーションではすべての Cg 関数の戻り値をチェックする必要があります。次に示す頂点プログラムとフラグメント プログラムは、78 ページの「拡張インタフェースの Direct3D 9 アプリケーション」および 81 ページの「拡張インタフェースの Direct3D 8 アプリケーション」で参照されています。

拡張インタフェース頂点プログラム

次のCgコードは、**VertexProgram.cg**というファイル内にあると想定しています。

```
void VertexProgram(
                  in  float4 position  : POSITION,
                  in  float4 color     : COLOR0,
                  in  float4 texCoord  : TEXCOORD0,
                  out float4 positionO : POSITION,
                  out float4 colorO    : COLOR0,
                  out float4 texCoordO : TEXCOORD0,
                  const uniform float4x4 ModelViewMatrix)
```

```
{
  positionO = mul(position, ModelViewMatrix);
  colorO = color;
  texCoordO = texCoord; }
```

拡張インタフェース フラグメント プログラム

次の Cg コードは、**FragmentProgram.cg** というファイル内にあることを想定します。

```
void FragmentProgram(
                     in  float4 color    : COLOR0,
                     in  float4 texCoord : TEXCOORD0,
                     out float4 colorO   : COLOR0,
                     const uniform sampler2D BaseTexture,
                     const uniform float4 SomeColor)
{
  colorO = color * tex2D(BaseTexture, texCoord) + SomeColor;
}
```

拡張インタフェースの Direct3D 9 **アプリケーション**

次の C コードは、前述の頂点プログラムとフラグメント プログラムを Direct3D 9 アプリケーションにリンクします。

```
#include <cg/cg.h>
#include <cg/cgD3D9.h>

IDirect3DDevice9* device;   // Initialized somewhere else
IDirect3DTexture9* texture; // Initialized somewhere else
D3DXCOLOR constantColor;    // Initialized somewhere else
CGcontext context;
IDirect3DVertexDeclaration9* vertexDeclaration;
CGprogram vertexProgram, fragmentProgram;
CGparameter baseTexture, someColor, modelViewMatrix;

// Called at application startup
void OnStartup()
{
  // Create context
  context = cgCreateContext();
}

// Called whenever the Direct3D device needs to be created
void OnCreateDevice()
{
  // Pass the Direct3D device to the expanded interface.
  cgD3D9SetDevice(device);
```

```
// Determine the best profiles to use
CGprofile vertexProfile = cgD3D9GetLatestVertexProfile();
CGprofile pixelProfile  = cgD3D9GetLatestPixelProfile();

// Grab the optimal options for each profile.
const char* vertexOptions[] = {
             cgD3D9GetOptimalOptions(vertexProfile), 0 };
const char* pixelOptions[]  = {
             cgD3D9GetOptimalOptions(pixelProfile), 0 };

// Create the vertex shader.
vertexProgram = cgCreateProgramFromFile(
            context, CG_SOURCE, "VertexProgram.cg",
            vertexProfile, "VertexProgram", vertexOptions);
// If your program uses explicit binding semantics, you
// can create a vertex declaration using those semantics.
const D3DVERTEXELEMENT9 declaration[] = {
  { 0, 0 * sizeof(float),
    D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
    D3DDECLUSAGE_POSITION, 0 },
  { 0, 3 * sizeof(float),
    D3DDECLTYPE_D3DCOLOR, D3DDECLMETHOD_DEFAULT,
    D3DDECLUSAGE_COLOR, 0 },
  { 0, 4 * sizeof(float),
    D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT,
    D3DDECLUSAGE_TEXCOORD, 0 },
  D3DD3CL_END()
};

// Ensure the resulting declaration is compatible with the
// shader. This is really just a sanity check.
assert(cgD3D9ValidateVertexDeclaration(vertexProgram,
                                       declaration));
device->CreateVertexDeclaration(
          declaration, &vertexDeclaration);
// Load the program with the expanded interface.
// Parameter shadowing is enabled (second parameter = TRUE).
cgD3D9LoadProgram(vertexProgram, TRUE, 0);

// Create the pixel shader.
fragmentProgram = cgCreateProgramFromFile(
            context, CG_SOURCE, "FragmentProgram.cg",
            pixelProfile, "FragmentProgram", pixelOptions);

// Load the program with the expanded interface. Parameter
// shadowing is enabled (second parameter = TRUE). Ignore
// vertex shader specifc flags, such as declaration usage.
cgD3D9LoadProgram(fragmentProgram, TRUE, 0);
```

```
  // Grab some parameters.
  modelViewMatrix = cgGetNamedParameter(vertexProgram,
                                        "ModelViewMatrix");
  baseTexture = cgGetNamedParameter(fragmentProgram,
                                    "BaseTexture");
  someColor = cgGetNamedParameter(fragmentProgram,
                                  "SomeColor");

  // Sanity check that parameters have the expected size
  assert(cgD3D9TypeToSize(cgGetParameterType(
                            modelViewMatrix)) == 16);
   assert(cgD3D9TypeToSize(cgGetParameterType(someColor))
          == 4);

  // Set parameters that don't change. They can be set
  // only once since parameter shadowing is enabled
  cgD3D9SetTexture(baseTexture, texture);
  cgD3D9SetUniform(someColor, &constantColor);
}

// Called to render the scene
void OnRender()
{
  // Load model-view matrix.
  D3DXMATRIX modelViewMatrix;
  // ...

  // Set the parameters that change every frame
  // This must be done before binding the programs
  cgD3D9SetUniformMatrix(modelViewMatrix, &modelViewMatrix);

  // Set the vertex declaration
  device->SetVertexDeclaration(vertexDeclaration);

  // Bind the programs. This downloads any parameter values
  // that have been previously set.
  cgD3D9BindProgram(vertexProgram);
  cgD3D9BindProgram(fragmentProgram);

  // Draw scene.
  // ...
}

// Called before the device changes or is destroyed
void OnDestroyDevice()
{
  // Calling this function tells the expanded interface to
  // release its internal reference to the Direct3D device
  // and free its Direct3D resources.
  cgD3D9SetDevice(0);
```

```
}

// Called before application shuts down
void OnShutdown()
{
  // This frees any core runtime resource.
  cgDestroyContext(context);
}
```

拡張インタフェースの Direct3D 8 **アプリケーション**

次の C コードは、前述の頂点プログラムとフラグメント プログラムを Direct3D 8 アプリケーションにリンクします。

```
#include <cg/cg.h>
#include <cg/cgD3D8.h>

IDirect3DDevice8* device;   // Initialized somewhere else
IDirect3DTexture8* texture; // Initialized somewhere else
D3DXCOLOR constantColor;    // Initialized somewhere else
CGcontext context;
CGprogram vertexProgram, fragmentProgram;
CGparameter baseTexture, someColor, modelViewMatrix;

// Called at application startup
void OnStartup()
{
  // Create context
  context = cgCreateContext();
}

// Called whenever the Direct3D device needs to be created
void OnCreateDevice()
{
  // Pass the Direct3D device to the expanded interface.
  cgD3D8SetDevice(device);

  // Determine the best profiles to use
  CGprofile vertexProfile = cgD3D8GetLatestVertexProfile();
  CGprofile pixelProfile  = cgD3D8GetLatestPixelProfile();

  // Grab the optimal options for each profile.
  const char* vertexOptions[] = {
                 cgD3D8GetOptimalOptions(vertexProfile), 0 };
  const char* pixelOptions[]  = {
              cgD3D8GetOptimalOptions(pixelProfile), 0 };
```

```
// Create the vertex shader.
vertexProgram = cgCreateProgramFromFile(
            context, CG_SOURCE, "VertexProgram.cg",
            vertexProfile, "VertexProgram", vertexOptions);
// If your program uses explicit binding semantics (like
// this one), you can create a vertex declaration
// using those semantics.
DWORD declaration[] = {
             D3DVSD_STREAM(0),
             D3DVSD_REG(D3DVSDE_POSITION, D3DVSDT_FLOAT3),
             D3DVSD_REG(D3DVSDE_DIFFUSE, D3DVSDT_D3DCOLOR),
             D3DVSD_REG(D3DVSDE_TEXCOORD0, D3DVSDT_FLOAT2),
             D3DVSD_END()
}

// Ensure the resulting declaration is compatible with the
// shader. This is really just a sanity check.
assert(cgD3D8ValidateVertexDeclaration(vertexProgram,
                                       declaration));

// Load the program with the expanded interface.
// Parameter shadowing is enabled (second parameter = TRUE).
cgD3D8LoadProgram(vertexProgram, TRUE, 0, 0, declaration);

// Create the pixel shader.
fragmentProgram = cgCreateProgramFromFile(
            context, CG_SOURCE, "FragmentProgram.cg",
            pixelProfile, "FragmentProgram", pixelOptions);

// Load the program with the expanded interface.
// Parameter shadowing is enabled (second parameter = TRUE).
// Ignore vertex shader specifc flags, like declaration and
// usage.
cgD3D8LoadProgram(fragmentProgram, TRUE, 0, 0, 0);

// Grab some parameters.
modelViewMatrix = cgGetNamedParameter(vertexProgram,
                                      "ModelViewMatrix");
baseTexture = cgGetNamedParameter(fragmentProgram,
                                  "BaseTexture");
someColor = cgGetNamedParameter(fragmentProgram,
                                "SomeColor");
```

```
  // Sanity check that parameters have the expected size
   assert(cgD3D8TypeToSize(cgGetParameterType(
                             modelViewMatrix)) == 16);
   assert(cgD3D8TypeToSize(cgGetParameterType(someColor))
           == 4);

  // Set parameters that don't change. They can be set
  // only once since parameter shadowing is enabled
  cgD3D8SetTexture(baseTexture, texture);
  cgD3D8SetUniform(someColor, &constantColor);
}

// Called to render the scene
void OnRender()
{
  // Load model-view matrix.
  D3DXMATRIX modelViewMatrix;
  // ...

  // Set the parameters that change every frame
  // This must be done before binding the programs
  cgD3D8SetUniformMatrix(modelViewMatrix, &modelViewMatrix);

  // Bind the programs. This downloads any parameter values
  // that have been previously set.
  cgD3D8BindProgram(vertexProgram);
  cgD3D8BindProgram(fragmentProgram);

  // Draw scene.
  // ...
}

// Called before the device changes or is destroyed
void OnDestroyDevice()
{
  // Calling this function tells the expanded interface to
  // release its internal reference to the Direct3D device
  // and free its Direct3D resources.
  cgD3D8SetDevice(0);
}
```

```
// Called before application shuts down
void OnShutdown()
{
  // This frees any core runtime resource.
  cgDestroyContext(context);
}
```

Direct3D デバッグ モード

86 ページの「Direct3D エラー レポート」で説明するエラー レポート メカニズムに加えて、Direct3D 9 または Direct3D 8 Cg ランタイムを使用したアプリケーションの開発を支援するために、Direct3D 9 または Direct3D 8 Cg ランタイム DLL のデバッグ バージョンが用意されています。このバージョンにはデバッグ シンボルはありませんが、通常のバージョンのかわりに使用された場合に、Win32 関数の **`OutputDebugString()`** を使用して多くの役立つメッセージとトレースをデバッグ出力コンソールに出力します。デバッグ DLL が出力する情報の例を次に示します。

- Direct3D ランタイムまたは Cg コア ランタイムのエラー
- 拡張インタフェースにより管理されるパラメータについてのデバッグ情報
- 潜在的なパフォーマンスに関する警告

トレースの例を次に示します。

```
cgD3D(TRACE): Creating vertex shader for program 3
cgD3D(TRACE): Discovering parameters for vertex program 3
cgD3D(TRACE): Discovered uniform parameter 'ModelViewProj'
of type float4x4
cgD3D(TRACE): Finished discovering parameters for vertex
program 3
cgD3D(TRACE): Creating pixel shader for program 24
cgD3D(TRACE): Discovering parameters for pixel program 24
cgD3D(TRACE): Discovered sampler parameter 'BaseTexture'
cgD3D(TRACE): Discovered uniform parameter 'SomeColor' of
type float4
cgD3D(TRACE): Finished discovering parameters for pixel
program 24
cgD3D(TRACE): Shadowing state for sampler parameter
BaseTexture
cgD3D(TRACE): Shadowing sampler state D3DTSS_MAGFILTER for
sampler parameter 'BaseTexture'
cgD3D(TRACE): Shadowing sampler state D3DTSS_MINFILTER for
sampler parameter 'BaseTexture'
cgD3D(TRACE): Shadowing sampler state D3DTSS_MIPFILTER for
sampler parameter 'BaseTexture'
…
```

```
cgD3D(TRACE): Shadowing 16 values for uniform parameter
'ModelViewProj' of type float4x4
cgD3D(TRACE): Activating vertex shader for program 3
cgD3D(TRACE): Setting shadowed parameters for program 3
cgD3D(TRACE): Setting registers for uniform parameter
'ModelViewProj' of type float4x4
cgD3D(TRACE): Setting constant registers [0 - 3] for
parameter 'ModelViewProj' of type float4x4
cgD3D(TRACE): Activating pixel shader for program 24
cgD3D(TRACE): Setting shadowed parameters for program 24
cgD3D(TRACE): Setting texture for sampler parameter
'BaseTexture'
cgD3D(TRACE): Setting SamplerState[0].D3DTSS_MAGFILTER for
sampler parameter 'BaseTexture'
cgD3D(TRACE): Setting SamplerState[0].D3DTSS_MINFILTER for
sampler parameter 'BaseTexture'
cgD3D(TRACE): Setting SamplerState[0].D3DTSS_MIPFILTER for
sampler parameter 'BaseTexture'
…
cgD3D(TRACE): Deleting vertex shader for program 3
cgD3D(TRACE): Deleting pixel shader for program 24
```

デバッグ DLL を使用するには、次のようにします。

1. アプリケーションを、**cgD3D9.lib**（または **cgD3D8.lib**）ではなく **cgD3D9d.lib**（または **cgD3D8d.lib**）にリンクします。
2. アプリケーションが **cgD3D9d.dll**（または **cgD3D8d.dll**）を参照できることを確認します。
3. **cgD3D9EnableDebugTracing()** を使用し、コードの各部分のトレースを有効または無効にします。

   void cgD3D9EnableDebugTracing(CGbool enable);

次に、アプリケーション コードの一部についてデバッグ トレースを有効にする方法を示します。

```
cgD3D9EnableDebugTracing(CG_TRUE);
// ...
// Application code that is traced
// ...
cgD3D9EnableDebugTracing(CG_FALSE);
```

各デバッグ トレース出力時には、エラーを **cgD3D9DebugTrace** に設定することに注意してください。このため、**cgSetErrorCallback()** を使用してエラー コールバックがコア ランタイムに登録されている場合は、各デバッグ トレース出力により、このエラー コールバックがコールされます（87 ページの「エラー コールバックの使用」を参照）。

Direct3D エラー レポート

Cg のエラー レポートには、定義済みのエラーの型、エラーをテストするための関数およびエラー コールバックのサポートが含まれます。

Direct3D のエラーの型

Direct3D ランタイムは、Cg コア ランタイムでレポートされる **CGerror** 型のエラーと Direct3D ランタイムでレポートされる **HRESULT** 型のエラーを生成します。さらに、次の 2 つのグループにリストする、Direct3D Cg ランタイムに固有のエラーを返します。

- **CGerror**
  - **cgD3D9Failed**: Direct3D ランタイム関数が、エラーを返す Direct3D コールを行った場合に設定されます。
  - **cgD3D9DebugTrace**: デバッグ DLL を使用してデバッグ メッセージがデバッグ コンソールに出力される場合に設定されます（84 ページの「Direct3D デバッグ モード」を参照）。
- **HRESULT**
  - **CGD3D9ERR_INVALIDPARAM**: パラメータ値を設定できない場合に返されます。
  - **CGD3D9ERR_INVALIDPROFILE**: 予期されていないプロファイルを持つプログラムが関数に渡された場合に返されます。
  - **CGD3D9ERR_INVALIDSAMPLERSTATE**: 有効な **sampler** ステートではない **D3DTEXTURESTAGESTATETYPE** 型のパラメータが **sampler** ステート関数に渡された場合に返されます。
  - **CGD3D9ERR_INVALIDVEREXDECL**: プログラムが拡張インタフェースでロードされ、与えられた宣言に互換性がない場合に返されます。
  - **CGD3D9ERR_NODEVICE**: 必要な Direct3D デバイスが 0 の場合に返されます。この状況は一般に、拡張インタフェース関数がコールされ、Direct3D デバイスが **cgD3D9SetDevice()** で設定されていない場合に発生します。
  - **CGD3D9ERR_NOTMATRIX**: 行列型を要求する関数に行列型ではないパラメータが渡された場合に返されます。
  - **CGD3D9ERR_NOTLOADED**: **cgD3D9LoadProgram()** により拡張インタフェースでパラメータがロードされていない場合に返されます。
  - **CGD3D9ERR_NOTSAMPLER**: **sampler** パラメータを要求する関数に **sampler** ではないパラメータが渡された場合に返されます。
  - **CGD3D9ERR_NOTUNIFORM**: uniform 型を要求する関数に uniform 型ではないパラメータが渡された場合に返されます。
  - **CGD3D9ERR_NULLVALUE**: 0 以外の値を要求する関数に 0 の値が渡された場合に返されます。

- **CGD3D9ERR_OUTOFRANGE**: 関数に対して指定された配列範囲が範囲外の場合に返されます。
- **CGD3D9_INVALID_REG**: 無効なパラメータ型に対してレジスタ番号が要求されている場合に返されます。このエラーは、最小インタフェース関数に固有であり、エラー コールバックを起動しません。

エラーのテスト

HRESULT 型のエラーを返す Direct3D ランタイム関数がコールされた場合、正常終了かエラーかをテストする適切な方法は、Win32 マクロ **FAILED()** および **SUCCEEDED()** を使用することです。複数の正常終了値が存在することがあるため、0 か **D3D_OK** かでエラーをテストするだけでは不十分です。

便利さを向上させるため、またコア ランタイムとの統一を図るために、Direct3D ランタイムにも **cgGetLastError()** に類似した **cgD3D9GetLastError()** が用意されています。ただし、これは **FAILED()** マクロが **TRUE** を返す **HRESULT** 型の直前の Direct3D ランタイム エラーを返します。

```
HRESULT cgD3D9GetLastError();
```

直前のエラーは、コールの直後に必ずクリアされます。

関数 **cgD3D9TranslateHRESULT()** は、**HRESULT** 型のエラーを文字列に変換します。

```
const char* cgD3D9TranslateHRESULT(HRESULT hr);
```

この関数も Cg Direct3D ランタイムが生成するエラーを変換するため、この関数は **DXGetErrorDescription9()** のかわりにコールする必要があります。

エラー コールバックの使用

次に、コア ランタイム エラーからのデバッグ トレースと、Direct3D ランタイム エラーからのデバッグ トレースを分類するエラー コールバックの例を示します。

```
void MyErrorCallback() {
  CGerror error = cgGetError();
  if (error == cgD3D9DebugTrace) {
  // This is a debug trace output.
  // A breakpoint could be set here to step from one
  // debug output to the other.
    return;
  }
  char buffer[1024];
  if (error == cgD3D9Failed)
    sprintf(buffer, "A Direct3D error occurred: %s'\n",
            cgD3D9TranslateHRESULT(cgD3D9GetLastError()));
  else
    sprintf(buffer, "A Cg error occurred: '%s'\n",
            cgD3D9TranslateCGerror(error));
  OutputDebugString(buffer);
}
cgSetErrorCallback(MyErrorCallback);
```

簡単なチュートリアル

この章では、提供されているサンプルの Cg Microsoft Visual Studio ワークスペースを、実験的に使用できる単純な Cg プログラムとあわせて、順を追って説明します。

ワークスペースのロード

Cg_Simple ファイルをロードすると、ワークスペースが図 3 のように表示されます。

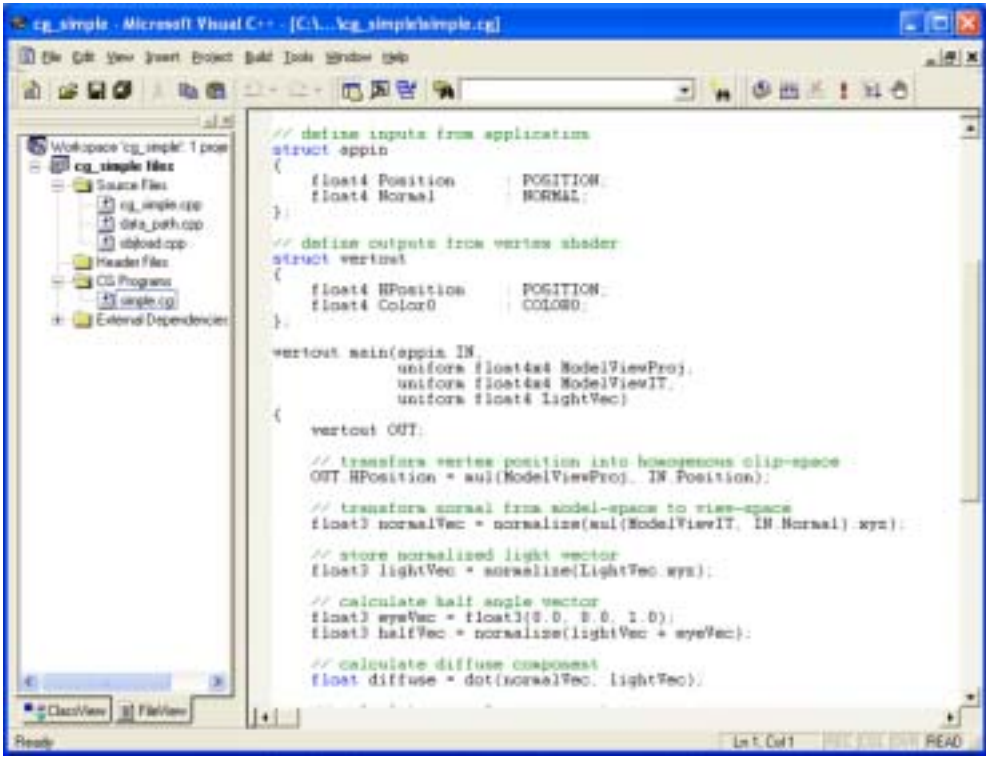


図 3 Cg_Simple ワークスペース

通常どおり、「FileView」タブをクリックしてプロジェクト内の各種ファイルを表示します。ただしここでは、通常の「Source Files」フォルダと「Header Files」フォルダに加えて、「Cg Programs」フォルダもある点が異なります。

この「Cg Program」フォルダには、実験に使用できる **simple.cg** という 1 つの Cg プログラムが含まれます。**simple.cg** を編集する場合は、このプログラムをダブルクリックして開きます。**simple.cg** の編集中は、**[Ctrl]** キーを押しながら **[F7]** キーを押して、いつでもコンパイルが可能です。プロジェクトの設定方法のため、コード内のエラーは、普通の C または C++ プログラムのコンパイル時と同様に表示されます。

エラーをダブルクリックして、ソース コード内でエラーの原因となった場所を参照することもできます。

simple.cg について

Cg_Simple アプリケーションでは、**simple.cg** で定義されているシェーダをトーラスに対して実行します。提供されているバージョンの **simple.cg** は、各頂点のディフューズ ライティングとスペキュラ ライティングを計算します。図 4 は、このシェーダのスクリーンショットです。

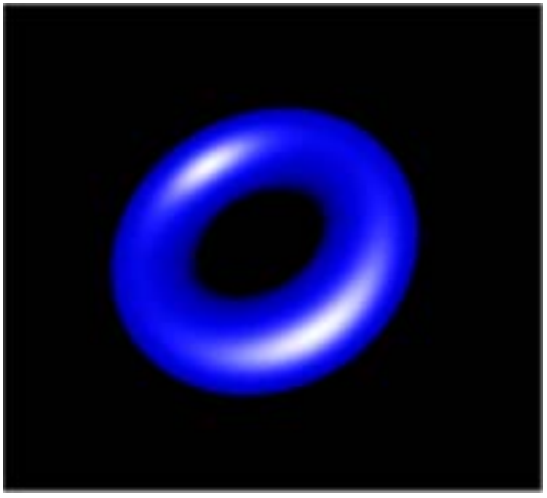


図 4　simple.cg **シェーダ**

simple.cg のプログラム リスティング

次に、**simple.cg** のプログラム リストを示します。

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

// Define outputs from vertex shader.
struct vertout
{
  float4 HPosition    : POSITION;
  float4 Color        : COLOR;
};

vertout main(appin IN,
             uniform float4x4 ModelViewProj,
             uniform float4x4 ModelViewIT,
             uniform float4 LightVec)
{
  vertout OUT;

  // Transform vertex position into homogenous clip-space.
  OUT.HPosition = mul(ModelViewProj, IN.Position);

  // Transform normal from model-space to view-space.
  float3 normalVec = normalize(mul(ModelViewIT,
                                   IN.Normal).xyz);

  // Store normalized light vector.
  float3 lightVec = normalize(LightVec.xyz);

  // Calculate half angle vector.
  float3 eyeVec = float3(0.0, 0.0, 1.0);
  float3 halfVec = normalize(lightVec + eyeVec);

  // Calculate diffuse component.
  float diffuse = dot(normalVec, lightVec);

  // Calculate specular component.
  float specular = dot(normalVec, halfVec);

  // Use the lit function to compute lighting vector from
```

```
  // diffuse and specular values.
  float4 lighting = lit(diffuse, specular, 32);

  // Blue diffuse material
  float3 diffuseMaterial = float3(0.0, 0.0, 1.0);

  // White specular material
  float3 specularMaterial = float3(1.0, 1.0, 1.0);

  // Combine diffuse and specular contributions and
  // output final vertex color.
  OUT.Color.rgb = lighting.y * diffuseMaterial +
                  lighting.z * specularMaterial;
  OUT.Color.a = 1.0;

  return OUT;
}
```

varying 型データを持つ構造体の定義

最初に注意することは、varying 型データに対するバインディング セマンティクスを持つ構造体の定義です。

appin 構造体を見てみましょう。

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

この構造体には、**Position** および **Normal** という 2 つのメンバのみが含まれています。このデータは頂点ごとに変化するため、バインディング セマンティクス **POSITION** および **NORMAL** は、座標情報を事前定義済みの属性 **POSITION** に関連付け、法線情報を事前定義済みの属性 **NORMAL** に関連付けるようコンパイラに指示します。

simple.cg で定義されているもう 1 つの構造体は **vertout** で、これは頂点からフラグメントへのコネクタです。

```
// define outputs from vertex shader
struct vertout
{
    float4 HPosition    : POSITION;
    float4 Color        : COLOR;
};
```

vertout 構造体にも、**Hposition**（同次座標での頂点座標）および **Color**（頂点の色）という 2 つのメンバのみが含まれています。ここでも、変数のレジスタ位置を指定するためにバインディング セマンティクスが使用されています。この場合、同次座標情報は **POSITION** に対応するハードウェア レジスタにあり、カラー情報は **COLOR** に対応するハードウェア レジスタにあります。

引数の受渡し

次に、**main()** の宣言から順に、プログラムの本体をセクションごとに見てみましょう。

```
vertout main(appin IN,
             uniform float4x4 ModelViewProj,
             uniform float4x4 ModelViewIT,
             uniform float4 LightVec)
```

頂点プログラムに必要であるため、**main()** はアプリケーション－頂点の構造体を入力として受け取り、頂点－フラグメントの構造体を返します。この場合は、すでに定義した **appin** および **vertout** という 2 つのタイプの構造体を使用しています。**main()** は 3 つの uniform 型パラメータを受け取ることに注意してください。2 つは行列、1 つはベクトルです。3 つのパラメータはすべて、ランタイム ライブラリを使用してアプリケーションにより **simple.cg** に渡されます。

最初の行列 **ModelViewProj** は、モデルビュー行列と射影行列を連結したものです。連結した行列によって、ポイントをモデル空間からクリップ空間に変換します。2 番目の行列 **ModelViewIT** は、モデルビュー行列の逆転置行列です。3 番目のパラメータ **LightVec** は、光源の位置を指定するベクトルです。

基本変換

次に、頂点プログラムの本体を説明します。

```
    vertout OUT;

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

頂点プログラムは、頂点の同次クリップ空間座標を計算します（モデル空間で与えられた頂点座標を変換します）。したがって、頂点のモデル空間座標（**IN.Position** で指定される）は、モデルビュー行列と射影行列の連結（この例では **ModelViewProj** と呼ぶ）によって変換する必要があります。変換された座標は、**OUT.HPosition** に直接代入されます。頂点プログラムを使用する場合は、パースペクティブ除算を行う必要はありません。ハードウェアが、頂点プログラムを実行した後に、分割を自動的に実行します。

視点空間でライティングを行う必要があるため、モデル空間法線 **IN.Normal** を視点空間に変換する必要があります。

```
// transform normal from model-space to view-space
 float3 normalVec = normalize(mul(ModelViewIT,
                                  IN.Normal).xyz);
```

法線を変換する際には、モデルビュー行列の逆転置行列によって乗算する必要があります。次に、視点空間法線ベクトルを正規化し、**normalVec** として格納します。

ライティングの準備

次のステップでは、ライティングを準備します。

```
 // store normalized light vector
 float3 lightVec = normalize(LightVec.xyz);

 // calculate half angle vector
 float3 eyeVec = float3(0.0, 0.0, 1.0);
 float3 halfVec = normalize(lightVec + eyeVec);
```

この時点で、すべてのベクトルを正規化する必要があります。**LightVec**[1] の正規化から開始します。次に、スペキュラ ライティングの準備として、光ベクトルと視点ベクトルの中間のベクトル（つまり、**(lightVec+eyeVec)/2**）であるハーフアングル ベクトル **halfVec** を定義する必要があります。**halfVec** を正規化するため、2 で割る必要はありません。正規化においては冗長な処理だからです。この例では、視点が **(0, 0, 1)** にあると想定していますが、視点座標は頂点ごとに変化するものではないため、アプリケーションでは視点座標を uniform 型パラメータとしても渡すのが一般的です。ここでは Cg のインライン ベクトル コンストラクタを使用して、視点座標を含む 3 成分 **float** ベクトルを作成し、この値を **eyeVec** に代入します。

1. **LightVec** は uniform 型であるため、頂点ごとに正規化するよりも、アプリケーションで 1 回のみ正規化するほうが効率的です。ここでは、例証の目的でこの処理を行っています。

頂点の色の計算

次に、出力する頂点の色を計算する必要があります。

ディフューズ ライティングとスペキュラ ライティングからの寄与の計算

この例では、ディフューズ ライティングとスペキュラ ライティングの単純な混合を計算します。

```
// calculate diffuse component
float diffuse = dot(normalVec, lightVec);

// calculate specular component
float specular = dot(normalVec, halfVec);

// Use the lit function to compute lighting vector from
// diffuse and specular values
float4 lighting = lit(diffuse, specular, 32);
```

ここでは、Cg 標準ライブラリを使用して、内積（**dot()** を使用）を実行します。標準ライブラリの **lit()** 関数を使用し、計算済みの内積に基づいて Blinn スタイルのライティング ベクトルも計算します。返されるベクトルでは、y 座標にディフューズ ライティングからの寄与、z 座標にスペキュラ ライティングからの寄与が保持されます。

標準ライブラリを利用して、開発サイクルをスピードアップすることを忘れないでください。

ディフューズ ライティングとスペキュラ ライティングからの寄与の変調

ディフューズ ライティングとスペキュラ ライティングからの寄与である **lighting.y** および **lighting.z** が計算された後で、オブジェクトのマテリアル特性を使用して変調する必要があります。

```
// blue diffuse material
float3 diffuseMaterial = float3(0.0, 0.0, 1.0);

// white specular material
float3 specularMaterial = float3(1.0, 1.0, 1.0);

// combine diffuse and specular contributions and
// output final vertex color
OUT.Color.rgb = lighting.y * diffuseMaterial +
                lighting.z * specularMaterial;
OUT.Color.a = 1.0;

return OUT;
```

オブジェクトのディフューズ マテリアル色を青として定義します。ライティングからの寄与をマテリアル特性で変調して、最終的な頂点色を計算し、それを出力構造体の色フィールド **OUT.Color** に代入します。最後に、最終的な色のアルファ チャネルを 1.0 に設定してオブジェクトが不透明になるようにし、**OUT** 構造体に格納された計算済みの座標と色の値を返します。

その他の実験

simple.cg をフレームワークとして使用し、他のパラメータをプログラムに追加したり、頂点プログラムでより複雑な計算を実行したりすることにより、高度な実験を行いましょう。実験をお楽しみください！

拡張プロファイルのサンプル シェーダ

この章では、Cg で記述された拡張プロファイルのサンプル シェーダを紹介します。各シェーダにはスナップショット、説明およびソース コードが付いています。

サンプルは次のとおりです。

- 改善されたスキニング
- 改善された水面
- 融解ペイント
- マルチペイント
- レイ トレーシングによる屈折
- 皮膚
- 薄膜エフェクト
- カー ペイント 9

改善されたスキニング

説明

このシェーダは、特定のボーンに影響する可能性のあるすべての変換行列のセットを受け取ります。各ボーンは、そのボーンに影響する行列のリストも送ります。次に、各頂点について、その頂点に影響する各ボーンを調べて変換する単純なループがあります。こうすることによって、ボーンが 1 つの場合にはあるプログラムを使用し、ボーンが 2 つの場合には別のプログラムを使用するなどではなく、任意の数のボーンに影響される頂点のスキニング処理のすべてを 1 つの Cg プログラムのみで実行できます。

図 5　改善されたスキニングの例

改善されたスキニングの頂点シェーダ ソース コード

```
struct inputs
{
  float4 position       : POSITION;
  float4 weights        : BLENDWEIGHT;
  float4 normal         : NORMAL;
  float4 matrixIndices  : TESSFACTOR;
  float4 numBones       : SPECULAR;
};

struct outputs
{
  float4 hPosition    : POSITION;
  float4 color        : COLOR0;
};

outputs main(inputs IN,
       uniform float4x4 modelViewProj,
       uniform float3x4 boneMatrices[30],
       uniform float4 color,
       uniform float4 lightPos)
{
  outputs OUT;

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

  float4 position;
  float3 normal;

  for (float i = 0; i < IN.numBones.x; i += 1) {
    // transform the offset by bone i
    position = position + weight.x *
      float4(mul(boneMatrices[index.x], IN.position).xyz,
             1.0);

    // transform normal by bone i
    normal = normal + weight.x *
        mul((float3x3)boneMatrices[index.x],
            IN.normal.xyz).xyz;

    // shift over the index/weight variables; this moves
    // the index and  weight for the current bone into
```

```
    // the .x component of the index and weight variables
    index = index.yzwx;
    weight = weight.yzwx;
  }

  normal = normalize(normal);

  OUT.hPosition = mul(modelViewProj, position);
  OUT.color = dot(normal, lightPos.xyz) * color;

  return OUT;
}
```

改善された水面

説明

このデモでは、(自由な回転により)大きな格子状の頂点で取り囲まれているシーンのように見えますが、ワイヤフレームに切り換えたりフラスタム角度を大きくしたりすると、頂点は静的メッシュであり、その高さ、法線およびテクスチャ座標は視線方向と視点の高さに基づいてその場で計算されていることが明らかになります。この手法では、静的メッシュを事前に計算できるため、GPU を有効に使用する水面アニメーションが可能になります。頂点はサイン波を使用して置換され、この例では、ループを使用して 5 つのサイン波を合計し、写実的なエフェクトを実現しています。

図 6　改善された水面の例

改善された水面の頂点シェーダ ソース コード

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

struct vert2frag
{
  float4 HPosition  : POSITION;
  float4 TexCoord0  : TEXCOORD0;
  float4 TexCoord1  : TEXCOORD1;
  float4 Color0     : COLOR0;
  float4 Color1     : COLOR1;
};

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 = 1 + dot(position.xyz, position.xyz)/1000;
float i, disp;
float2 norm;

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;
}

OUT.HPosition = mul(ModelViewProj, position);

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

// 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);

// output two reflection vectors for the two
// environment cubemaps
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), these two color coefficients are used
  // to calculate the contribution from each of the two
  // environment cubemaps (one bright, one dark)
  OUT.Color0 = (fres*1.4 + min(reflected.y,0)).xxxx +
    float4(.2,.3,.3,0);
  OUT.Color1 = (fres*1.26).xxxx;

  return OUT;
}
```

改善された水面のピクセル シェーダ ソース コード

```
float4 main(in float3 color0          : COLOR0,
            in float3 color1          : COLOR1,
            in float3 reflectVec      : TEXCOORD0,
            in float3 reflectVecDark  : TEXCOORD1,
            uniform samplerCUBE environmentMaps[2]
            ) : COLOR
{
  float3 reflectColor = texCUBE(environmentMaps[0],
      reflectVec).rgb;
  float3 reflectColorDark = texCUBE(environmentMaps[1],
      reflectVecDark).rgb;

  float3 color = (reflectColor * color0) +
      (reflectColorDark * color1);
  return float4(color, 1.0);
}
```

融解ペイント

説明

このシェーダは、プロシージャルに修正されるテクスチャ ルックアップで環境マップを使用し、表面テクスチャ（この例では NVIDIA ロゴ）に対する融解エフェクトを作成します。反射ベクトルがノイズ関数を使用してシフトされ、表面にでこぼこの外観を与えます。表面テクスチャのテクスチャ座標も、ノイズ テクスチャに基づいて時間依存方式でシフトされます。

図 7　融解ペイントの例

融解ペイントの頂点シェーダ ソース コード

```
// define inputs from application
struct app2vert
{
    float4 Position     : POSITION;
    float4 Normal       : NORMAL;
```

```
    float4 Color0       : COLOR0;
    float4 TexCoord0    : TEXCOORD0;
};

struct vert2frag
{
    float4 HPosition    : POSITION;
    float3 OPosition    : TEXCOORD2;
    float3 EPosition    : TEXCOORD3;
    float3 Normal       : TEXCOORD1;
    float3 TexCoord0    : TEXCOORD0;
    float4 Color0       : COLOR0;

    float3 LightPos     : TEXCOORD4;
    float3 ViewerPos    : TEXCOORD5;
};

vert2frag main(app2vert In,
               uniform float4x4 ModelViewProj,
               uniform float4x4 ModelView,
               uniform float4x4 ModelViewI,
               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.EPosition = mul(ModelView, In.Position).xyz;

    Out.Normal = normalize(In.Normal.xyz);
    // Copy the texture coordinates
    Out.TexCoord0 = In.TexCoord0.xyz;
    // Generate a white color
    Out.Color0 = LightPos;

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

    return Out;
}
```

融解ペイントのピクセル シェーダ ソース コード

```
struct vert2frag
{
  float4 HPosition  : POSITION;
  float3 OPosition  : TEXCOORD2;
  float3 EPosition  : TEXCOORD3;
  float3 Normal     : TEXCOORD1;
  float3 TexCoord0  : TEXCOORD0;
  float4 Color0     : COLOR0;

  float3 LightPos   : TEXCOORD4;
  float3 ViewerPos  : TEXCOORD5;
};

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(eyePos + light);

  float attenuation = 1. / (.3 * len);

  float4 lighting = lit(dot(light, normal),
      dot(halfVec, normal), specularExp);
  diffuse = lighting.y * attenuation;
  specular = lighting.z * attenuation;
}

float4 main(vert2frag IN,
      uniform float4  LightPos,
      uniform sampler3D noise_map,
      uniform sampler2D nv_map,
      uniform samplerCUBE cube_map,
      uniform float4  interpolate
      ) : COLOR
{
  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 noiseTex = float3((IN.OPosition.x+IN.OPosition.z)*6,
    IN.OPosition.y/2, 0);
  float3 noiseSum = tex3D(noise_map, biVariate/3).rgb/12 +
      tex3D(noise_map, noiseTex).rgb/18 +
            tex3D(noise_map, biVariate*6).rgb/18;
  normal = normalize(normal + noiseSum);

  calcLighting(diffuse, specular, normal, IN.OPosition,
    IN.LightPos, IN.ViewerPos, 32);

  float3 nvShift = tex3D(noise_map, uniVariate/3).rgb / 2 +
           tex3D(noise_map, uniVariate).rgb / 4 +
           tex3D(noise_map, biVariate*3).rgb / 16;
  nvShift.x = nvShift.x*nvShift.x * interpolate.x * 3;
  nvShift.y = 0;

  biVariate = float3(IN.OPosition.x - IN.OPosition.z,
    IN.OPosition.y, 0);
  float2 texCoord = biVariate.xy/4 + float2(1.1, .5) +
    nvShift.yx + float2(0, interpolate.x/8);
  float3 nvDecal =
    tex2D(nv_map, float2(1-texCoord.x, texCoord.y)).rgb *
      (1-interpolate.x * .7).xxx;

  float3 eye = IN.ViewerPos - IN.OPosition;
  float3 lightMetal = texCUBE(cube_map,
                              reflect(normal, eye)).rgb;
  float3 darkMetal = (diffuse * float3(.5,.25,0) +
    specular * float3(.7,.4,0));

  float3 finalColor = lerp(lightMetal, darkMetal, nvDecal.x);
  return float4(finalColor, 1);
}
```

マルチペイント

説明

マルチペイントは、1 つの多角形の表面に複数種類の素材が混在するという、一般的な問題に対して、単一パスによる解決策を提示します。マルチペイントは、多くの一般的な金属面や誘電面を表現することが可能な単純 BRDF（双方向反射分布関数）を提供し、テクスチャリングを介して可変 BRDF のすべての主要要素を制御します。これによって、シェーダを切り換えたりモデルを分割したり、あるいはマルチパスを使用したりせずに、複数の素材を作成できます。

マルチペイントの用途としては、金属、木および石をはめ込んだ鎧（すべて単一の単純ポリメッシュでモデリングされる）、単純なキューブとして表現される複数タイプの石、ガラスおよび金属から構成される建物、金属繊維を織り込んだ衣服、あるいはこのデモで示すような部分的に塗装がはがれた金属などがあります。

リアルタイム処理以外の世界では複数の BRDF の使用は一般的ですが、最適化されているとは言えません。2 つの異なるシェーダを実行し、その結果をマスク テクスチャを使用してブレンドするか、**if** ステートメントを介してつなぎ合わせているといった具合です。リアルタイムで最大限のパフォーマンスを実現するために、マルチペイントでは、複数のペイントが行われたテクスチャとして BRDF のすべての主要部分を統合するため、様々な要素が混在する外観を作成する際にはシェーダを 1 回実行するだけで済みます。これによって、コンパクトな高速実行パッケージで、ディフューズ ライティング、スペキュラ ライティングおよび環境ライティングのエフェクトを含む単一パス シェーダを実現できます。

図 8　マルチペイントの例

マルチペイントの頂点シェーダ ソース コード

```
// define inputs from vertex buffer
struct appin
{
  float4 Position     : POSITION;
  float4 UV           : TEXCOORD0;
  float4 Tangent      : TEXCOORD1;
  float4 Binormal     : TEXCOORD2;
  float4 Normal       : TEXCOORD3;
};

// output -- same struct is the input to "cg_multipaint.cg"
struct MultiPaintV2F {
  float4 HPosition    : POSITION;  // position (clip space)
  float4 TexCoords    : TEXCOORD0; // base ST coordinates
  float3 OPosition    : TEXCOORD1; // position (obj space)
  float3 Normal       : TEXCOORD2; // normal (eye space)
  float3 VPosition    : TEXCOORD3; // view pos (obj space)
  float3 T            : TEXCOORD4; // tangent (obj space)
  float3 B            : TEXCOORD5; // binormal (obj space)
  float3 N            : TEXCOORD6; // normal (obj space)
  float4 LightVecO    : TEXCOORD7; // light dir (obj space)
};

MultiPaintV2F main(appin IN,
                   uniform float4x4 ModelViewProj,
                   uniform float4x4 ModelViewIT,
                   uniform float4x4 ModelViewI,
                   uniform float4 TexRepeats,
                   uniform float4 LightVec) // (eye space)
{
  MultiPaintV2F OUT;

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

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

  // transform normal to eye space
  OUT.Normal = normalize(mul(ModelViewIT, IN.Normal).xyz);

  OUT.TexCoords = IN.UV * TexRepeats;
```

```
  // pass through object-space normal, tangent, binormal.
  OUT.N = normalize(IN.Normal.xyz);
  OUT.T = IN.Tangent.xyz;
  OUT.B = IN.Binormal.xyz;

  // transform view pos (origin) to obj space
  OUT.VPosition = mul(ModelViewI, float4(0,0,0,1)).xyz;

  // transform light vector to obj space
  OUT.LightVecO = mul(ModelViewI, LightVec);

  return OUT;
}
```

マルチペイントのピクセル シェーダ ソース コード

```
#define WHITE half4(1.0h,1.0h,1.0h,1.0h)

// input -- same struct is output from "cg_multipaintVP.cg"
struct MultiPaintV2F {
  float4 HPosition    : POSITION;  // position (clip space)
  float4 TexCoords    : TEXCOORD0; // base ST coordinates
  float3 OPosition    : TEXCOORD1; // position (obj space)
  float3 Normal       : TEXCOORD2; // normal (eye space)
  float3 VPosition    : TEXCOORD3; // view pos (obj space)
  float3 T            : TEXCOORD4; // tangent (obj space)
  float3 B            : TEXCOORD5; // binormal (obj space)
  float3 N            : TEXCOORD6; // normal (obj space)
  float4 LightVecO    : TEXCOORD7; // light dir (obj space)
};

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

// subfields in "SpecData"
#define MINPOWER x
#define MAXPOWER y
#define MAXSPEC z

// subfields in "ReflData"
#define FRESNEL_MIN x
#define FRESNEL_MAX y
```

```
#define FRESNEL_EXPON z
#define REFL_STRENGTH w

// subfields in "BumpData"
#define BUMP_SCALE x

half4 main(MultiPaintV2F IN,
     uniform sampler2D ColorMap,    // color
     uniform sampler2D MaterialMap, // see above
     uniform sampler2D NormalMap,   // tangent-space normals
     uniform samplerCUBE EnvMap,    // environment skybox
     uniform float4 SpecData,       // see above
     uniform float4 ReflData,       // see above
     uniform float4 BumpData        // see above
     ) : COLOR
{
  half4 surfCol = tex2D(ColorMap, IN.TexCoords.xy);
  half4 material = tex2D(MaterialMap, IN.TexCoords.xy);
  half3 Nt = tex2D(NormalMap, IN.TexCoords.xy).rgb -
             half3(0.5h,0.5h,0.5h);

  // SpecData.MAXSPEC *should* range from 0 - 1.
  half specStr = material.SPEC_STR * SpecData.MAXSPEC;
  half specPower = SpecData.MINPOWER +
                   material.NORM_SPEC_EXPON *
                   (SpecData.MAXPOWER - SpecData.MINPOWER);

  half3 Vn = -normalize(IN.VPosition - IN.OPosition);
  half3 Ln = normalize(IN.LightVecO).xyz;
  half3 Nb = normalize(BumpData.BUMP_SCALE *
                       (Nt.x*IN.T + Nt.y*IN.B) +
                       (Nt.z*IN.N));

  half  diff = dot(-Ln, Nb);
  half3 Hn = -normalize(Vn + Ln);
  half4 lighting = lit(diff, dot(Hn, Nb), specPower);

  half4 diffResult = lighting.y * surfCol;
  half4 specCol = lerp(WHITE, surfCol, material.METALNESS);
  half4 specResult = lighting.z * specStr * specCol;

  half3 reflVect = reflect(Vn, Nb);
  half4 reflColor = texCUBE(EnvMap, reflVect);
  half fakeFresnel = ReflData.FRESNEL_MIN +
                     ReflData.FRESNEL_MAX *
```

```
                    pow(saturate(1.0h-dot(-Vn,IN.N)),
                        ReflData.FRESNEL_EXPON);
  half4 paintShine = fakeFresnel * reflColor;
  half4 metalShine = surfCol * reflColor;
  half4 shineCol = ReflData.REFL_STRENGTH *
                   lerp(paintShine, metalShine,
                        material.METALNESS);

  half4 finalColor = specResult + diffResult + shineCol;
  finalColor.w = 1.0h;

  return finalColor;
}
```

レイ トレーシングによる屈折

説明

このシェーダでは、1 回の反射を行うレイ トレーシングを使用し、小さなオブジェクトに高品質のディテールを追加する方法を示します。この例では、多角形の表面がサンプリングされ、屈折ベクトルが計算されています。このベクトルは、オブジェクトの x 軸に直交する面と交差します。計算された交点は、虹彩を着色するためのテクスチャ インデックスとして使用されます。

デモでは、レンズの屈折、深度および密度のインデックスを変化させることができます。ジオメトリの選択は任意であることに注意してください。このサンプルは球体ですが、任意の多角形モデルを使用できます。

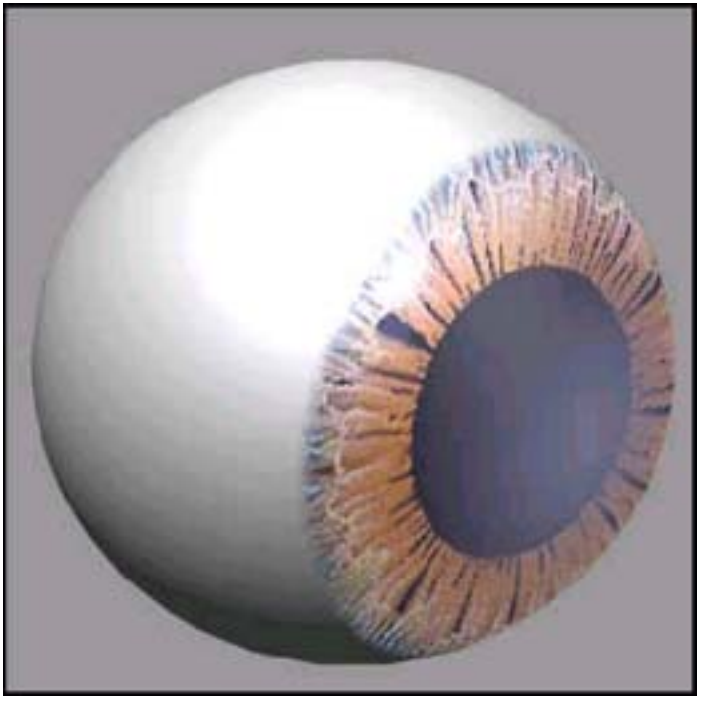


図 9　レイ トレーシングによる屈折の例

レイ トレーシングによる屈折の頂点シェーダ ソース コード

```
struct appin
{
  float4 Position   : POSITION;
  float4 Normal     : NORMAL;
};

// output -- same struct is the input to fragment shader
struct EyeV2F {
  float4 HPosition  : POSITION;  // clip space pos
  float3 OPosition  : TEXCOORD0; // Obj-coords location
  float3 VPosition  : TEXCOORD1; // eye pos (obj space)
  float3 N          : TEXCOORD2; // normal (obj space)
  float4 LightVecO  : TEXCOORD3; // light dir (obj sp)
};

EyeV2F main(appin IN,
      uniform float4x4 ModelViewProj,
      uniform float4x4 ModelViewI,
      uniform float4 LightVec)  // in EYE coords
{
  EyeV2F OUT;

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

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

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

  // transform view pos and light vec to obj space
  OUT.VPosition = mul(ModelViewI, float4(0,0,0,1)).xyz;
  OUT.LightVecO = normalize(mul(ModelViewI, LightVec));

  return OUT;
}
```

レイ トレーシングによる屈折のピクセル シェーダ ソース コード

```
// Assume ray direction is normalized.
// Vector "planeEq" is encoded half3(A,B,C,D) where
// (Ax+By+Cz+D)=0 and half3(A,B,C) has been normalized.
// Returns distance along to to intersection; distance is
// negative if no intersection.
half intersect_plane(half3 rayOrigin,half3 rayDir,
                     half4 planeEq) {
  half3 planeN = planeEq.xyz;
  half denominator = dot(planeN, rayDir);
  half result = -1.0h;

  // d==0 -> parallel || d>0 -> faces away
  if (denominator < 0.0h) {
    half top = dot(planeN,rayOrigin) + planeEq.w;
    result = -top/denominator;
  }
  return result;
}

// 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

struct EyeV2F {
  float4 HPosition  : POSITION;
  float3 OPosition  : TEXCOORD0;
  float3 VPosition  : TEXCOORD1;
  float3 N          : TEXCOORD2;
  float4 LightVecO  : TEXCOORD3;
};

half4 main(EyeV2F IN,
  uniform sampler2D   ColorMap, // color
  // components: {radius,irisDepth,eta,lensDensity)
```

```
  uniform float4 BallData,
  // components: {phongExp,gloss1,gloss2,drop)
  uniform float4 GlossData,
  uniform float3 AmbiColor,
  uniform float3 DiffColor,
  uniform float3 SpecColor,
  uniform float3 LensColor,
  uniform float3 BgColor) : COLOR
{
  const half3 baseTex = half3(1.0h,1.0h,1.0h);
  const half GRADE = 0.05h;
  const half3 yAxis = half3(0.0h,1.0h,0.0h);
  const half3 xAxis = half3(1.0h,0.0h,0.0h);
  const half3 ballCtr = half3(0.0h,0.0h,0.0h);

  // (actually constants - could be done in VP or on CPU)
  half irisSize = BallData.RADIUS *
    sqrt(1.0h-BallData.IRIS_DEPTH * BallData.IRIS_DEPTH);
  half irisScale = 0.3333h / max(0.01h, irisSize);
  half irisDist = BallData.RADIUS * BallData.IRIS_DEPTH;
  half3 pupilCenter = ballCtr + half3(irisDist,0.0h,0.0h);
  // if x axis, returns simple -irisDist
  half D = -dot(pupilCenter, xAxis);
  half slice = IN.OPosition.x - irisDist;
  half4 planeEquation = half4(xAxis, D);

  // view vector TO surface
  half3 Vn = normalize(IN.OPosition - IN.VPosition);
  half3 Nf = normalize(IN.N);
  half3 Ln = IN.LightVecO.xyz;
  half3 DiffLight = DiffColor * saturate(dot(Nf, -Ln));
  half3 missColor = AmbiColor + baseTex * DiffLight;
  half3 DiffPupil = AmbiColor + saturate(dot(xAxis, -Ln));

  half3 halfAng = normalize(-Ln - Vn);
  half ndh = abs(dot(Nf,halfAng));
  half spec1 = pow(ndh, GlossData.PHONG);
  half s2 = smoothstep(GlossData.GLOSS1, GlossData.GLOSS2,
    spec1);
  spec1 = lerp(GlossData.DROP, spec1, s2);
  half3 SpecularLight = SpecColor * spec1;

  half3 hitColor = missColor;

  if (slice >= 0.0h) {
```

```
    half gradedEta = BallData.ETA;
    gradedEta = 1.0h/gradedEta;
    half3 faceColor = BgColor;

    half3 refVector = refract(Vn, Nf, gradedEta);
    if (dot(refVector, refVector) > 0) {
      // now let's intersect with the iris plane
      half irisT = intersect_plane(IN.OPosition, refVector,
          planeEquation);
      half fadeT = irisT * BallData.LENS_DENSITY;
      fadeT = fadeT * fadeT;
      faceColor = DiffPupil.xxx;
      if (irisT > 0) {
        half3 irisPoint = IN.OPosition + irisT*refVector;
        half3 irisST = (irisScale*irisPoint) +
          half3(0.0h, 0.5h, 0.5h);
        faceColor = tex2D(ColorMap, irisST.yz).rgb;
      }
      faceColor = lerp(faceColor, LensColor, fadeT);
      hitColor = lerp(missColor, faceColor,
        smoothstep(0.0h, GRADE, slice));
    }
  }

  hitColor = hitColor + SpecularLight;
  return half4(hitColor, 1.0h);
}
```

皮膚

説明

このエフェクトは、単純な Blinn-Phong バンプ マッピングから、より複雑な表面下散乱ライティング モデルまでの皮膚のレンダリング手法を示します。複合的、非局所的なライティングの相互作用による皮膚の微細な特性をいくつか取得するためのリム ライティングと、単純な半透明度の使用法も示します。最後に、様々な手法を組み合せて、説得力のある様式化された皮膚を生成する方法を示します。

図 10 **皮膚の例**

皮膚のピクセル シェーダ ソース コード

```
struct fragin
{
  float2 texcoords         : TEXCOORD0;
  float4 shadowcoords      : TEXCOORD1;
```

```
  float4 tangentToEyeMat0  : TEXCOORD4;
  float3 tangentToEyeMat1  : TEXCOORD5;
  float3 tangentToEyeMat2  : TEXCOORD6;
  float3 eyeSpacePosition  : TEXCOORD7;
};

float3 hgphase( 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 * hgphase( 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 Compendeum.)
  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;
}

float4 main( fragin In,
  uniform sampler2D tex0,
  uniform sampler2D tex1,
  uniform sampler2D tex2,
  uniform sampler2D tex3,
  uniform float3 eyeSpaceLightPosition,
  uniform float thickness,
  uniform float4 ambient ) : COLOR
```

```
{
  float bscale = In.tangentToEyeMat0.w;

  float eta = (1.0/1.4);

  // ratio of indices of refraction (air/skin)
  float  m = 34.;                      // specular exponent
  float4 lightColor = { 1, 1, 1, 1 };  // light color
  float4 sheenColor = { 1, 1, 1, 1 };  // sheen color
  float4 skinColor  = tex2D( tex1, In.texcoords );
  float3 g          = { 0.8, 0.3, 0.0 };
  float3 albedo     = { 0.8, 0.5, 0.4 };

  // oiliness mask
  float4 oiliness = 0.9 * tex2D( tex2, In.texcoords);

  // Get eye-space eye vector.
  float3 v = normalize( -In.eyeSpacePosition );

  // Get eye-space light and halfangle vectors.
  float3 l = normalize( eyeSpaceLightPosition -
                        In.eyeSpacePosition );
  float3 h = normalize( v + l );

  // Get tangent-space normal vector from normal map.
  float3 tangentSpaceNormal = tex2D(tex0, In.texcoords).rgb;
  float3 bumpscale = { bscale, bscale, 1.0 };
  tangentSpaceNormal = tangentSpaceNormal * bumpscale;

  // Transform it into eye-space.
  float3 n;
  n[0] = dot( In.tangentToEyeMat0.xyz, tangentSpaceNormal );
  n[1] = dot( In.tangentToEyeMat1, tangentSpaceNormal );
  n[2] = dot( In.tangentToEyeMat2, tangentSpaceNormal );
  n = normalize( n );

  // Compute the lighting equation.
  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);

  // Compute oil, sheen, subsurf scattering contributions.
  float4 oil;
  float4 sheen;
  float4 subsurf;
```

```
  float  Kr, Kr2;
  float  Kt, Kt2;
  float3 T, T2;
  float3 R, R2;

  // Compute fresnel at sheen layer, 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 oil contribution, modulate the oiliness mask by a
  // specular term.
  oil = 0.5 * oiliness * pow( ndoth, m );

  // For sheen contribution, modulate Fresnel term by
  // sheen color times specular.  Modulate by 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
  // simultaneously and the results end up in the x,y,z
  // components of a float3.  Using 3 terms approximates
  // 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 light color and result
  // of a shadow map lookup.
  return lightColor*tex2Dproj( tex3, In.shadowcoords ).r *
         (oil + sheen + subsurf);
}
```

薄膜エフェクト

説明

このデモでは、薄膜干渉エフェクトを示します。スペキュラ ライティングとディフューズ ライティングは、ビュー深度パラメータとともに Cg プログラムで頂点ごとに計算されます。ビュー深度パラメータは、ビュー ベクトル、表面法線およびオブジェクト表面の薄膜の深度を使用して計算されます。ビュー深度は、下部レイヤのデカール テクスチャによりフラグメントごとに適当に摂動されます。次に、ビュー深度を使用して、特定のビュー深度の場合の赤 / 緑 / 青の波長について事前計算された 1 次元テクスチャがルックアップされます。この干渉値は、標準ライティング方程式のスペキュラ ライティング コンポーネントの変調に使用されます。

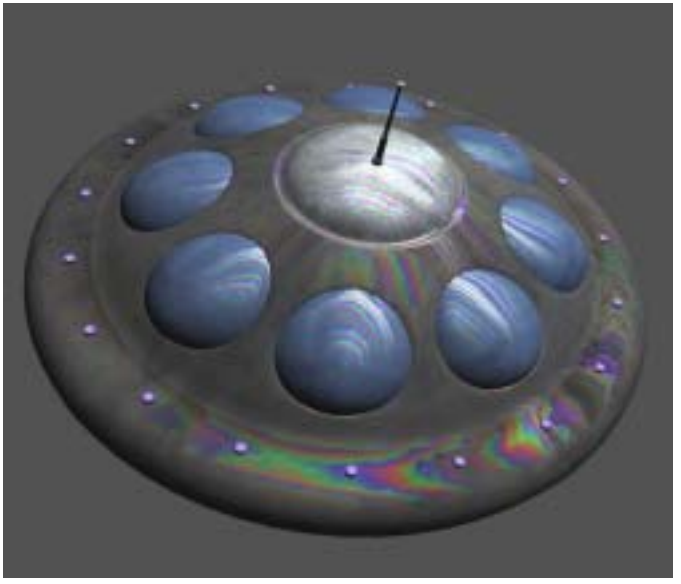


図 11 薄膜エフェクトの例

薄膜エフェクトの頂点シェーダ ソース コード

```
// define inputs from application
struct a2v
{
  float4 Position : POSITION;
  float3 Normal   : NORMAL;
```

```
};

// define outputs from vertex shader
struct v2f
{
  float4 HPOS      : POSITION;
  float4 diffCol   : COLOR0;
  float4 specCol   : COLOR1;
  float2 filmDepth : TEXCOORD0;
};

v2f main(a2v IN,
         uniform float4x4 WorldViewProj,
         uniform float4x4 WorldViewIT,
         uniform float4x4 WorldView,
         uniform float4 LightVector,
         uniform float4 FilmDepth,
         uniform float4 EyeVector)
{
  v2f OUT;

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

  float4 tempnorm = float4(IN.Normal, 0.0);

  // transform normal from model-space to view-space
  float3 normalVec = mul(WorldViewIT, tempnorm).xyz;
  normalVec = normalize(normalVec);

  // compute the eye->vertex vector
  float3 eyeVec = EyeVector.xyz;

  // compute the view depth for the thin film
  float viewdepth = (1.0 / dot(normalVec, eyeVec)) *
                    FilmDepth.x;

  OUT.filmDepth = viewdepth.xx;

  // store normalized light vector
  float3 lightVec = normalize((float3)LightVector);

  // calculate half angle vector
  float3 halfAngleVec = normalize(lightVec + eyeVec);
```

```
  // calculate diffuse component
  float diffuse = dot(normalVec, lightVec);

  // calculate specular component
  float specular = dot(normalVec, halfAngleVec);

  // use the lit instruction to calculate lighting,
  // automatically clamp
  float4 lighting = lit(diffuse, specular, 32);

  // output final lighting results
  OUT.diffCol = (float4)lighting.y;
  OUT.specCol = (float4)lighting.z;

  return OUT;
}
```

薄膜エフェクトのピクセル シェーダ ソース コード

```
struct v2f
{
  float3 diffCol   : COLOR0;
  float3 specCol   : COLOR1;
  float2 filmDepth : TEXCOORD0;
};

void main( v2f IN,
           out float4 color : COLOR,
           uniform sampler2D fringeMap,
           uniform sampler2D diffMap)
{
  // diffuse material color
  float3 diffCol  = float3(0.3, 0.3, 0.5);

  // lookup fringe value based on view depth
  float3 fringeCol = (float3)tex2D(fringeMap, IN.filmDepth);

  // modulate specular lighting by fringe color,
  // combine with regular lighting
  color.rgb = fringeCol*IN.specCol + IN.diffCol*diffCol;
  color.a = 1.0;
}
```

カー ペイント 9

説明

このカー ペイント シェーダは、Cornell 大学で測定された反射角測定ペイント サンプルを使用します。サンプル値は、**NdotL** および **NdotH** を **s**,**t** 座標のペアとして使用してインデクシングされ、ライティング方程式のディフューズ コンポーネントを提供する、2 次元テクスチャ マップに変換されています。スペキュラ項はブリン モデルを使用して計算され、クリア コートの金属粒をシミュレートする項も含んでいます。

粒の法線ミップマップ チェインには、接空間の正の Z 円錐内に位置するようにランダムに生成されたベクトルがあります。円錐は、遠くの方で粒が主に上を指すように各レベルで徐々に縮小されます。近づくにつれて外観が粗くなり、離れるにつれて外観が均一になるよう、粒のスペキュラ出力とそこからの寄与は距離によって小さくされます。次に、ビュー ベクトルは波状の法線マップ（オブジェクトの自然波形を表す）から反射して環境マップにインデクシングされます。クリア コート自体の光沢は、フレネル項を環境マップの輝度[1]でスケーリングすることにより計算されます。最後に、シェーダはディフューズ ペイント色と、フレネル項に基づく反射との間を線形補間し、スペキュラ ハイライトを追加します。

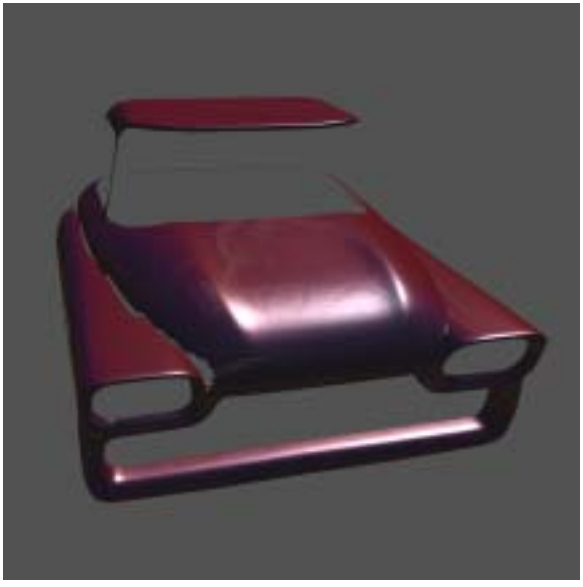


図 12 カー ペイント 9 の例

1. 輝度転移関数は、場面の暗い領域では反射しないように、環境マップの知覚的に明るい領域のみを選択します。

カー ペイント 9 の頂点シェーダ ソース コード

```
// This shader is based on the Time Machine temporal rust
// shader.  Car paint data was measured by Cornell
// University from samples provided by Ford Motor Company.

struct a2v {
  float4 OPosition : POSITION;
  float3 ONormal   : NORMAL;
  float2 uv        : TEXCOORD0;
  float3 Tangent   : TEXCOORD1;
  float3 Binormal  : TEXCOORD2;
  float3 Normal    : TEXCOORD3;
};

struct VS_OUTPUT {
  float4 HPosition : POSITION;  // coord position in window
  float2 uv        : TEXCOORD0; // wavy/fleckmap coords
  float3 light     : TEXCOORD1; // light pos (tangent space)
  float4 halfangle : TEXCOORD2; // Blinn halfangle
  float3 reflection: TEXCOORD3; // Refl vector (per-vertex)
  float4 view      : TEXCOORD4; // view (tangent space)
  float3 tangent   : TEXCOORD5; // view-tangent matrix
  float3 binormal  : TEXCOORD6; // ...
  float3 normal    : TEXCOORD7; // ...
  float  fresn     : COLOR0;
};

VS_OUTPUT main( a2v vert,
      // TRANSFORMATIONS
      uniform float4x4 ModelView,
      uniform float4x4 ModelViewIT,
      uniform float4x4 ModelViewProj,
      uniform float3   LightVector,    // Obj space
      uniform float3   EyePosition )   // Obj space
{
  VS_OUTPUT O;

  // Generate homogeneous POSITION
  O.HPosition = mul(ModelViewProj, vert.OPosition);

  // Generate BASIS matrix
  float3x3 ModelTangent = { normalize(vert.Tangent),
                            normalize(vert.Binormal),
                            normalize(vert.Normal) };
```

```
// FRESNEL           = { OFFSET, SCALE, POWER, UNUSED };
float4 Fresnel       = { 0.1f, 4.2f, 4.4f, 0.0f };

float3x3 ViewTangent = mul(ModelTangent,
                           (float3x3)ModelViewIT);

// Generate VIEW SPACE vectors
float3 viewN = normalize(mul((float3x3)ModelView,
                             vert.ONormal));
float4 viewP = mul(ModelView, vert.OPosition);
viewP.w = 1-saturate(sqrt(dot(viewP.xyz,
                              viewP.xyz))*0.01);
float3 viewV = -viewP.xyz;

// Generate OBJECT SPACE vectors
float3 objV = normalize(EyePosition-vert.OPosition.xyz);
float3 objL = normalize(LightVector);
float3 objH = normalize(objL + objV);


// Generate TANGENT SPACE vectors
float3 tanL = mul(ModelTangent, objL);
float3 tanV = mul(ModelTangent, objV);
float3 tanH = mul(ModelTangent, objH);

// Generate REFLECTION vector for per-vertex
// reflection look-up
float3 reflection = reflect(-viewV, viewN);

// Generate FRESNEL term
float ndv  = saturate(dot(viewN, viewV));
float FresnelApprox = (pow((1-ndv),Fresnel.z)*Fresnel.y +
                       Fresnel.x);

// Fill OUTPUT parameters
O.uv.xy      = vert.uv;        // TEXCOORD0.xy
O.light      = tanL;           // Tangent space LIGHT
// Tangent space HALF-ANGLE
O.halfangle  = float4(tanH.x, tanH.y,
                      tanH.z, 1-exp(-viewP.w));
O.reflection = reflection;     // View space REFLECTION
// Tangent space VIEW + distance attenuation
O.view       = float4(tanV.x, tanV.y,
                      tanV.z, viewP.w);
```

```
  // VIEWTANGENT
  O.tangent    = normalize(ViewTangent[0]); // column 0
  O.binormal   = normalize(ViewTangent[1]); // column 1
  O.normal     = normalize(ViewTangent[2]); // column 2
  O.fresn      = FresnelApprox;

  return O;
}
```

カー ペイント 9 のピクセル シェーダ ソース コード

```
// This shader is based on the Time Machine temporal rust
// shader.  Car paint data was measured by Cornell
// University from samples provided by Ford Motor Company.
//

struct VS_OUTPUT {
  float4 HPosition : POSITION;  // coord position in window
  float2 uv        : TEXCOORD0; // wavy/fleckmap coords
  float3 light     : TEXCOORD1; // light pos (tangent space)
  float4 halfangle : TEXCOORD2; // Blinn halfangle
  float3 reflection: TEXCOORD3; // Refl vector (per-vertex)
  float4 view      : TEXCOORD4; // view (tangent space)
  float3 tangent   : TEXCOORD5; // view-tangent matrix
  float3 binormal  : TEXCOORD6; // ...
  float3 normal    : TEXCOORD7; // ...
  float  fresn     : COLOR0;
};

// PIXEL SHADER
float4 main( VS_OUTPUT vert,
       uniform sampler2D WavyMap          : register(s0),
       uniform samplerCUBE EnvironmentMap : register(s1),
       uniform sampler2D PaintMap         : register(s2),
       uniform sampler2D FleckMap         : register(s3),
       uniform float Ambient ) : COLOR
{
  // NEWPAINTSPEC      = { UNUSED, SPEC POWER, GLOSSINESS,
  //                       FLECK SPEC POWER }
  float4 NewPaintSpec  = { 0.0f, 64.0f, 3.8f, 8.0f };
  float3 ClearCoat     = { 0.299f,0.587f, 0.114f };
  float3 FleckColor    = { 0.9, 1.05, 1.0 };
  float3 WavyScale     = { 0.2, -0.2, 1.0 };
```

```
// Tangent space LIGHT vector
float3 L = normalize(vert.light);

// Tangent space HALF-ANGLE vector
float3 H = normalize(vert.halfangle.xyz);

// Tangent space VIEW vector
float3 V = normalize(vert.view.xyz);
float v_dist = vert.view.w;

// Tangent space WAVY_NORMAL
float3 wavyN = (float3)tex2D(WavyMap, vert.uv)*2-1;
wavyN = normalize(wavyN*WavyScale);


// PAINT
// A normal map map could be loaded here instead if
// we wanted more detail. In this case we have a
// uniform tangent space normal (0,0,1)
float n_d_l = L.z;
float n_d_h = H.z;
float3 paint_color = (float3)tex2D(PaintMap,
                                   float2(n_d_l, n_d_h));

// SPECULAR POWER - use a saturated diffuse term
// to clamp the backlighting
n_d_h = saturate(n_d_l*4)*pow(n_d_h, NewPaintSpec.y);

// REFLECTION ENVIRONMENT
// Reflect view vector about wavy normal and bring
// to view space
float3 R = reflect(-V, wavyN);
R = R.x*vert.tangent + R.y*vert.binormal +
    R.z*vert.normal;
float3 reflect_color = (float3)texCUBE(EnvironmentMap, R);

// FLECKS
// Load random 3-vector flecks from fleck_map
// Reduce tiling artifacts by sampling at
// different frequencies
float3 fleckN = (float3)tex2D(FleckMap, vert.uv*37)*2-1;
fleckN = ((float3)tex2D(FleckMap, vert.uv*23)*2-1)/2 +
         fleckN/2;
float  fleck_n_d_h = saturate(dot(fleckN, H));
```

```
  float3 fleck_color = FleckColor * pow(fleck_n_d_h,
            lerp(NewPaintSpec.y, NewPaintSpec.w, v_dist));
  // Control the ambient fleckiness and also
  // attenuate with distance
  fleck_color = fleck_color*Ambient*vert.halfangle.w;

  // DIFFUSE
  float k_d = saturate(n_d_l*1.2);
  float3 paintResult = lerp(Ambient*paint_color,
                            paint_color, k_d);

  // FRESNEL
  float Fresnel = saturate(dot(ClearCoat, reflect_color));
  Fresnel = pow(Fresnel, NewPaintSpec.z);

  // This helps make the clear coat less omnipresent --
  // only the really (perceptually) bright areas reflect
  // the most.
  Fresnel = saturate(vert.fresn*Fresnel);
  // Show more of the specular reflection environment
  // when in fresnel zones
  // diffuse * (1-fresnel) + environment * (fresnel)
  paintResult = lerp(paintResult, reflect_color, Fresnel);

  // SPECULAR
  // diffuse + specular + flecks
  paintResult = paintResult + n_d_h + fleck_color;

  // OUTPUT
  return paintResult.xyzz;
}
```

基本プロファイルのサンプル シェーダ

この章では、Cg で記述された基本プロファイルのサンプル シェーダを紹介します。各シェーダにはスナップショット、説明およびソース コードが付いています。

サンプルは次のとおりです。

- 異方性ライティング
- バンプ dot3x2 ディフューズおよびスペキュラ
- バンプ反射マッピング
- フレネル
- 草
- 屈折
- シャドウ マッピング
- シャドウ ボリューム掃引
- サイン波デモ
- 行列パレット スキニング

異方性ライティング

説明

異方性ライティングのエフェクト（図 13）では、頂点プログラムのハーフアングル ベクトル計算を示します。頂点ごとの **HdotN** および **LdotN** を使用して 2D テクスチャをルックアップし、おもしろいライティング エフェクトを実現します。

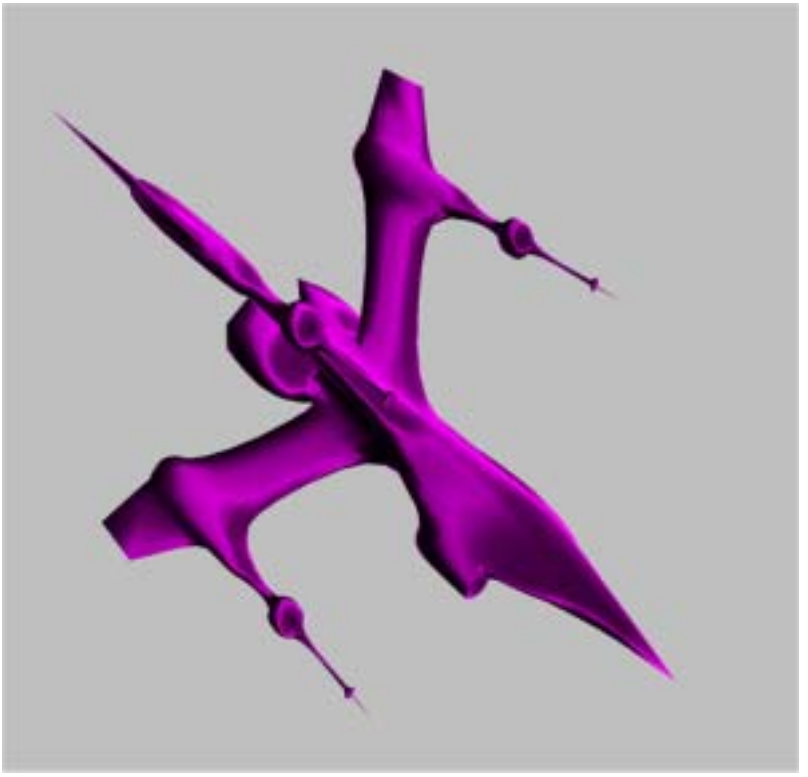


図 13 異方性ライティングの例

異方性ライティングの頂点シェーダ ソース コード

```
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;
}
```

バンプ dot3x2 ディフューズおよびスペキュラ

説明

バンプ dot3x2 ディフューズおよびスペキュラのエフェクトでは、DirectX 8 のピクセル シェーダ命令である **texm3x2tex**（OpenGL の **DOT_PRODUCT_TEXTURE_2D** に相当）に基づいてバンプ マッピングをディフューズおよびスペキュラ ライティングに使用します。この命令（**texm3x2tex**）は、ディフューズ ライティング コンポーネントに対応する法線およびライト ベクトルの内積を計算し、スペキュラ ライティング コンポーネントに対応する法線およびハーフアングル ベクトルの内積を計算します。この計算により、2 つのスカラ値が算出されます。このスカラ値は、アルファ コンポーネント内にディフューズ色とスペキュラ項を含む 2D イルミネーション テクスチャをルックアップする際に、テクスチャ座標として使用されます。法線マップから取り出した法線は接空間にあるため、ライト ベクトルとハーフアングル ベクトルの両方が、頂点シェーダによってこの空間に変換されています（図 14）。

図 14 バンプ dot3x2 ディフューズおよびスペキュラの例

バンプ dot3x2 の頂点シェーダ ソース コード

```
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;

  // 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;

  // 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;
}
```

バンプ dot3x2 のピクセル シェーダ ソース コード

```
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
};
```

```
float4 main(v2f IN,
      uniform sampler2D DiffuseMap,
      uniform sampler2D NormalMap,
      uniform sampler2D IlluminationMap,
      uniform float Ambient) : COLOR
{
  // fetch base color
  float4 color = tex2D(DiffuseMap, IN.TexCoord0.xy);

  // fetch bump normal and expand it to [-1,1]
  float4 bumpNormal = 2 *
    (tex2D(NormalMap, IN.TexCoord1.xy) - 0.5);

  // 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

  float2 illumCoord =
    float2(dot(IN.LightVector.xyz, bumpNormal.xyz),
           dot(IN.HalfAngleVector.xyz, bumpNormal.xyz));
  float4 illumination = tex2D(IlluminationMap, illumCoord);

  // expand iterated normal to [-1,1]
  float4 normal = 2 * (IN.Normal - 0.5);

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

  // compute final color
  return (Ambient * color + shadow)
      * (illumination * color + illumination.wwww);
}
```

バンプ反射マッピング

説明

このエフェクトでは、DirectX 8 のピクセル シェーダ命令である **texm3x3vspec**（OpenGL の **DOT_PRODUCT_REFLECT_CUBE_MAP** に相当）に基づいてバンプ マッピングと反射マッピングを混合します。この命令では、3 つの内積を計算して法線マップから取り出した法線をキューブ環境マップ空間に変換し、視点ベクトルを基準としてその変換後の法線を反射し、さらにキューブ マップを取り出して最終的な色を取得します。頂点シェーダは、変換行列と視点ベクトルを計算しなくてはなりません（図 15）。

図 15 バンプ反射マッピングの例

バンプ反射マッピングの頂点シェーダ ソース コード

```
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 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;
}
```

バンプ反射マッピングのピクセル シェーダ ソース コード

```
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;
};

float4 main(v2f IN,
      uniform sampler2D NormalMap,
      uniform samplerCUBE EnvironmentMap,
      uniform float3 EyeVector) : COLOR
{
  // fetch the bump normal from the normal map
  float4 normal = tex2D(NormalMap, IN.TexCoord.xy);

  // 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
  return texCUBE_reflect_eye_dp3x3(EnvironmentMap,
                  IN.TangentToCubeSpace2.xyz,
                  IN.TangentToCubeSpace0,
                  IN.TangentToCubeSpace1,
                  normal,
                  EyeVector);
}
```

フレネル

説明

このエフェクトでは、反射の環境マップをルックアップするための反射ベクトルを計算し、これをフレネル項で変調します。結果として、浅い角度でのみ反射が効いてきます（図 16）。

図 16 フレネルの例

フレネルの頂点シェーダ ソース コード

```
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,
               uniform float4x4 ModelView,
               uniform float4x4 ModelViewIT)
{
  vert2frag OUT;

#ifdef PROFILE_ARBVP1
  ModelViewProj = glstate.matrix.mvp;
  ModelView = glstate.matrix.modelview[0];
  ModelViewIT = glstate.matrix.invtrans.modelview[0];
#endif

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

  float3 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 = float4(reflect(eyeToVert, normal), 1.0);

  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;
}
```

草

説明

このエフェクトは、Sin 関数によるジオメトリのプロシージャル アニメーションと、変形されたジオメトリの法線ベクトルの計算を実装しています（図 17）。

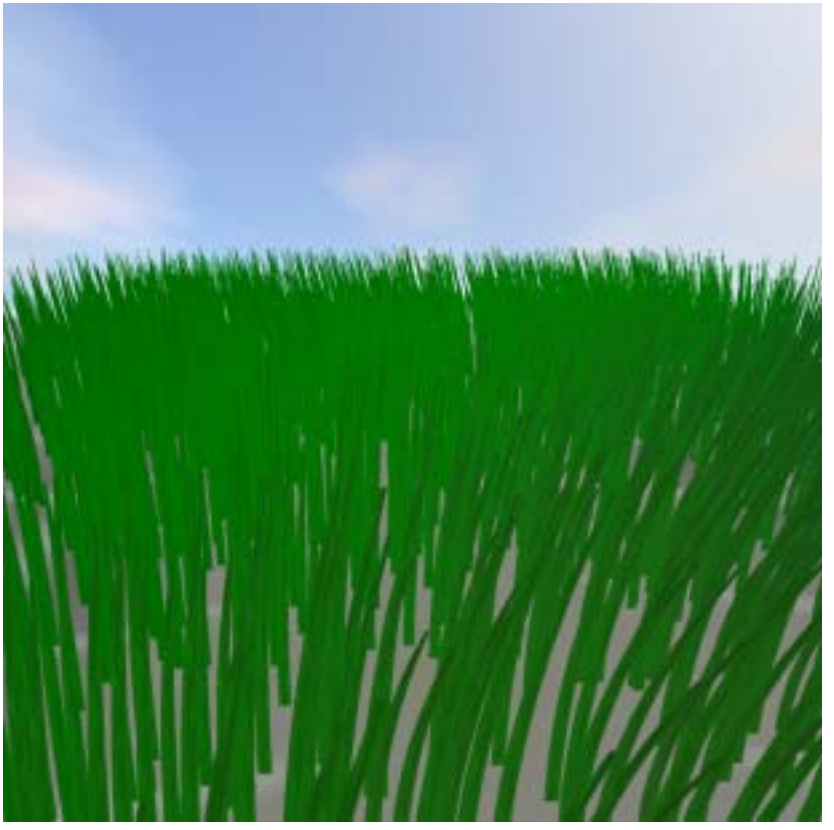


図 17 草の例

草の頂点シェーダ ソース コード

```
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
  float normal = cross((result - newResult).xyz,
                       float3(1,0,0));
  normal = normalize(normal);

  // calculate diffuse lighting off the normal
  //   that was just calculated
  float3 lightPos = float3(0,5,15);
  float3 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;
}
```

屈折

説明

このエフェクトでは、カスタム テクスチャ座標生成を実行して、頂点ごとの屈折ベクトルを計算し、次にそれを使用してキューブマップをルックアップします。反射と屈折をブレンドするためにフレネル係数も計算されます（図 18）。

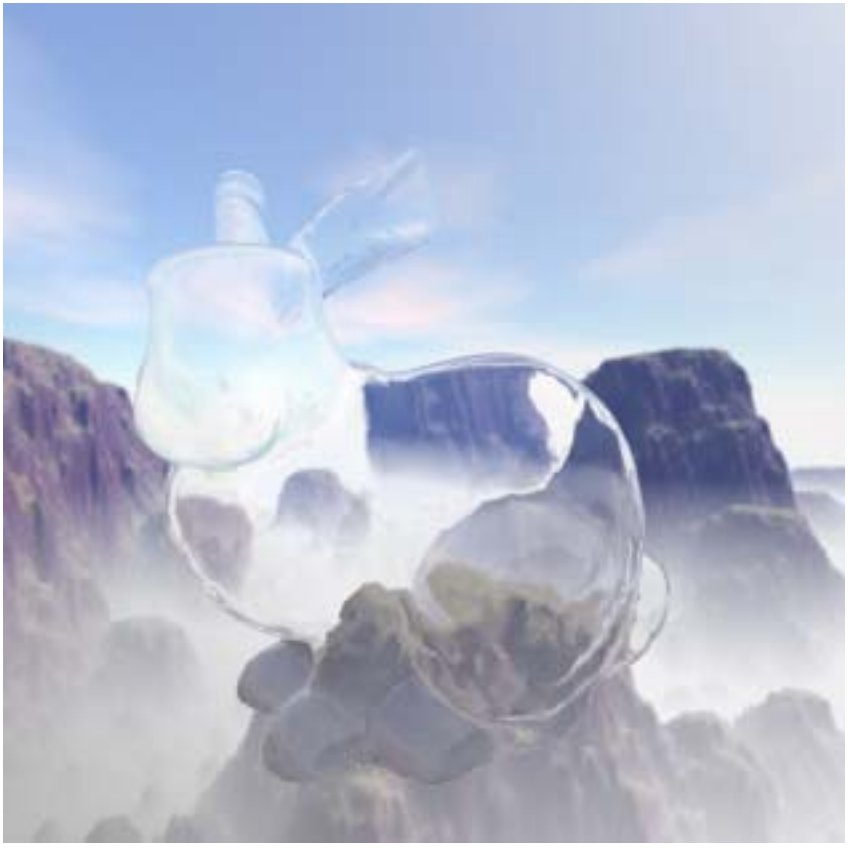


図 18 屈折の例

屈折の頂点シェーダ ソース コード

```
struct inputs
{
  float4 Position     : POSITION;
  float4 Normal       : NORMAL;
};

struct outputs
{
  float4 hPosition    : POSITION;
  float4 fresnelTerm  : COLOR0;
  float4 refractVec   : TEXCOORD0;
  float4 reflectVec   : TEXCOORD1;
};

// fresnel approximation
fixed fast_fresnel(float3 I, float3 N,
                   float3 fresnelValues)
{
  fixed power = fresnelValues.x;
  fixed scale = fresnelValues.y;
  fixed bias = fresnelValues.z;

  return bias + pow(1.0 - dot(I, N), power) * scale;
}

outputs main(inputs IN,
           uniform float4x4 ModelViewProj,
           uniform float4x4 ModelView,
           uniform float4x4 ModelViewIT,
           uniform float theta)
{
  outputs 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);
```

```
  OUT.refractVec.xyz = refract(eyeToVert, normal, theta);
  OUT.refractVec.w = 1;

  OUT.reflectVec.xyz = reflect(eyeToVert, normal);
  OUT.reflectVec.w = 1;

  // calculate the fresnel reflection
  OUT.fresnelTerm = fast_fresnel(-eyeToVert, normal,
                                 float3(5.0, 1.0, 0.0));
  return OUT;
}
```

屈折のピクセル シェーダ ソース コード

```
float4 main(in float3 refractVec    : TEXCOORD0,
            in float3 reflectVec    : TEXCOORD1,
            in float3 fresnelTerm   : COLOR0,

            uniform samplerCUBE environmentMaps[2],
            uniform float enableRefraction,
            uniform float enableFresnel) : COLOR
{
  float3 refractColor = texCUBE(environmentMaps[0],
                                refractVec).rgb;
  float3 reflectColor = texCUBE(environmentMaps[1],
                                reflectVec).rgb;

  float3 reflectRefract = lerp(refractColor, reflectColor,
                               fresnelTerm);

  float3 finalColor = enableRefraction ?
      (enableFresnel ? reflectRefract : refractColor) :
      (enableFresnel ? reflectColor : fresnelTerm);

  return float4(finalColor, 1.0);
}
```

シャドウ マッピング

説明

このエフェクトでは、ピクセルごとのライティング方程式でシャドウ マップを使用しつつ、シャドウ マッピングのテクスチャ座標を生成します（図 19）。

図 19 シャドウ マッピングの例

シャドウ マッピングの頂点シェーダ ソース コード

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

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

vpconn 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;
}
```

シャドウ マッピングのピクセル シェーダ ソース コード

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

float4 main(v2f_simple IN,
      uniform sampler2D ShadowMap,
      uniform sampler2D SpotLight) : COLOR
{
    float4 shadow    = tex2D(ShadowMap, IN.TexCoord0.xy);
    float4 spotlight = tex2D(SpotLight, IN.TexCoord1.xy);
    float4 lighting  = IN.Color0;

    return shadow * spotlight * lighting;
}
```

シャドウ ボリューム掃引

説明

このエフェクトでは頂点プログラムを使用し、ライト ベクトルに沿ってジオメトリを掃引することにより、シャドウ ボリュームを生成します（図 20）。

図 20 シャドウ ボリューム掃引の例

シャドウ ボリューム掃引の頂点シェーダ ソース コード

```
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,
           uniform float4 LightPos, // (in object space)
           uniform float4 Fatness,
           uniform float4 ShadowExtrudeDist,
           uniform float4 Factors
   )
{
  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
  float ndotl = dot(-light_to_vert.xyz, IN.Normal.xyz);

  // 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 ndotl < 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.
  float away = (float)(ndotl < 0);

  // Move the back-facing shadow volume points
  float4 new_position = extrusion_vec * away + 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 = float4(0, 0, Factors.x, 0);

  OUT.Color0 = color;
  OUT.TexCoord0.xy = IN.TexCoord0;
  return OUT;
}
```

サイン波デモ

説明

このエフェクトでは、現在の時刻に基づき、Sin 関数を使用して頂点座標を変更します。組込みの **sin()** 関数の使用例です。また、摂動されたメッシュの法線を計算し、この法線を使用して、キューブ マップをルックアップするための反射ベクトルを計算します（図 21）。

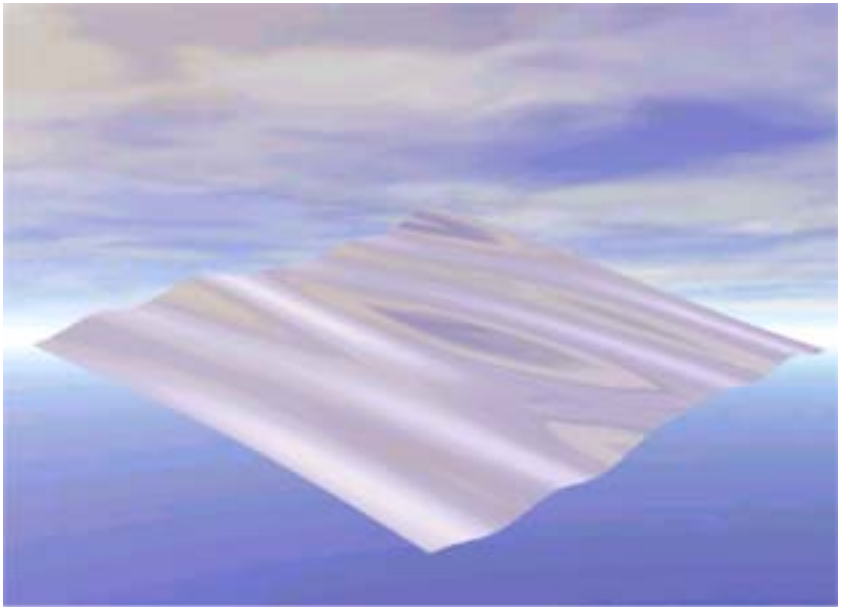


図 21 サイン波の例

サイン波の頂点シェーダ ソース コード

```
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 float3   WavesX,
          uniform float3   WavesY,
          uniform float3   WavesH,
          uniform float3   Time
   )
{
  vpconn OUT;

  float3 angle = WavesX * IN.TexCoord0.x +
          WavesY * IN.TexCoord0.y;
  angle = angle + Time;

  float3 sine, cosine;
  sincos(angle, sine, cosine);

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

  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;
}
```

行列パレット スキニング

説明

このエフェクトでは、頂点ごとに 2 つのボーンを使用し、行列パレット スキニングを実行します。メッシュに対するすべてのボーンは定数メモリに設定され、各頂点には、この頂点に影響するボーンを示す 2 つのインデックスが含まれます。最終的にスキニングされる座標は、頂点ごとに提供されている重みとともに、これらのボーンを使用して計算されます。接空間基底が同様の方式でスキニングされ、接空間でのピクセルごとでのバンプマッピングでライトベクトルを変換するために使用されます（図 22）。

図 22 行列パレット スキニングの例

行列パレット スキニングの頂点シェーダ ソース コード

```
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._m00_m01_m02 = Bones[i]._m00_m01_m02;
  m._m10_m11_m12 = Bones[i]._m10_m11_m12;
  m._m20_m21_m22 = Bones[i]._m20_m21_m22;

  // 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._m00_m01_m02 = Bones[i]._m00_m01_m02;
m._m10_m11_m12 = Bones[i]._m10_m11_m12;
m._m20_m21_m22 = Bones[i]._m20_m21_m22;

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._m00_m01_m02 = finalS;
worldToTangentSpace._m10_m11_m12 = finalT;
worldToTangentSpace._m20_m21_m22 = 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;
}
```

付録 A
Cg 言語仕様

言語の概要

Cg 言語は、基本的に ANSI C をモデルとしていますが、C++ および Java などの新しい言語や、RenderMan および Stanford シェーディング言語など従来のシェーディング言語からも概念を一部採用しています。この言語には、新しい概念もいくつか導入されています。特に、GPU などのストリーム処理アーキテクチャでデータ フローを表すように設計された機能も含まれています。コンパイル時に指定される各プロファイルでは、ループを実装する機能や特定の計算を実行する精度など、言語の特定の機能をサブセット化できます。

明示されていない非互換性

ANSI C からの変更の大部分は削除または追加ですが、説明されない可能性のある非互換性がいくつかあります。Cg では次の点が変更されており、プログラムは正常にコンパイルされますが、C とは異なる動作をします。

- 定数が型キャストまたは型接尾辞を使用して明示的に型指定されていない場合は、定数の型の昇格規則が異なります。一般に、明示的に型指定されていない定数および変数の間での 2 項演算は、定数のデフォルトの精度ではなく変数の精度で実行されます。
- **`struct`** の宣言は、(C++ の場合と同様に) 自動の **`typedef`** を実行するため、以前に宣言された型をオーバーライドすることがあります。
- 配列は、ポインタとは異なる基本型タイプです。そのため、配列割当ては、意味的には配列全体のコピー操作を実行します。

異なる表現が必要な類似の演算

いくつかの変更のため、Cg では、同じ演算を C と異なる方法で表現しなければならない場合があります。

- ブール型（**`bool`**）が導入されています。演算子と制御コンストラクトに関連があります。
- Cg ではポインタがサポートされていないため、配列は基本型タイプです。

- 関数は値/結果で値を渡すため、仮パラメータ リストに**out**または**inout**修飾子を使用してパラメータを返します。デフォルトでは、仮パラメータは **in** ですが、これを明示的に指定することも可能です。パラメータは **in out** として指定することもでき、これは意味的に **inout** と同等です。

ANSI C との相違

Cg は、ANSI-C 言語に基づいて開発され、主として次のような追加、削除および変更が行われています。（これは概要です。詳細は、このマニュアルで後述します。）

- 言語の各機能は、言語プロファイル（168 ページの「プロファイル」で説明）ごとにサブセット化されています。特に、**for** および **while** によるループは、言語プロファイルによって制限される場合があります。たとえば、一部のプロファイルは、コンパイル時に完全にアンロールできるループのみサポートしている場合があります。
- バインディング セマンティクスは、特定のハードウェアまたは API リソースへのオブジェクトのマッピングを示すために、構造体タグ、変数または構造体要素に関連付けられることがあります。183 ページの「バインディング セマンティクス」を参照してください。
- 予約キーワード **goto**、**break** および **continue** はサポートされていません。
- 予約キーワード **switch**、**case** および **default** はサポートされていません。ラベルもサポートされていません。
- ポインタまたはポインタ関連機能（**&** 演算子や **->** 演算子など）はサポートされていません。
- 配列はサポートされていますが、サイズと次元に制限事項があります。計算された添字の使用を制限することも許可されています。配列は、**packed** として指定できます。パックされた配列に対して許可される演算は、アンパックされた配列に対して許可される演算とは異なります。ベクトルと行列には定義済みの **packed** 型が用意されており、これらの定義済みの型を使用することを強くお薦めします。
- ベクトル用に組込みのスウィズル演算子 **.xyzw** または **.rgba** があります。この演算子により、ベクトルの成分の再配置および複製が可能です。スカラからベクトルを作成することもできます。
- 左辺値の場合は、スウィズル演算子によってベクトルまたは行列の要素を選択的に記述できます。
- 行例用には、次のような類似の組込みのスウィズル演算子があります。

  ._m<*row*><*col*>[_m<*row*><*col*>][...]

  この演算子により、個々の行列要素にアクセスでき、行列の要素からベクトルを作成できます。DirectX 8 表記規則との互換性のため、行列スウィズルには第 2 の形式があります。これについては後述します。

- 数値データ型は、C と異なります。Cg の主な数値データ型は、**float**、**half** および **fixed** です。フラグメント プロファイルでは、これら 3 つのデータ型をすべてサポートする必要がありますが、**float** の精度で **half** および **fixed** を実装することもできます。頂点プロファイルでは、**half** および **float** をサポートする必要がありますが、その多くは **float** の精度で **half** を実装しています。頂点プロファイルでは、**fixed** 演算のサポートを省略できますが、その場合でも **fixed** 変数の定義はサポートする必要があります。Cg の各プロファイルでは、ランタイムに **int** のサポートを省略できます。Cg の各プロファイルでは、**double** を **float** として扱うことができます。
- 多くの演算子では、成分ごとのベクトル演算がサポートされています。
- **?:**、**||**、**&&**、**!** および比較演算子を、**bool** 4 成分ベクトルとともに使用して、4 つの条件付き演算を同時に実行できます。**?:**、**||** および **&&** の各演算子に対するすべてのオペランドの副作用は、常に実行されます。
- 非静的グローバル変数、および最上位レベル関数（**main()** など）へのパラメータは、**uniform** として指定できます。**uniform** 変数は、他の変数と同じようにプログラム内で読込みおよび書込みできます。一方、**uniform** 修飾子は、プログラムが何度も呼び出されても、変数またはパラメータの初期値が一定である必要があることを示します。
- 新しい **sampler*** 型のセットは、テクスチャ オブジェクトへのハンドルを表します。
- 関数では、C++ と同様、パラメータがデフォルト値を持つ場合があります。デフォルト値は、代入構文を使用して表現されます。
- 関数のオーバーロードがサポートされています。
- **enum** または **union** はありません。
- 構造体のビットフィールド宣言は許可されていません。
- 構造体のビットフィールド宣言はありません。
- 変数は、C のようにスコープの先頭だけではなく、使用場所より前であれば任意の場所で定義できます（つまり、変数の宣言場所を管理する C++ 規則を採用しています）。変数を、同じスコープ内で再宣言することはできません。
- ベクトル コンストラクタ（たとえば、**float4(1,2,3,4)** の形式）は、式の任意の場所で使用できます。
- **struct** 定義は、C++ の場合と同様、対応する **typedef** を自動的に実行します。
- C スタイルのコメント **/* ... */** に加えて、C++ スタイルのコメント **//** も使用できます。

詳細な言語仕様

定義

次の定義は、ANSI C 規格に基づいています。

- オブジェクト
  オブジェクトは、実行環境での一定範囲のデータ記憶域であり、その内容は値を表すことができます。オブジェクトが参照される場合、オブジェクトは特定の型を持つものと解釈されます。
- 宣言
  宣言は、一連の識別子の解釈と属性を指定します。
- 定義
  識別子で指定された関数に対して生成されるオブジェクトまたはコードのために記憶域を確保するような宣言は、定義です。

プロファイル

Cg プログラム（最上位レベル関数）のコンパイルは、常にコンパイル プロファイルのコンテキストに従って行われます。各プロファイルは、特定のオプション言語機能がサポートされるかどうかを指定します。これらのオプション言語機能には、特定の制御コンストラクトおよび標準ライブラリ関数が含まれます。コンパイル プロファイルによって、**`float`**、**`half`** および **`fixed`** データ型の精度も定義され、**`fixed`** および **`sampler*`** データ型のサポートが完全か部分的かも指定されます。コンパイル プロファイルの選択は、言語の外部で（たとえば、コンパイラのコマンドライン スイッチを使用して）行います。

プロファイルによる制限が適用されるのは、コンパイルされる最上位レベル関数と、それが直接あるいは間接的に参照している変数または関数のみです。ソースコード内にあっても、最上位レベル関数により直接または間接的にコールされていない関数では、現在のプロファイルでサポートされていない機能も任意に使用できます。

このような規則があるために、Cg では 1 つのソース ファイルに、異なるプロファイルを対象とした複数の最上位レベル関数を記述することが可能です。Cg 言語のコア仕様は、これらの関数すべてを受け入れることができます。コンパイル プロファイルによる制限が必要なのは、コード生成のためのみです。したがってこの制限は、コード生成の対象となる関数だけに適用されます。このマニュアルで**プログラム**と言う語を使用する場合は、最上位レベル関数、最上位レベル関数がコールする任意の関数、および最上位レベル関数が参照する任意のグローバル変数または **`typedef`** 定義を指しています。

各プロファイルには、特性と制限事項を記述する別個の仕様が必要です。

このコア仕様により、すべてのプロファイルには一定の最小機能が要求されます。それぞれ最小機能が異なる頂点プログラムとフラグメント プログラムが、コア仕様によって区別される場合もあります。

uniform 修飾子

非静的グローバル変数と、**main()** などの関数に渡されるパラメータは、オプションの修飾子 **uniform** を付けて宣言できます。**uniform** 変数を指定するには、次の構文を使用します。

uniform <*type*> <*variable*>

次に例を示します。

```
uniform float4 myVector;
```

または

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

最上位レベルではない関数に対して **uniform** 修飾子を指定しても、その修飾子は無意味であり無視されます。この規則の意図は、1 つの関数が最上位レベル関数としても、あるいは最上位レベルではない関数としても機能できるようにすることです。

uniform 型の変数は、非 **uniform** 型の変数と同じように読込みおよび書込みできることに注意してください。**uniform** 修飾子は、変数の初期値の指定および格納方法に関する情報を、言語外部のメカニズムを使用して提供するだけです。

通常、**uniform** 型の変数またはパラメータの初期値は、ハードウェア レジスタの別のクラスに格納されます。さらに、**uniform** 型の変数またはパラメータの初期値を指定する外部メカニズムは、非 **uniform** 型の変数またはパラメータの初期値の指定に使用されるメカニズムとは異なる場合があります。**uniform** 修飾子の付いたパラメータは、通常は不変なものとして扱われますが、非 **uniform** 型のパラメータは、各ストリーム レコードに新しい値が指定された（頂点配列でのように）ストリーミング データとして扱われます。

関数の宣言

関数は、基本的には C と同じように宣言されます。値を返さない関数は、**void** 戻り型で宣言する必要があります。パラメータをとらない関数は、次の 2 通りの方法のいずれかで宣言できます。

- C と同様に、**void** キーワードを使用 **functionName(void)**
- パラメータをまったく使用せずに **functionName()**

関数は、**static** として宣言できます。その場合、関数はプログラムとしてコンパイルされず、他のコンパイル ユニットから参照できなくなります。

プロファイルによる関数のオーバーロード

Cg では、コンパイル プロファイルによる関数のオーバーロードをサポートします。この機能により、関数をプロファイルごとに異なる方法で実装できます。サポートされる言語機能サブセットはプロファイルごとに異なる場合があり、また最も効率的な関数の実装もプロファイルごとに異なる場合があるため、この機能が役立ちます。

関数の定義では、プロファイル名の直後に型の名前を指定します。たとえば、**profileA** と **profileB** のプロファイルに対して関数 **myfunc()** の 2 つの異なるバージョンを定義するには、次のようにします。

```
profileA float myfunc(float x) {/*...*/};
profileB float myfunc(float x) {/*...*/};
```

プロファイルと同じ名前の型を定義（**typedef** を使用）した場合、識別子は型の名前として扱われるため、ファイルのこれ以降にプロファイルをオーバーロードすることはできません。

関数定義にプロファイルが含まれない場合、その関数は、オープンプロファイル関数と呼ばれます。オープンプロファイル関数は、すべてのプロファイルに適用されます。

プロファイル名には、いくつかのワイルドカードが定義されています。**vs** という名前は、任意の頂点プロファイルに一致し、**ps** という名前は任意のフラグメントまたはピクセル プロファイルに一致します。

ps_1 という名前は、DirectX 8 の任意のピクセル シェーダ 1.*x* プロファイルに、**ps_2** という名前は、DirectX 9 のピクセル シェーダ 2.*x* プロファイルにそれぞれ一致します。同様に、**vs_1** という名前は、DirectX の任意の頂点シェーダ 1.*x* に、**vs_2** という名前は、2.*x* にそれぞれ一致します。これ以外にも、個々のプロファイルで有効なワイルドカード プロファイル名が定義されている場合があります。

一般的には、最も明確なバージョンの関数が使用されます。詳細は 181 ページの「関数のオーバーロード」で説明しますが、検索順序の概略は次のようになります。

1. プロファイルにより厳密にオーバーロードされたバージョンの関数
2. 最も詳細なワイルドカード プロファイルによりオーバーロードされたバージョンの関数（**vs** や **ps_1** など）
3. プロファイルによりオーバーロードされていないバージョンの関数

このような検索プロセスがあるため、汎用バージョンの関数を定義しておき、特定のハードウェアでの必要に応じてそれをオーバーライドすることが可能です。

関数の宣言におけるパラメータの構文

関数は、C と同じような方法で宣言されますが、関数の宣言におけるパラメータには、バインディング セマンティクス（183 ページの「バインディング セマンティクス」を参照）とデフォルト値を指定できます。

関数定義の各パラメータは、次のような形をとります。

[uniform] <*type*> *identifier* [: <*binding_semantic*>] [= <*default*>]

ここで

- **<*type*>** には修飾子 **in**、**out**、**inout** および **const** を含めることができます。詳細は 175 ページの「型修飾子」を参照してください。
- **<*default*>** は、コンパイル時に定数として使用される式です。

デフォルト値を使用できるのは、**uniform** パラメータと、最上位レベルではない関数に対する **in** パラメータのみです。

関数コール

関数コールでは、右辺値が返されます。したがって、関数が配列を返す場合、配列を読み取ることはできますが、書き込むことはできません。たとえば、次の式は許可されます。

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

一方、次の式は許可されません。**myfunc(x)[2] = y;**.

1 つの式で複数の関数をコールする場合、コールは任意の順序で発生します。順序を定義することはできません。

型

Cg には、次のデータ型があります。

- **int** 型は、原則的には 32 ビットの 2 の補数です。各プロファイルでは、**int** を **float** として扱うこともできます。
- **float** 型は、IEEE の単精度（32 ビット）浮動小数点数に可能なかぎり近似します。各プロファイルでは、**float** データ型をサポートする必要があります。
- **half** 型は、IEEE に似た低精度の浮動小数点数です。各プロファイルでは、**half** 型をサポートする必要がありますが、**float** 型と同じ精度で実装することもできます。
- **fixed** 型は、少なくとも [-2,2) の範囲を持つ符号付きの型で、少なくとも 10 ビットの精度の小数部を持ちます。このデータ型に対するオーバーフロー処理は、ラップではなくクランプされます。フラグメント プロファイルでは、**fixed** 型をサポートする必要がありますが、**half** または **float** 型と同じ精度で実装することもできます。頂点プロファイルでは、**fixed** 型を部分的にサポート（174 ページの「型の部分的なサポート」を参照）する必要があります。頂点プロファイルには、**fixed** 型を完全にサポートするオプション

と、**half** または **float** 型と同じ精度で **fixed** 型を実装するオプションがあります。

- **bool** 型は、ブール値を表します。**bool** 型のオブジェクトは、true と false のいずれかになります。
- **cint** 型は、32 ビットの 2 の補数です。この型は、コンパイル時にのみ意味があります。**cint** 型のオブジェクトを宣言することはできません。
- **cfloat** 型は、IEEE の単精度（32 ビット）浮動小数点数です。この型は、コンパイル時にのみ意味があります。**cfloat** 型のオブジェクトを宣言することはできません。
- **void** 型は、式では使用できません。値を返さない関数の戻り型としてのみ使用できます。
- **sampler*** 型は、テクスチャ オブジェクトへのハンドルです。プログラムの仮パラメータまたは関数は、**sampler*** 型にすることができます。**sampler*** 変数のその他の定義は許可されていません。**sampler*** 変数は、**in** パラメータとして、別の関数へ渡すことによってのみ使用できます。**sampler*** 変数への代入は許可されず、**sampler*** として評価される式も許可されません。**sampler*** 型のうち、**sampler**、 **sampler1D**、**sampler2D**、**sampler3D**、**samplerCUBE** および **samplerRECT** は常に定義済みです。基底となる **sampler** 型は、個々の sampler* 型が有効である任意のコンテキスト内で使用可能です。ただし、**sampler** 変数の使用はプログラム中で一貫している必要があります。たとえば同じプログラム中で、**sampler1D** と **sampler2D** の両方のかわりに使うことはできません。
  フラグメント プロファイルでは、**sampler**、 **sampler1D**、**sampler2D**、**sampler3D** および **samplerCUBE** データ型を完全にサポートする必要があります。フラグメント プロファイルでは、**samplerRECT** データ型を部分的にサポート（174 ページの「型の部分的なサポート」を参照）する必要があり、このデータ型を完全にサポートすることもできます。
  頂点プロファイルでは、6 つの sampler* データ型を部分的にサポートする必要があり、それらのデータ型を完全にサポートすることもできます。
- **array** 型は、同じ型の 1 つ以上の要素からなる集合です。**array** 変数は 1 つのインデックスを持ちます。
- 一部の array 型は、オプションとして、**packed** 型修飾子を使用して packed 型として指定できます。packed 型の格納形式は、対応する unpacked 型の格納形式とは異なる場合があります。packed 型の格納形式は実装に依存しますが、特定のコンパイラとプロファイルの組合せに対して一貫している必要があります。特定のプロファイルで packed 型に対してサポートされる演算は、同じプロファイルで対応する unpacked 型に対してサポートされる演算とは異なる場合があります。各プロファイルではパックされた配列を最大許容サイズで定義できますが、少なくとも、パックされたベクトル（1 次元の配列）型についてはサイズ 4、パックされた行列（2 次元の配列）型については 4x4 のサイズをサポートする必要があります。
- 1 つの宣言で配列の配列を宣言する場合、**packed** 修飾子は、一番外側の配列のみを参照します。ただし、**packed** キーワードを使用して **typedef** で配列

の最初のレベルを宣言し、次に 2 番目のステートメントでこの型のパックされた配列を宣言することにより、パックされた配列のパックされた配列を宣言できます。アンパックされた配列のパックされた配列は宣言できません。

- サポートされる任意の数値データ型 ***TYPE*** について、実装では、次のパックされた配列型をサポートする必要があります。これをベクトル型と呼びます。これらの型の型識別子は、グローバル スコープ内で事前定義する必要があります。

```
typedef packed TYPE TYPE1[1];
typedef packed TYPE TYPE2[2];
typedef packed TYPE TYPE3[3];
typedef packed TYPE TYPE4[4];
```

たとえば実装では、型識別子 **float1**、**float2**、**float3**、**float4** と、その他サポートされているすべての数値型の型識別子を事前定義する必要があります。

- サポートされる任意の数値データ型 ***TYPE*** について、実装では、次のパックされた配列型をサポートする必要があります。これを行列型と呼びます。実装では、これらの型を表す型識別子を（グローバル スコープで）事前定義する必要もあります。

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

たとえば実装では、型識別子 **float2x1**、**float3x3**、**float4x4** などを事前定義する必要があります。**typedef** は、***TYPE_rows_X_columns*** という通常の行列の命名規則に従います。**float4x4 a** と宣言したとすると、**a[3]** は **a._m30_m31_m32_m33** に等しくなります。
どちらの式も、行列の 3 行目を取り出します。

- 実装では、定数インデックスを使用したベクトルと行列のインデクシングをサポートする必要があります。
- **struct** 型は、型が異なる可能性のある 1 つ以上のメンバで構成される集合です。

型の部分的なサポート

この仕様は、一部のデータ型の部分的なサポートを規定するものです。型の部分的なサポートの要件は、次のとおりです。

- 型を使用する定義と宣言がサポートされます。
- 代入と、その型のオブジェクトのコピー（関数のパラメータを渡す際の暗黙的なコピーも含む）がサポートされます。
- 最上位レベル関数のパラメータは、その型を使用して定義できます。

型が部分的にサポートされている場合、その型を使用して変数を定義できますが、この変数に対して有効な演算を実行することはできません。型の部分的なサポートによって、異なるプロファイルを対象としたコード内で、データ構造の共有が容易になります。

型のカテゴリ

- 整数型カテゴリには、**cint** 型と **int** 型が含まれます。
- 浮動小数点型カテゴリには、**cfloat** 型、**float** 型 **half** 型および **fixed** 型が含まれます。（浮動小数点は、実際には浮動小数点または固定 / 小数部を表すことに注意してください。）
- 数値型カテゴリには、整数型と浮動小数点型が含まれます。
- コンパイル時型カテゴリには、**cfloat** 型と **cint** 型が含まれます。これらの型は、定数の型変換のためにコンパイラによって使用されます。
- 具象型カテゴリには、コンパイル時型カテゴリに含まれない型がすべて含まれます。
- スカラ型カテゴリには、数値型カテゴリのすべての型、**bool** 型、およびコンパイル時型カテゴリのすべての型が含まれます。この仕様で ***<category>*** 型と表記（たとえば、数値型）する場合は、そのカテゴリに含まれる型の 1 つ（たとえば **float**、**half** または **fixed**）を意味するものとします。

定数

定数は、明示的または暗黙的に型指定できます。明示的に定数を型指定する場合は、C と同様に、定数の型を示す次のような 1 文字の接尾辞を付けます。

- **f** = **float**
- **d** = **double**
- **h** = **half**
- **x** = **fixed**

明示的に型指定されない定数は、暗黙的に型指定されます。定数に小数点が含まれる場合、その定数は暗黙的に **cfloat** として型指定されます。定数に小数点が含まれない場合、その定数は暗黙的に **cint** として型指定されます。

デフォルトでは、定数の基数は 10 です。C との互換性のために、整数の 16 進定数は定数に **0x** という接頭辞を付けることによって指定でき、整数の 8 進定数は定数に **0** という接頭辞を付けることによって指定できます。

コンパイル時定数の畳み込みは、原則的にランタイム時と同様の精度で行われます。コンパイル プロファイルの一部では、ハードウェア依存の精度の柔軟性が認められています。この場合、コンパイラはそのプロファイルで特定のデータ型に対して可能な最大のハードウェア精度で定数の畳み込みを実行するのが理想的です。

ランタイム時の精度で定数の畳み込みを行えない場合は、各数値データ型について次に示す精度を使用して畳み込みを行うこともできます。

- **float**: s23e8（**fp32**）の IEEE 単精度浮動小数点
- **half**: s10e5（**fp16**）の浮動小数点、IEEE セマンティクスあり
- **fixed**: s1.10 の固定小数点数、[-2, 2) にクランプ
- **double**: s52e11（**fp64**）の IEEE 倍精度浮動小数点
- **int**: 符号付き 32 ビット整数

型修飾子

オブジェクトの型は、1 つ以上の修飾子を使用して修飾できます。修飾子は、オブジェクトにのみ適用されます。修飾子は、式で使用される場合はオブジェクトの値から削除されます。修飾子は次のとおりです。

- **const**
  const修飾されたオブジェクトの値は、初期代入後に変更できません。**const** 修飾された、パラメータ以外のオブジェクトの定義には、イニシャライザが含まれている必要があります。コンパイル時に指定される値は **const** として修飾されますが、明示的な修飾も可能です。
  static const の値は、コンパイル後に変更できなくなるため、その値をコンパイル時の畳み込みで使用することができます。これに対して、**uniform const** はプログラムの実行時にのみ **const** であり、その値は実行と実行の間にランタイムを介して変更できます。
- **in** と **out**
  仮パラメータは **in**、**out**、またはその両方（**in out** または **inout** を使用）として修飾できます。デフォルトでは、仮パラメータは **in** 修飾されます。**in** 修飾されたパラメータは、値渡しのパラメータと等価です。**out** 修飾されたパラメータは結果渡しのパラメータと等価であり、**inout** 修飾されたパラメータは値 / 結果渡しのパラメータと等価です。**out** 修飾されたパラメータは **const** 修飾できず、デフォルト値を持つこともできません。

型変換

一部の型変換は暗黙的に行うことができますが、それ以外は型キャストが必要です。暗黙的な変換で警告が発生する場合は、明示的な型キャストを使用することでそれを抑制できます。明示的な型キャストは、C と同じ構文で指定します。***variable*** を **float4** 型にキャストするには、**(float4)*variable*** とします。

- スカラの変換
  スカラ数値型は、他のスカラ数値型に暗黙的に変換できます。暗黙的な変換では警告が発生することがあり、精度が低下する可能性もあります。スカラ オブジェクト型は、互換性のある他のスカラ オブジェクト型に暗黙的に変換できます。互換性のないスカラ オブジェクト型への変換、あるいはオブジェクト型と数値型の相互の変換は、たとえ明示的なキャストを使用しても許可されていません。**sampler** は、**sampler1D**、**sampler2D**、**sampler3D**、**samplerCUBE** および **samplerRECT** と互換です。これ以外のオブジェクト型には互換性がありません。つまり **sampler1D** は **sampler** とは互換であっても、**sampler2D** とは等しくありません。
  スカラ型は、型に互換性のあるベクトルおよび行列に暗黙的に変換できます。スカラは、ベクトルまたは行列のすべての要素に複製されます。また、スカラ型を構造体の各メンバに正しくキャストできる場合には、スカラ型を構造体型に明示的にキャストすることもできます。
- ベクトルの変換
  ベクトルは、スカラ型に変換できます（ベクトルの最初の要素が選択されます）。この変換を暗黙的に行った場合は、警告が発生します。ベクトルは、サイズが等しく要素の型に互換性のある他のベクトルに暗黙的に変換することもできます。
  ベクトルは、より小さいベクトルまたは全体サイズの等しい行列に変換できますが、明示的なキャストを使用しない場合は警告が発生します。
- 行列の変換
  行列は、スカラ型に変換できます（要素 (0,0) が選択されます）。ベクトルと同様、この変換を暗黙的に行った場合は警告が発生します。行列は、サイズと形が等しく要素の型が等しい行列に暗黙的に変換することもできます。
  行列は、より小さい行列型に変換する（右上の部分行列が選択されます）ことも、全体サイズの等しいベクトルに変換することもできますが、明示的なキャストを使用しない場合は警告が発生します。
- 構造体の変換
  構造体は、その最初のメンバの型に、あるいはメンバ数が等しい他の構造体型に明示的にキャストできますが、これは **struct** の各メンバが新しい **struct** の対応するメンバに変換可能な場合に限ります。**struct** の暗黙的な変換は許可されていません。

- 配列の変換
  配列型の変換は許可されていません。

ここまでに説明した型変換を、表 6 にまとめます。表の各項目の意味は次のとおりですが、脚注に注意してください。

- 可能： 暗黙的または明示的に変換可能
- 警告： 変換可能だが、暗黙的な場合は警告が発生
- 明示的： 明示的なキャストのみ可能
- 不可： 型変換は不可

表 6 型変換

変換後の型	変換前の型				
	スカラ	ベクトル	行列	構造体	配列
スカラ	可能	警告	警告	明示的[i]	不可
ベクトル	可能	可能[ii]	警告[iii]	明示的[i]	不可
行列	可能	警告[iii]	可能[ii]	明示的[i]	不可
構造体	明示的	不可	不可	明示的[iv]	不可
配列	不可	不可	不可	不可	不可

i. 変換前の最初のメンバが変換可能な場合のみ可能。
ii. 変換後のサイズが変換前より大きい場合は不可。変換後のサイズが変換前より小さい場合は**「警告」**。
iii. 変換前と変換後で全体サイズが等しい場合のみ可能。
iv. 変換前と変換後のメンバ数が等しく、かつ変換前の各メンバが対応するメンバに変換可能な場合のみ可能。

明示的なキャストは、次のとおりです。

- コンパイル時型の式に適用される場合は、コンパイル時型
- 数値型またはコンパイル時型の式に適用される場合は、数値型
- 要素数が等しい他のベクトル型に適用される場合は、数値ベクトル型
- 行数と列数が等しい他の行列に適用される場合は、数値行列型

型の等価性

型 T1 は、次のいずれかが真である場合に型 T2 と等価です。

- T2 が T1 と等価である場合。
- T1 と T2 が同じスカラ、ベクトル、または構造体型である場合。
  パックされた配列は、同じサイズのアンパックされた配列と等価ではありません。
- T1 が T2 の **typedef** 名である場合。
- T1 と T2 が、同数の要素を持つ等価な型の配列である場合。
- 修飾していない T1 と T2 が等価であり、両方の型が同じ修飾子を持つ場合。
- T1 と T2 が、等価な戻り型と同数のパラメータを持つ関数であり、対応するすべてのパラメータがペアごとに等価である場合。

型の昇格規則

cfloat 型と **cint** 型は、通常の算術変換動作と関数オーバーロード規則（181 ページの「関数のオーバーロード」を参照）を除いて、**float** および **int** のように動作します。

2 項演算子に対する通常の算術変換は、次のように定義されます。

1. 一方のオペランドが **double** の場合、もう一方も **double** に変換されます。
2. 一方のオペランドが **float** の場合、もう一方のオペランドも **float** に変換されます。
3. 一方のオペランドが **half** の場合、もう一方のオペランドも **half** に変換されます。
4. 一方のオペランドが **fixed** の場合、もう一方のオペランドも **fixed** に変換されます。
5. 一方のオペランドが **cfloat** の場合、もう一方のオペランドも **cfloat** に変換されます。
6. 一方のオペランドが **int** の場合、もう一方のオペランドも **int** に変換されます。
7. これら以外の場合は、両方のオペランドが **cint** 型になります。

変換は、演算の実行前に行われることに注意してください。

代入

オブジェクトまたはコンパイル時に型指定される値へ式を代入すると、式はそのオブジェクトまたは値の型に変換されます。その結果の値が、オブジェクトまたは値に代入されます。

代入式（**=**、***=** など）の定義は、C と同じです。代入式は、代入後に左辺のオペランドの値を持ちますが、左辺値ではありません。代入式の型は、左辺のオペランドの型になります。ただし、左辺のオペランドに修飾された型がある場合は、左辺のオペランドの型から修飾子をとったものになります。左側のオペランドの値は、次のオペレーションに入る前に変更されます。

スカラからベクトルへの展開

2 項演算子がベクトルおよびスカラに適用される場合、スカラを各成分に複製することによって、スカラが同じサイズのベクトルに自動的に型昇格されます。3 項演算子 **?:** も、展開をサポートします。2 項演算の規則が、最初に 2 番目と 3 番目のオペランドに適用され、続いてこの結果と 1 番目のオペランドに適用されます。

ネームスペース

C の場合と同様に、2 つのネームスペースがあります。各ネームスペースには、C の場合と同様に複数のスコープがあります。

- タグ ネームスペース： **struct** タグで構成される
- 通常ネームスペース：
  - **typedef** 名（**struct** 宣言からの自動 **typedef** を含む）
  - 変数
  - 関数名

配列と添え字

配列は、**packed** として宣言できる（171 ページの「型」を参照）点を除いて、C の場合と同様に宣言されます。Cg の配列は基本型タイプであるため、関数とプログラムへの配列のパラメータは、ポインタ構文ではなく配列構文を使用して宣言する必要があります。同様に、**array** 型指定されたオブジェクトの代入は、ポインタのコピーではなく配列のコピーを意味します。

サイズ **[1]** の配列は宣言可能ですが、同等の非配列型とは異なる型とみなされます。

Cg 言語では現在ポインタをサポートしていないため、配列の格納順序は、アプリケーションがパラメータを頂点プログラムまたはフラグメント プログラムに渡すときにのみ参照できます。したがって、コンパイラは現在、適合するとみなした場合はテンポラリ変数を自由に割り当てることができます。

配列の配列の宣言と使用は、C の場合と同じスタイルで行います。つまり、2 次元配列 **A** を、

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

と宣言した場合、次の記述はすべて真です。

- 配列は **A[*row*][*column*]** のようにインデクシングされる。
- 配列は、次の式を使用したコンストラクタで構築できる。

```
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[2][0], A[2][1], A[2][2], A[2][3]},
       {A[3][0], A[3][1], A[3][2], A[3][3]} };
```

- **A[0]** は **{A[0][0], A[0][1], A[0][2], A[0][3]}** と等価である。

配列を含む **structs** がサポートされている必要があります。

配列に関する最小要件

各プロファイルは、特定の種類の配列を部分的にサポートする必要があります。これは、すべてのプロファイルでベクトルと行列をサポートするためです。頂点プロファイルの場合はさらに、uniform 型パラメータとして渡されるライティングのステートの配列（光源番号によってインデクシングされる）と、uniform 型パラメータとして渡されるスキニング行列の配列をサポートするためでもあります。

各プロファイルは、ベクトルと行列の添え字、コピーおよびスウィズルをサポートする必要があります。ただし、実行時に計算されるインデックスでの添え字をサポートする必要はありません。

頂点プロファイルは、プログラムへの uniform 型パラメータであるパックされていない配列、またはプログラムへの uniform 型パラメータである構造体の要素であるパックされていない配列に対して、次の操作をサポートする必要があります。この要件は、配列が間接的に uniform 型のプログラム パラメータである場合（つまり、その配列またはその配列を含む構造体、あるいはその両方が **in** パラメータのチェインを介して渡された場合）にも適用されます。サポートする必要のある 2 つの操作は次のとおりです。

- 実行時に計算される値やコンパイル時の値による、右辺値の添え字。
- 対応する仮パラメータが **in** として宣言されている関数に、配列全体をパラメータとして渡す。

次の操作は、明示的にサポートする必要はありません。

- 左辺値による添え字
- コピー
- 乗算、加算、比較などその他の演算

配列の添え字に右辺値がある場合、結果は式となり、この式は **uniform** 型のパラメータとはみなされなくなります。したがって、この式が配列の場合、その後の使用では配列の使用法に関する標準規則に従う必要があります。

これらの規則は、数値型の配列に限定されないため、配列が **uniform** 型プログラム パラメータである場合は **struct** の配列、行列の配列およびベクトルの配列がサポートされることを意味します。最大配列サイズは、使用可能なレジスタ数またはその他のリソースによって制限される場合があり、コンパイラはこのような場合にエラー メッセージを発行できます。ただし、各プロファイルは、少なくとも **float arr[8]**、**float4 arr[8]** および **float4x4 arr[4][4]** の各サイズをサポートする必要があります。

フラグメント プロファイルでは、任意のサイズの配列に対する演算をサポートする必要はありません。ベクトルと行列のサポートのみ必要です。

関数のオーバーロード

修飾していないパラメータの型によって定義を区別でき、オープンプロファイルの競合が発生しないかぎり（170 ページの「プロファイルによる関数のオーバーロード」を参照）、複数の関数を同じ名前で定義できます。

関数の一致規則は次のとおりです。

1. コール側のスコープ内で同じ名前を持つ可視の関数すべてを、関数候補のセットに追加します。
2. プロファイルが現在のコンパイル プロファイルと競合する関数は除外します。
3. 仮パラメータの数が正しくない関数は除外します。関数候補で仮パラメータ数が超過しており、超過した各パラメータがデフォルト値を持っている場合、その関数は除外しません。
4. セットが空の場合は失敗します。
5. 連続した実パラメータの式それぞれについて、次の操作を実行します。
   a. 実パラメータの型が、セット内の任意の関数において対応する仮パラメータの非修飾型と一致する場合は、対応するパラメータが厳密に一致しないパラメータをすべて削除します。
   b. 実パラメータの型について、任意の関数の非修飾型の仮パラメータへの昇格が定義されている場合は、これに当てはまらないすべての関数をセットから削除します。
   c. 実パラメータの型を任意の関数の非修飾型の仮パラメータに変換する、有効な暗黙的キャストがある場合は、このキャストのない関数を削除します。
   d. 失敗します。

6. プロファイルに基づいて関数を選択します。
   a. コンパイル プロファイルと厳密に一致するプロファイルを持つ関数が1つでもある場合は、厳密に一致しない関数をすべて破棄します。
   b. あるいは、コンパイル プロファイルと一致するワイルドカード プロファイルを持つ関数が 1 つでもある場合は、候補セットの中で最も類似性の高いワイルドカード プロファイルを特定します。最も類似性の高いこのワイルドカード プロファイルを持つ関数以外は、すべて破棄されます。任意のワイルドカード プロファイルが特定のプロファイルと比べてどの程度類似しているかは、プロファイルの仕様によって決定されます。
7. セットに複数の関数が残った場合は、失敗します。

グローバル変数

グローバル変数は、C の場合と同様に宣言および使用されます。非静的な uniform 型変数には、セマンティクスが関連付けられている場合があります。非静的な uniform 型変数は、ランタイム API を介して値を設定できます。

初期化されていない変数の使用

初期化されていない変数をプログラムで使用することは誤りです。ただし、コンパイル時のデータフロー分析によって検出が可能だったとしても、コンパイラはこのようなエラーを検出しません。初期化されていない変数の読取りによって取得された値は未定義です。最上位レベル関数によって返される際に発生する、変数の暗黙的な使用にも同じ規則が適用されます。特に、最上位レベル関数が **struct** を返し、その **struct** の一部の要素が一度も書き込まれていない場合、その要素の値は未定義です。

注意： **変数は、0 に初期化するよう定義することはできません。変数がプログラマによって正しく初期化されたかどうかをコンパイラが判断できないと、パフォーマンスの低下につながるためです。**

プリプロセッサ

Cg の各プロファイルでは、**#if**、**#define** など ANSI C 標準のプリプロセッサ機能をサポートする必要があります。ただし、マクロ型の **#define** や、**#include** ディレクティブの使用をサポートする必要はありません。

バインディング セマンティクスの概要

ストリーム処理アーキテクチャでは、異なるプログラマブル ユニット間をデータ パケットが流れます。たとえば、GPU では、頂点データのパケットが、アプリケーションから頂点プログラムへ流れます。

パケットは一方のプログラム（この場合、アプリケーション）によって生成され、もう一方のプログラム（頂点プログラム）で使用されるため、この両者間のインタフェースを定義する方法が必要です。Cg では、ユーザは 2 つの異なるアプローチのどちらかを選択して、これらのインタフェースを定義できます。

最初のアプローチでは、パケットの各要素にバインディング セマンティクスを関連付けます。これが、名前によるバインドのアプローチです。たとえば、バインディング セマンティクス **FOO** での出力は、バインディング セマンティクス **FOO** での入力に転送されます。各プロファイルにより、ユーザはこのセマンティック ネームスペースに任意の識別子を定義できます。または、許可される識別子を定義済みのセットに限定することもできます。これらの定義済みの名前は、多くの場合、ハードウェア レジスタまたは API リソースの名前に対応しています。

場合によっては、定義済みの名前で、ハードウェアの非プログラマブルな部分を制御できます。たとえば、頂点プログラムは、通常はラスタライザに転送される位置座標を計算し、この位置座標は、バインディング セマンティクス **POSITION** での出力に格納されます。

各プロファイルとも、定義済みバインディング セマンティクスに 2 つのネームスペースがあります。**in** 変数に使用されるネームスペースと、**out** 変数に使用されるネームスペースです。2 つのネームスペースがあるために、バインディング セマンティクスを使用して、変数が **in** であるか **out** であるかを暗黙的に指定することはできません。

データ パケットを定義する 2 番目のアプローチでは、パケット内に存在するデータを記述し、その格納方法をコンパイラで決定させます。Cg では、ユーザは、データ パケットのすべての内容を **struct** に配置することにより、データ パケットの内容を記述できます。**struct** がこの方式で使用される場合、これをコネクタと呼びます。2 つのアプローチは、後で説明するように、相互に排他的ではありません。コネクタ アプローチでは、ユーザはユーザ指定のセマンティック バインディングとコンパイラによる自動バインディングを組み合せて使用できます。

バインディング セマンティクス

バインディング セマンティクスは、次の 3 通りの方法のいずれかで、最上位レベル関数への入力に関連付けることができます。

- バインディング セマンティクスは、関数の仮パラメータ宣言で指定します。関数への仮パラメータの構文は次のとおりです。

  [**const**] [**in | out | inout**]
  <type> ***<identifier>*** **[: *<binding-semantic>*][= *<initializer>*]**

- 仮パラメータが **struct** の場合、バインディング セマンティクスは、**struct** が定義される時点に **struct** の要素で指定できます。

  ```
  struct <struct-tag> {
  <type> <identifier>[ : <binding-semantic>];
  /*...*/ };
  ```

- 関数への入力が暗黙的に行われる場合（すなわち関数で非静的グローバル変数が読まれることによって入力が行われる場合）、バインディング セマンティクスは非静的グローバル変数が宣言される時点で指定されます。

  ```
  <type> <identifier>[ : <binding-semantic>][ = <initializer>]
  ```

  非静的グローバル変数が **struct** の場合、上の第 2 項で説明したように、バインディング セマンティクスは **struct** が定義される時点で指定されます。

- バインディング セマンティクスは、同様の方法で、最上位レベル関数の出力に関連付けることができます。

  ```
  <type> <identifier> ( <parameter-list> )[ : <binding-semantic>]
  { <body> }
  ```

  出力値にセマンティックを指定できるもう 1 つの方法は、**struct** を返して、**struct** が定義される時点で **struct** の要素にバインディング セマンティクスを指定することです。また、出力が仮パラメータである場合には、入力のバインディング セマンティクスの指定に使用したのと同じ方法でバインディング セマンティクスを指定することもできます。

セマンティクスのエイリアス化

セマンティクスは、入力時コピーおよび出力時コピーのモデルに従う必要があります。このため、2 つの異なる変数に同じ入力バインディング セマンティクスが使用される場合、これらの変数は同じ値で初期化されますが、変数はその後エイリアス化されません。出力のエイリアス化は不正ですが、実装でこれを検出する必要はありません。コンパイラが、出力バインディング セマンティクスをエイリアス化するプログラムでエラーを発行しない場合、結果は未定義です。

構造体内のセマンティクスに対する制限

特定のプロファイルでは、特定の **struct** 内で入力バインディング セマンティクスと出力バインディング セマンティクスを混在させることは不正です。つまり、特定の最上位レベル関数では、**struct** は入力のみまたは出力のみのいずれかである必要があります。同様に、**struct** は uniform 型の入力のみ、または非 uniform 型の入力のみから構成される必要があります。バインディング セマンティクスを使用して、1 つの **struct** 内に両方を混在させることは不正です。

バインディング セマンティクスに関するその他の詳細

次の規則は、多少冗長になりますが、理解を深めるためここに示します。

- セマンティクス名では、大文字小文字が区別されません。
- main 以外の関数へのパラメータに添付されたセマンティクスは無視されます。
- 入力セマンティクスは、複数の変数によってエイリアス化できます。
- 出力セマンティクスはエイリアス化できません。

構造体を使用したバインディング セマンティクス（コネクタ）の定義

Cg の各プロファイルでは、最上位レベル プログラムへの非 uniform 型のすべての入力（または出力）変数に対して、バインディング セマンティクスを指定しなくて済む方法もあります。そのためには、すべての非 uniform 型の変数を 1 つの **struct** に含める必要があります。コンパイラは、この **struct** を返すプログラムが、この **struct** を入力として使用する任意のプログラムと相互作用できるような方法で、この構造体の要素をハードウェア リソースに自動的に割り当てます。

バインディング セマンティクスを省略するために、非 uniform 型の変数すべてを 1 つの **struct** に含めることは必須ではありません。バインディング セマンティクスは任意の入力または出力から省略可能であり、その入力または出力はコンパイラによって自動的にハードウェア リソースに割り当てられるからです。ただし、自動バインディングが実行される際に、あるプログラムの出力と他のプログラムの入力との相互操作を保証するためには、すべての変数を 1 つの **struct** に含める必要があります。

様々な理由により、ハードウェア リソースへの構造体要素の割当てを自動的に実行する際の特定の規則セットを選択できるようにすることをお薦めします。使用される規則セット（割当て規則識別子）は、次のように、コロン識別子表記を使用して構造体タグに添付される識別子によって指定できます。

```
struct <struct-tag> : <allocation-rule-identifier> {
/*... */ };
```

ハードウェア リソースへの構造体要素の割当ては、この構造体定義（および割当て規則識別子）にのみ依存する再生可能な方式で実行できます。特に、割当てアルゴリズムは、特定の関数が構造体の要素を読み取るまたは書き込む方法に依存しないことがあります。

実装では、割当て規則識別子の指定をオプションにすることを選択できます。割当て規則識別子を省略すると、割当て規則のデフォルト セットが暗黙的に使用されます。

struct の一部の要素には、明示的にバインディング セマンティクスを指定できますが、それ以外には指定できません。コンパイラの自動割当ては、これらの明示的バインディングを優先する必要があります。明示的に指定可能なバインディ

ング セマンティクスのセットは、割当て規則識別子によって定義します。この機能は、ハードウェアの非プログラマブルな部分に対して読取りまたは書込みを行うハードウェア レジスタに変数をバインドする際に使用するのが最も一般的です。たとえば、一般的な頂点プログラム プロファイルで、出力 **struct** には明示的に指定された **POSITION** セマンティクスを持つ要素が含まれます。この要素は、ハードウェア ラスタライザの制御に使用されます。

任意の割当て規則識別子について、追加の制限事項を特定のバインディング セマンティクスに関連付けることができます。たとえば、特定のバインディング セマンティクスを持つ変数からの読取りは不正な場合があります。

プログラムがデータを受け取り、返す方法

プログラムは、コンパイル時のメイン エントリ ポイントとして指定されている、単なる非静的関数です。プログラムへの varying 型入力は、この最上位レベル関数の varying 型 **in** パラメータに由来します。プログラムへの uniform 型入力は、最上位レベル関数の uniform 型 **in** パラメータに由来するか、あるいは最上位レベル関数またはそれがコールする任意の関数によって参照される非静的グローバル変数に由来します。プログラムの出力は、関数の戻り値（常に暗黙的に varying 型）と、任意の **out** パラメータ（varying 型）に由来します。

プログラムのパラメータのうち **sampler*** 型のものは、暗黙的に **const** です。

ステートメント

ステートメントは、このマニュアルの他の場所で例外が記述されていないかぎり、C の場合と同様に表現されます。加えて、次の特徴があります。

- **if**、**while** および **for** ステートメントでは、適切な場所に **bool** 式が必要です。
- 代入は、**=** を使用して実行されます。代入演算子は C と同様に値を返すので、代入を連鎖できます。
- 新しい **discard** ステートメントは、現在のデータ要素（現在の頂点または現在のフラグメントなど）についてプログラムの実行を終了し、その出力を抑制します。頂点プロファイルでは、**discard** のサポートを省略することもできます。

if、while および for ステートメントに関する最小要件

各ステートメントの最小要件は、次のとおりです。

- すべてのプロファイルで **if** をサポートする必要がありますが、古いハードウェアの場合は厳密に必須ではありません。
- ループの反復回数をコンパイル時に決定できる場合は、すべてのプロファイルで **for** および **while** ループをサポートする必要があります。

「コンパイル時に決定できる」とは、次のような意味です。

処理ブロック内の定数の伝播と畳み込みを利用して、ループ反復式をコンパイル時に評価でき、定数値を伝播する変数が、制御ステートメント（**if**、**for**または**while**）や**?:**定数の内部で左辺値として出現しないこと。

各プロファイルでは、これより汎用的な定数伝播のテクニックをサポートすることもできますが、必須ではありません。

- 各プロファイルでは、完全に汎用的な **for** および **while** ループをサポートすることもできます。

新しいベクトル演算子

ベクトル型には、次の新しい演算子が定義されています。

- 構成演算子： ***<typeID>*(...)**
  複数のスカラまたは複数の短いベクトルから、1つのベクトルを作成します。

  ```
  float4(scalar, scalar, scalar, scalar)
  float4(float3, scalar)
  ```

- 行列構成演算子： ***<typeID>*(...)**
  複数の行から 1 つの行列を作成します。各行は、複数のスカラとして、または適切なサイズを持つスカラとベクトルの任意の組合せとして指定できます。

  ```
  float3x3(1, 2, 3, 4, 5, 6, 7, 8, 9)
  float3x3(float3, float3, float3)
  float3x3(1, float2, float3, float3, 1, 1, 1)
  ```

- スウィズル演算子： **(.)**

  ```
  a = b.xxyz;  // A swizzle operator example
  ```

  - 演算子の後に、少なくとも 1 つのスウィズル文字が必要です。
  - スウィズル文字には 2 つのセットがあり、それらを混在させることはできません。
    1 つ目のセットは **xyzw = 0123**、2 つ目のセットは **rgba = 0123** です。
  - ベクトル スウィズル演算子は、ベクトルまたはスカラのみに適用できます。
  - ベクトル スウィズル演算子をスカラに適用すると、長さ 1 のベクトルに適用した場合と同じ結果を得られます。
    したがって、**myscalar.xxx** と **float3(myscalar,myscalar,myscalar)** は同じ結果になります。
  - スウィズル文字を1つのみ指定した場合、結果は長さ1のベクトルではなくスカラです。したがって、式 **b.y** はスカラを返します。

- 定数スカラをスウィズルする場合は、小数点文字（.）の使用に両義性があるため注意する必要があります。たとえば、スカラから 3 成分ベクトルを作成するには、次のいずれかの式を使用します。

  (1).xxx or **1..xxx** or **1.0.xxx** or **1.0f.xxx**

- 返されるベクトルのサイズは、スウィズル文字の数によって決定されます。そのため、返されるベクトルのサイズは元のベクトルより大きい場合もあります。
  たとえば、**float2(0,1).xxyy** と **float4(0,0,1,1)** は同じ結果になります。

- 行列スウィズル演算子 :
  <type><rows>x<columns> という形式の行列型の場合、

  <matrixObject>._m<row><col>[_m<row><col>][...]

  という表記法を使用して、個々の行列要素にアクセスしたり（***<row><col>*** ペアが1つのみの場合）、行列の要素からベクトルを構成したり（***<row><col>*** ペアが複数存在する場合）することができます。行と列の番号は 0 から始まります。
  次に例を示します。

```
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_m11_m22_m33;
```

  - **D3DMatrix** データ型との互換性を保つために、Cg では、_ 記号の後の **m** を省略した次のような形式を使用する、1 から始まるスウィズルも許可されています。

    <matrixObject>._<row><col>[_<row><col>][...]

    この形式では、***<row>*** と ***<col>*** のインデックスは、C 標準のように 0 から始まるのではなく 1 から始まります。したがって、次の 2 つの形式は機能的に等価です。

```
float4x4 myMatrix;
float4   myVec;

// These two statements are functionally equivalent:
myVec = myMatrix._m00_m23_m11_m31;
myVec = myMatrix._11_34_22_42;
```

    1 から始まるインデクシングが原因で混乱することがあるため、後の表記法は使用しないことを強くお薦めします。

- 行列スウィズルは、行列にのみ適用できます。スウィズルを使用して行列から複数の要素を抽出する場合、結果は適切にサイズ設定されたベクトルになります。スウィズルを使用して行列から 1 つの要素を抽出する場合、結果はスカラになります。

- 書込みマスク演算子： (.)
  書込みマスク演算子は、ベクトルである左辺値にのみ適用できます。ベクトルまたは行列の特定の要素に代入し、他の要素は変更しないことも可能です。制限事項は、要素を反復できないことのみです。

算術の精度および範囲

一部のハードウェアは、IEEE の算術規則に厳密には準拠していない場合があります。固定小数点データ型には、IEEE 定義の規則がありません。

最適化を行った場合に、最適化していないコードとは多少異なる結果を生成することが許可されています。定数の畳み込みは、ほぼ正しい精度と範囲で行う必要がありますが、ビット単位で厳密な結果を生成する必要はありません。コンパイラでは、このような最適化を禁止するオプション、またはビット単位で厳密な方式で最適化が行われることを保証するオプションを提供した方がよいでしょう。

演算子の優先順位

Cg では、C と Cg で共通の演算子については C と同様の優先順位が使用されます。

スウィズル演算子と書込みマスク演算子（.）は、構造体メンバ演算子（.）および配列インデックス演算子（**[]**）と同じ優先順位です。

演算子の拡張

ベクトルと行列をサポートするために、C 標準の算術演算子（**+**、**-**、*****、**/**、**%**、**unary（単項）-**）が拡張されています。ベクトルと行列のサイズは、標準数学規則に従って、一致している必要があります。スカラからベクトルへの昇格（179 ページの「スカラからベクトルへの展開」を参照）により、これらの規則が緩和されます。

表 7　拡張された演算子

演算子	説明
`M[n][m]`	*n* 行および *m* 列の行列
`V[n]`	*n* 成分のベクトル
`-V[n] -> V[n]`	単項ベクトルの符号反転
`-M[n] -> M[n]`	単項行列の符号反転
`V[n] * V[n] -> V[n]`	要素ごとの *

表 7　拡張された演算子 （続き）

演算子	説明
V[*n*] / V[*n*] -> V[*n*]	要素ごとの /
V[*n*] % V[*n*] -> V[*n*]	要素ごとの %
V[*n*] + V[*n*] -> V[*n*]	要素ごとの +
V[*n*] - V[*n*] -> V[*n*]	要素ごとの -
M[*n*][*m*] * M[*n*][*m*] -> M[*n*][*m*]	要素ごとの *
M[*n*][*m*] / M[*n*][*m*] -> M[*n*][*m*]	要素ごとの /
M[*n*][*m*] % M[*n*][*m*] -> M[*n*][*m*]	要素ごとの %
M[*n*][*m*] + M[*n*][*m*] -> M[*n*][*m*]	要素ごとの +
M[*n*][*m*] - M[*n*][*m*] -> M[*n*][*m*]	要素ごとの -

演算子

ブール

&& || !

ブール演算子は、**bool** とパックされた **bool** ベクトルに適用できます。この場合、ブール演算子は成分ごとに適用され、同じサイズの結果ベクトルが生成されます。各オペランドは、同じサイズの **bool** ベクトルである必要があります。

&&および**||**の両辺は常に評価されます。Cのように短絡評価されることはありません。

比較

< > <= >= != ==

比較演算子は、数値ベクトルに適用できます。両方のオペランドが同じサイズのベクトルである必要があります。比較演算は成分ごとに実行され、同じサイズの **bool** ベクトルが生成されます。

比較演算子は、**bool** ベクトルにも適用できます。関係比較の目的で、**true** は 1、**false** は 0 として扱われます。比較演算は成分ごとに実行され、同じサイズの **bool** ベクトルが生成されます。

比較演算子は、数値または **bool** スカラにも適用できます。

算術

+ - * / % ++ -- unary- unary+

算術演算子 **%** は、C の場合と同様に剰余演算子です。この演算子は、**cint** または **int** 型の 2 つのオペランドにのみ適用できます。

/ または **%** が **cint** または **int** のオペランドに対して使用される場合、整数の **/** および **%** に対する C の規則が適用されます。

対応する演算子が Cg でサポートされていれば、算術演算子と代入を組み合せた C の演算子（**+=** など）も使用できます。

条件演算子

?:

最初のオペランドの型が **bool** の場合、2 番目と 3 番目のオペランドについては次のいずれかの条件を満たす必要があります。

- 両方のオペランドが構造体型と互換性がある。
- 両方のオペランドが数値型または **bool** 型のスカラである。
- 両方のオペランドが数値型または **bool** 型のベクトルで、2 つのベクトルが 4 以下の同じサイズである。

最初のオペランドが **bool** のパックされたベクトルの場合、条件選択は成分ごとに実行されます。2 番目と 3 番目の両方のオペランドは、1 番目のオペランドと同じサイズの数値ベクトルである必要があります。

C とは異なり、2 番目と 3 番目のオペランドの式における副作用は、条件にかかわらず常に実行されます。

その他の演算子

(typecast) ,

Cg では、C と同じ型キャスト演算子とカンマ演算子がサポートされています。

予約語

次に、Cg の予約語を示します。

asm*	**asm_fragment**	**auto**
bool	**break**	**case**
catch	**char**	**class**
column major	**compile**	**const**
const_cast	**continue**	**decl***
default	**delete**	**discard**
do	**double**	**dword***
dynamic_cast	**else**	**emit**
enum	**explicit**	**extern**
false	**fixed**	**float***
for	**friend**	**get**
goto	**half**	**if**
in	**inline**	**inout**
int	**interface**	**long**
matrix*	**mutable**	**namespace**
new	**operator**	**out**
packed	**pass***	**pixelfragment***
pixelshader*	**private**	**protected**
public	**register**	**reinterpret_cast**
return	**row major**	**sampler**
sampler_state	**sampler1D**	**sampler2D**
sampler3D	**samplerCUBE**	**shared**
short	**signed**	**sizeof**
static	**static_cast**	**string***
struct	**switch**	**technique***
template	**texture***	**texture1D**
texture2D	**texture3D**	**textureCUBE**
textureRECT	**this**	**throw**
true	**try**	**typedef**
typeid	**typename**	**uniform**
union	**unsigned**	**using**
vector*	**vertexfragment***	**vertexshader***
virtual	**void**	**volatile**
while	***__identifier*** (先頭はアンダースコア 2 つ)	

Cg 標準ライブラリ関数

GPU プログラミングを簡略化するため、Cg には組込み関数と、バインディング セマンティクスを持つ事前定義済みの構造体のセットが用意されています。これらの関数の詳細は、19 ページの「Cg 標準ライブラリ関数」を参照してください。

頂点プログラム プロファイル

Cg 言語の頂点プログラム プロファイルに固有のいくつかの機能は、すべての頂点プログラム プロファイルで同じように実装される必要があります。

座標出力の計算

頂点プログラム プロファイルでは、プログラムが座標出力を計算する必要があります（計算するのが通例です）。この同次クリップ空間座標は、ハードウェア ラスタライザによって使用されるもので、**POSITION**（または下位互換用の **HPOS**）の出力バインディング セマンティクスとともにプログラム出力に格納される必要があります。

位置座標の不変性

多くのグラフィック API では、頂点ごとの計算を指定する方法として 2 つの選択肢があります。組込みの構成可能な固定機能パイプラインを使用する方法と、ユーザ作成の頂点プログラムを指定する方法です。ユーザがこの 2 つの方法を併用したい場合は、最初の方法で計算される座標が、2 番目の方法で計算される座標とビット単位で等しいように保証されることが望ましいといえます。この位置座標の不変性は、マルチパス レンダリングで特に重要です。

位置座標の不変性のサポートは、Cg 頂点プロファイルではオプションですが、これをサポートする頂点プロファイルでは次の規則が適用されます。

- 固定機能パイプラインとの位置座標の不変性は次の2つの条件が満たされる場合に保証されます。
  - 頂点プログラムのコンパイルで、位置座標の不変性を示すコンパイラ オプション（**-posinv** など）が使用される。
  - 頂点プログラムが、座標を次のように計算する。

    ```
    OUT_POSITION = mul(MVP, IN_POSITION)
    ```

    ここで
    OUT_POSITION は、**float4** 型で、**POSITION** または **HPOS** の出力バインディング セマンティクスを持つ変数（または構造体要素）。
    IN_POSITION は、**float4** 型で、**POSITION** の入力バインディング セマンティクスを持つ変数（または構造体要素）。
    MVP は、**float4x4** 型で、固定機能のモデルビュー射影行列と、同じ値が入力される、入力バインディング セマンティクスを持つ uniform 型変数

（または構造体要素）。（このバインディング セマンティクスの名前は、現在プロファイル固有です。OpenGL プロファイルの場合、セマンティクス **_GL_MVP** が推奨されています。）

- 最初の条件が満たされ、2 番目の条件が満たされない場合、コンパイラは警告を発生します。
- 実装では、2 番目の条件をより汎用的な形で受け入れることもできます（変数が元の入力および出力からコピー伝播される、など）が、この汎用性は必須ではありません。

出力のバインディング セマンティクス

表 8 に示すように、頂点プログラム プロファイルには 2 つの出力バインディング セマンティクスがあります。

表 8　頂点の出力バインディング セマンティクス

名前	意味	型	デフォルト値
POSITION	同次クリップ空間座標、ラスタライザに転送	float4	未定義
PSIZE	ポイント サイズ	float	未定義

各プロファイルでは、特定の動作をする出力バインディング セマンティクスを追加で定義でき、それらの定義は一般的に使用されるプロファイル間では一定でなければなりません。

フラグメント プログラム プロファイル

Cg 言語のフラグメント プログラム プロファイルに固有のいくつかの機能は、すべてのフラグメント プログラム プロファイルで同じように実装される必要があります。

出力のバインディング セマンティクス

表 9 に示すように、フラグメント プログラム プロファイルには 3 つの出力バインディング セマンティクスがあります。各プロファイルでは、特定の動作をする出力バインディング セマンティクスを追加で定義でき、それらの定義は一般的に使用されるプロファイル間では一定でなければなりません。

表 9　フラグメントの出力バインディング セマンティクス

名前	意味	型	デフォルト値
COLOR	RGBA の出力カラー	float4	未定義
COLOR0	COLOR と同じ	-	-
DEPTH	フラグメント深度値（値域 [0,1]）	float	ラスタライザからの補間された深度値（値域 [0,1]）

プログラムで出力カラー アルファを 1.0 としたい場合は、**COLOR** 出力の **w** 成分に明示的に 1.0 の値を記述する必要があります。Cg 言語では、この出力のデフォルト値は定義されていません。

注意：　ターゲット　ハードウェアがこの出力のデフォルト値を使用する場合、ユーザが指定した明示的な書込みがハードウェアのデフォルト値と一致すれば、コンパイルではこの書込みを無視することもできます。このようなデフォルトは、Cg 言語では公開されていません。

逆に、**DEPTH** 出力については Cg 言語でデフォルト値を定義しています。このデフォルト値は、ラスタライザから取得される補間された深度です。意味的には、このデフォルト値はフラグメント プログラムの実行を開始する時点で出力にコピーされます。

前述したように、バインディング セマンティクスが出力に適用される際、出力変数の型はバインディング セマンティクスの型と一致している必要はありません。たとえば、推奨はされませんが次のような式が許可されます。

```
struct myfragoutput {
  float2 mycolor : COLOR; }
```

この場合、変数はプログラムの完了時に暗黙的に（**typecast** によって）セマンティクにコピーされます。変数のベクトル サイズがセマンティクのベクトル サイズより小さい場合、セマンティクのうちの過剰な成分はデフォルト値を受け取り、デフォルト値がなければ未定義となります。上の例では、出力カラーの **R** および **G** 成分は ***mycolor*** から取得され、**B** および **A** 成分は未定義となります。

付録 B
言語プロファイル

この付録では、Cg コンパイラでサポートされている次の各プロファイルで使用できる言語機能について説明します。

- DirectX 頂点シェーダ 2.X プロファイル（`vs_2_0`、`vs_2_x`）
- DirectX ピクセル シェーダ 2.X プロファイル（`ps_2_0`、`ps_2_x`）
- OpenGL ARB 頂点プログラム プロファイル（`arbvp1`）
- OpenGL ARB フラグメント プログラム プロファイル（`arbfp1`）
- OpenGL NV_vertex_program 2.0 プロファイル（`vp30`）
- OpenGL NV_fragment_program プロファイル（`fp30`）
- DirectX 頂点シェーダ 1.1 プロファイル（`vs_1_1`）
- DirectX ピクセル シェーダ 1.x プロファイル（`ps_1_1`、`ps_1_2`、`ps_1_3`）
- OpenGL NV_vertex_program 1.0 プロファイル（`vp20`）
- OpenGL NV_texture_shaderおよびNV_register_combinersプロファイル（`fp20`）

どのプロファイルでも、その言語機能は 165 ページの「Cg 言語仕様」の Cg 言語仕様で説明されている全機能のサブセットです。

DirectX 頂点シェーダ 2.X プロファイル（vs_2_0、vs_2_x）

DirectX 頂点シェーダ 2.0 プロファイルは、Cg のソース コードを、DirectX 9 VS 2.0 頂点シェーダ[1] および DirectX 9 VS 2.0 拡張頂点シェーダにコンパイルする際に使用されます。

- **プロファイル名**:
  vs_2_0（DirectX 9 VS 2.0 頂点シェーダ用）
  vs_2_x（DirectX 9 VS 2.0 拡張頂点シェーダ用）
- **起動方法**:
  コンパイラ オプション**-profile vs_2_0**または**-profile vs_2_x**を使用。

この節では、**vs_2_0** および **vs_2_x** プロファイルを使用することによって、開発者が記述する Cg のソース コードにどのような影響があるかを説明します。

概要

vs_2_0 プロファイルは、DirectX VS 2.0 頂点シェーダの機能と一致するように Cg を制限します。**vs_2_x** プロファイルは、**vs_2_0** プロファイルと同じですが、動的フロー制御（分岐）などの拡張機能が利用できます。

メモリ

DirectX 9 頂点シェーダでは、命令とデータ用のメモリ量が限られています。

プログラム命令の制限

DirectX 9 頂点シェーダの命令は、256 個までに制限されています。プログラムのコンパイル時に 256 個を超える命令を生成する必要がある場合、コンパイラはエラーをレポートします。

ベクトル レジスタの制限

同様に、プログラム パラメータと一時的な結果を保持するレジスタ数も制限されています。具体的には、読取り専用ベクトル レジスタが 256 個、読取り / 書込みベクトル レジスタが 12 ～ 32 個です。プログラムのコンパイル時に使用可能な数よりも多くのレジスタを必要とする場合、コンパイラはエラーを生成します。

1. DirectX VS 2.0 頂点シェーダと、コンパイラにより生成されるコードを理解するには、DirectX 9 SDK のドキュメントで頂点シェーダのリファレンスを参照してください。

ステートメントと演算子

vs_2_0 プロファイルを使用する場合、**if**、**while**、**do** および **for** ステートメントは、定義されるループがアンロール可能である場合のみ使用できます。拡張版ではない VS 2.0 シェーダには、動的な分岐がないためです。

vs_2_x プロファイルを使用する場合は、**DynamicFlowControlDepth** オプションが 0 でないかぎり、**if**、**while** および **do** ステートメントがサポートされます。

比較演算子（**>**、**<**、**>=**、**<=**、**==**、**!=**）とブール演算子（**||**、**&&**、**?:**）は使用可能です。ただし、論理演算子（**&**、**|**、**^**、**~**）は使用できません。

データ型

このプロファイルでは、データ型の実装は次のとおりです。

- **float** データ型は、IEEE 32 ビットの単精度として実装されます。
- **half** および **double** データ型は、**float** として扱われます。
- **int** データ型は、浮動小数点の操作を利用してサポートされ、このとき浮動小数点型から除算、モジュロおよびキャストの適切な切捨てを行うための命令が追加されます。
- **fixed** または **sampler*** データ型はサポートされませんが、コア言語仕様によって各プロファイルではこれらのデータ型に必要な最小限の部分的なサポートが提供されます。つまり、変数に対して演算を実行しないかぎり、これらのデータ型を使用する変数を宣言することは可能です。

配列の使用

配列の変数インデクシングは、配列が uniform 型定数であるかぎり認められます。互換性の理由から、変数式でインデックスされる配列は uniform 宣言のみ必要であり、const 宣言は不要です。ただし、後から変数式でインデックスされる配列に書込みを行った場合、その結果は予測できません。

頂点プログラムのインデクシングで許可されていないため、配列データはパックされません。配列の各要素は、1 つの 4-float プログラム パラメータレジスタを必要とします。たとえば、**float arr[10]**、**float2 arr[10]**、**float3 arr[10]** および **float4 arr[10]** は、いずれも 10 個のプログラム パラメータ レジスタを使用します。

行列の配列よりも、ベクトルの配列にアクセスする方が効率的です。行列にアクセスするには、floor 計算したうえで定数を乗算して、レジスタ インデックスを計算する必要があります。ベクトル（およびスカラ）は 1 つのレジスタを使用するため、floor 関数も乗算も不要です。行列の配列を使用するよりも、インデックスを事前に乗算したベクトルの配列を使用して行列スキニングを行う方が高速です。

バインディング

uniform 型データのバインディング セマンティクス

表 10 は、**vs_2_0** および **vs_2_x** プロファイルにおける uniform 型パラメータで有効なバインディング セマンティクスをまとめたものです。

表 10 vs_2_0/vs_2_x の uniform 型入力バインディング セマンティクス

バインディング セマンティクス名	対応するデータ
register(c0)-register(c255) **C0-C255**	**定数レジスタ** [0..95]。 **エイリアスの** c0-c95（**小文字**）**も許可される。** **複数の定数レジスタを必要とする変数（行列など）で使用した場合、セマンティクスは使用される最初のレジスタを指定。**

varying 型入力 / 出力データのバインディング セマンティクス

このプロファイルでは、バインディング セマンティクス名のみ指定する必要があります。頂点パラメータ入力レジスタは、動的に割り当てられます。**POSITION** 以外のすべてのセマンティクス名は、その後に 0 ～ 15 の数字を付けることができます。

表 11 vs_2_0/vs_2_x の varying 型入力バインディング セマンティクス

POSITION	**PSIZE**
BLENDWEIGHT	**BLENDINDICES**
NORMAL	**TEXCOORD**
COLOR	**TANGENT**
TESSFACTOR	**BINORMAL**

表 12 は、**vs_2_0** および **vs_2_x** プロファイルにおける varying 型出力パラメータで有効なバインディング セマンティクスをまとめたものです。

これらは、DirectX 9 頂点シェーダにおいて出力レジスタにマッピングされます。

表 12 vs_2_0/vs_2_x の varying 型出力バインディング セマンティクス

バインディング セマンティクス名	対応するデータ
POSITION	出力位置座標： oPos
PSIZE	出力ポイント サイズ： oPts
FOG	出力フォグ値： oFog
COLOR0-COLOR1	出力カラー値： oD0、oD1
TEXCOORD0-TEXCOORD7	出力テクスチャ座標： oT0-oT7

オプション

vs_2_x プロファイルには、プロファイル固有の次のオプションがあります。

DynamicFlowControlDepth=<*n*>	（***n***=0 または 24。デフォルトは 24）
NumTemps=<*n*>	（12<=***n***<=32。デフォルトは 16）
Predication	（デフォルトは true）

DirectX ピクセル シェーダ 2.X プロファイル（ps_2_0、ps_2_x）

DirectX ピクセル シェーダ 2.0 プロファイルは、Cg のソース コードを、DirectX 9 PS 2.0 ピクセル シェーダ[2]および DirectX 9 PS 2.0 拡張頂点シェーダにコンパイルする際に使用されます。

- **プロファイル名**:
  ps_2_0（DirectX 9 PS 2.0 ピクセル シェーダ用）
  ps_2_x（DirectX 9 PS 2.0 拡張ピクセル シェーダ用）
- **起動方法**:
  コンパイラ オプション**-profile ps_2_0**または**-profile ps_2_x**を使用。

ps_2_0 プロファイルは、DirectX PS 2.0 ピクセル シェーダの機能と一致するように Cg を制限します。**ps_2_x** プロファイルは、**ps_2_0** プロファイルと同じですが、任意のスウィズルなどの拡張機能が利用できます。また、命令数の上限が引き上げられ、テクスチャ命令やテクスチャ依存読込みに関する制限がなくなり、プレディケーションがサポートされます。

この節では、これらのプロファイルを使用する場合の Cg の機能と制限について説明します。

メモリ

プログラム命令の制限

DirectX 9ピクセル シェーダでは、ピクセル シェーダでの命令数に制限があります。

- PS 2.0（**ps_2_0**）ピクセル シェーダでは、テクスチャ命令が 32 個、算術命令が 64 個までに制限されています。
- 拡張 PS 2.0（**ps_2_x**）ピクセル シェーダでは、命令の最大合計数が 96 ～ 1024 個までに制限されています。
  拡張ピクセル シェーダの場合、テクスチャ命令に関する個別の制限はありません。

プログラムのコンパイル時に許容最大数を超える命令を生成する必要がある場合、コンパイラはエラーをレポートします。

ベクトル レジスタの制限

同様に、プログラム パラメータと一時的な結果を保持するレジスタ数も制限されています。具体的には、読取り専用ベクトル レジスタが 32 個、読取り / 書込みベクトル レジスタが 12 ～ 32 個です。プログラムのコンパイル時に使用可能な数よりも多くのレジスタを必要とする場合、コンパイラはエラーを生成します。

2. DirectX PS 2.0 ピクセル シェーダの機能と、コンパイラにより生成されるコードを理解するには、DirectX 9 SDK のドキュメントでピクセル シェーダのリファレンスを参照してください。

言語コンストラクトとサポート

データ型

このプロファイルでは、データ型の実装は次のとおりです。

- **float** データ型は、IEEE 32 ビットの単精度として実装されます。
- **half**、**fixed** および **double** データ型は、**float** として扱われます。**half** データ型を使用して、ピクセル シェーダ命令の部分的な精度のヒントを指定できます。
- **int** データ型は、浮動小数点の操作を使用してサポートされます。
- **sampler*** 型は、テクスチャの取出しに使用されるサンプラ オブジェクトを指定するためにサポートされます。

ステートメントと演算子

vs_2_0 プロファイルを使用する場合、**if**、**while**、**do** および **for** ステートメントは、定義されるループがアンロール可能である場合のみ使用できます。拡張版ではない VS 2.0 シェーダには、動的な分岐がないためです。現在の Cg の実装では、拡張版である **ps_2_x** シェーダにも同じ制限があります。

比較演算子（**>**、**<**、**>=**、**<=**、**==**、**!=**）とブール演算子（**||**、**&&**、**?:**）は使用可能です。ただし、論理演算子（**&**、**|**、**^**、**~**）は使用できません。

配列および構造体の使用

配列の変数インデクシングは許可されていません。配列および構造体データはパックされません。

バインディング

uniform 型データのバインディング セマンティクス

表 13 は、**ps_2_0** および **ps_2_x** プロファイルにおける uniform 型パラメータで有効なバインディング セマンティクスをまとめたものです。

表 13 ps_2_0/ps_2_x の uniform 型入力バインディング セマンティクス

バインディング セマンティクス名	対応するデータ
register(s0)-register(s15) **TEXUNIT0-TEXUNIT15**	Texunit **ユニット *n*。*n* の値域は** [0..15]。**sampler* 型の** uniform **型入力でのみ使用可能。**
register(c0)-register(c31) **C0-C31**	Texunit **ユニット *N*。*N* の値域は** [0..31]。uniform **型入力でのみ使用可能。**

varying 型入力 / 出力データのバインディング セマンティクス

表 14 は、**ps_2_0** および **ps_2_x** プロファイルにおける varying 型入力パラメータで有効なバインディング セマンティクスをまとめたものです。

表 14 ps_2_0/ps_2_x の varying 型入力バインディング セマンティクス

バインディング セマンティクス名	対応するデータ（型）
COLOR0	入力カラー 0（float4）
COLOR1	入力カラー 1（float4）
TEXCOORD0-TEXCOORD7	入力テクスチャ座標（float4）

表 15 は、**ps_2_0** および **ps_2_x** プロファイルにおける varying 型出力パラメータで有効なバインディング セマンティクスをまとめたものです。

表 15 ps_2_0/ps_2_x の varying 型出力バインディング セマンティクス

バインディング セマンティクス名	対応するデータ
COLOR、COLOR0	出力カラー（float4）
DEPTH	出力深度（float4）

オプション

ps_2_x プロファイルには、プロファイル固有の次のオプションがあります。

NumTemps=<*n*>	（12<=***n***<=32。デフォルトは 32）
NumInstructionSlots=<*n*>	（96<=***n***<=1024。デフォルトは 1024）
Predication=<*b*>	（***b***=0 または 1。デフォルトは 1）
ArbitrarySwizzle=<*b*>	（***b***=0 または 1。デフォルトは 1）
GradientInstructions=<*b*>	（***b***=0 または 1。デフォルトは 1）
NoDependentReadLimit=<*b*>	（***b***=0 または 1。デフォルトは 1）
NoTexInstructionLimit=<*b*>	（***b***=0 または 1。デフォルトは 1）

この実装での制限

現在、このプロファイルの実装には次のような制限があります。

- 拡張ピクセル シェーダで、動的フロー制御はサポートされません。
- ピクセル シェーダで、複数のカラー出力はサポートされません。サポートされるのは、**Color0** のみです。

OpenGL ARB 頂点プログラム プロファイル（arbvp1）

OpenGL ARB 頂点プログラム プロファイルは、Cg ソース コードを、**GL_ARB_vertex_program** 拡張のバージョン 1.0 と互換性のある頂点プログラムにコンパイルする際に使用されます。

- **プロファイル名**： **arbvp1**
- **起動方法**： コンパイラ オプション **-profile arbvp1** を使用。

この節では、**arbvp1** プロファイルを使用する場合の Cg の機能と制限について説明します。

概要

- **arbvp1** プロファイルは、出力の形式が異なることと、OpenGL のステートへ容易にアクセスできることを除いて、**vp20** プロファイルに類似しています。
- **ARB_vertex_program** の機能は、**NV_vertex_program** および DirectX 8 頂点シェーダの機能と同等であるため、プログラマが記述する Cg ソース コードに関する制限事項は、**NV_vertex_program**[3] プロファイルの場合と同じです。

OpenGL ステートへのアクセス

arbvp1 プロファイルでは、**vp20** プロファイルとは異なり、Cg プログラムが OpenGL ステートを直接参照できます。ただし、**vp20** および **dx8vs** プロファイルと互換性のある Cg プログラムを記述したい場合は、Cg ランタイムを使って必要なステートを uniform 型変数に設定する別のメカニズムを使用する必要があります。ARB 頂点アセンブリ プログラムの機能により、OpenGL ステートの一部をステートの変更として自動的にプログラム パラメータ レジスタに書き込むことができるので、コンパイルはその機能を利用します。このステート追跡機能は、OpenGL ドライバが処理しています。構造体として定義される **glstate** という特別な変数があり、これを使用すると、ARB 頂点プログラムが参照できる OpenGL ステートはすべての部分を参照できます。次の段落では、アクセス可能な **glstate** フィールドの 3 つのリストを示します。配列のインデックスは **0** として記載されていますが、配列の上限より小さい正の整数を使用すれば、配列にはアクセスできます。たとえば、**GL_MAX_LIGHTS** が少なくとも **2** であると仮定して、第 2 光源のディフューズ要素にアクセスする場合は、**glstate.light[1].diffuse** を使用します。

3. このプロファイルでサポートされるデータ型、ステートメントおよび演算子の詳細な説明は、222 ページの「DirectX 頂点シェーダ 1.1 プロファイル（vs_1_1）」を参照してください。

表 16 は、アクセスできる **float4x4** 型の **glstate** フィールドのリストです。

表 16 float4x4 glstate フィールド

glstate.matrix.modelview[0]	glstate.matrix.projection
glstate.matrix.mvp	glstate.matrix.texture[0]
glstate.matrix.palette[0]	glstate.matrix.program[0]
glstate.matrix.inverse.modelview[0]	glstate.matrix.inverse.projection
glstate.matrix.inverse.mvp	glstate.matrix.inverse.texture[0]
glstate.matrix.inverse.palette[0]	glstate.matrix.inverse.program[0]
glstate.matrix.transpose.modelview[0]	glstate.matrix.transpose.projection
glstate.matrix.transpose.mvp	glstate.matrix.transpose.texture[0]
glstate.matrix.transpose.palette[0]	glstate.matrix.transpose.program[0]
glstate.matrix.invtrans.modelview[0]	glstate.matrix.invtrans.projection
glstate.matrix.invtrans.mvp	glstate.matrix.invtrans.texture[0]
glstate.matrix.invtrans.palette[0]	glstate.matrix.invtrans.program[0]

表 17 は、アクセスできる **float4** 型の **glstate** フィールドのリストです。

表 17 float4 glstate フィールド

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

表 17 float4 glstate フィールド （続き）

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

表 18 は、アクセスできる **float** 型の **glstate** フィールドのリストです。

表 18 float glstate フィールド

glstate.point.size	glstate.point.attenuation

位置座標の不変性

- **arbvp1** プロファイルでは、コア言語仕様に記載されているように、位置座標の不変性がサポートされます。
- モデルビュー射影行列は、**_GL_MVP**のバインディング セマンティクスを使用して指定されません。

データ型

このプロファイルでは、データ型の実装は次のとおりです。

- **float**データ型は、**ARB_vertex_program**仕様での定義どおりに実装されます。
- **half** データ型は、**float** として実装されます。
- **fixed** または **sampler*** データ型はサポートされませんが、コア言語仕様によって各プロファイルではこれらのデータ型に必要な最小限の部分的なサポートが提供されます。つまり、変数に対して演算を実行しないかぎり、これらのデータ型を使用する変数を宣言することは可能です。

vp20 頂点プログラム プロファイルとの互換性

vp20 プロファイルで動作するプログラムは、Cg ランタイムを使用してすべての uniform 型パラメータ（OpenGL ステートを含む）を管理するかぎり、**arbvp1** プロファイルと互換性があります。そのため、**arbvp1** プロファイルと **vp20** プロファイルは、Cg ソース コードやアプリケーション プログラムを変更することなく、別々のプロファイルを指定するだけで互換的に使用できます。ただし、**glProgramParameterxxNV()** ルーチンのいずれかを使用している場合は、アプリケーション プログラムを変更し、対応する ARB 関数を使用する必要があります。

ARB には **glTrackMatrixNV()** に対応する関数がないため、**glTrackMatrixNV()** 関数と **arbvp1** プロファイルを使用するアプリケーションは、修正が必要です。1 つの解決策としては、**glstate** 構造体を使用して行列を参照するよう Cg ソース コードを変更し、**GL_ARB_vertex** サポートの一環としてその行列が OpenGL ドライバにより自動的に追跡されるようにする方法があります。別の解決策として、Cg ランタイム ルーチンの **cgBindUniformStateMatrix()** を使用し、必要なときに適切な行列をロードするという方法もあります。

arbvp1 プロファイルと **vp20** プロファイルの非互換性としてもう 1 つ可能性があるのは、varying 型入力セマンティクスの処理方法です。**vp20** プロファイルの場合、**POSITION** や **ATTR0** などのセマンティクス名は、**NV_vertex_program** で Vertex と Attribute 0 がエイリアスであるのと同様、相互に対するエイリアスです（241 ページの表 42「vp20 の varying 型入力バインディング セマンティクス」を参照）。**arbvp1** プロファイルの場合、**ARB_vertex_program** では従来の属性（頂点位置座標など）と汎用属性（Attribute 0 など）を独立させられるため、セマンティクス名はエイリアスではありません。このため、**arbvp1** プログラムが **ARB_vertex_program** の実装すべてに対して機能するためには、210 ページの表 20「arbvp1 の varying 型入力バインディング セマンティクス」で指定された規則に従うことが重要です。**arbvp1** の規則は、**vp20** プロファイルおよび **vp30** プロファイルと互換性があります。

定数のロード

Cg ランタイムを使用しないアプリケーションの場合には、Cg コンパイラ出力における **#const** 式で表されるような、プログラム パラメータ レジスタへの定数値のロードは不要です。コンパイラが生成する出力のはたらきによって、OpenGL ドライバが定数値をロードします。ただし、デフォルト定義を持つ uniform 型変数は、適切なプログラム パラメータ レジスタに定数値をロードする必要があります。ARB 頂点プログラムではその機能をサポートしていないためです。アプリケーション プログラムでは、Cg ランタイムを使用して **#default** コマンドを解析および処理するか、あるいは Cg ソース コードの中で uniform 型変数が初期化されないようにする必要があります。

バインディング

uniform 型データのバインディング セマンティクス

表 19 は、**arbvp1** プロファイルにおける uniform 型パラメータで有効なバインディング セマンティクスをまとめたものです。

表 19 arbvp1 の uniform 型入力バインディング セマンティクス

バインディング セマンティクス名	対応するデータ
`register(c0)-register(c255)` `C0-C255`	インデックス n を持つローカル パラメータ。n の値域は [0..255]。 エイリアスの c0-c255（小文字）も許可される。 複数の定数レジスタを必要とする変数（行列など）で使用した場合、セマンティクスは使用される最初のローカル パラメータを指定。

varying 型入力 / 出力データのバインディング セマンティクス

表 20 は、**arbvp1** プロファイルにおける uniform 型パラメータで有効なバインディング セマンティクスをまとめたものです。

arbvp1 への varying 型入力データに対するバインディング セマンティクスのセットは **POSITION**、**BLENDWEIGHT**、**NORMAL**、**COLOR0**、**COLOR1**、**TESSFACTOR**、**PSIZE**、**BLENDINDICES** および **TEXCOORD0-EXCOORD7** から構成されています。**TEXCOORD6** および **TEXCOORD7** のかわりに、**TANGENT** および **BINORMAL** も使用可能です。さらに、バインディング セマンティクス **ATTR0-ATTR15** のセットも使用できます。これらのセマンティクスと設定コマンドとの対応は、表にリストされています。

表 20 arbvp1 の varying 型入力バインディング セマンティクス

バインディング セマンティクス名	対応するデータ
POSITION	Vertex コマンドによる入力頂点
BLENDWEIGHT	WeightARB コマンドおよび VertexWeightEXT コマンドによる入力頂点ウェイト
NORMAL	Normal コマンドによる入力法線
COLOR0、DIFFUSE	Color コマンドによる入力プライマリ カラー
COLOR1、SPECULAR	SecondaryColorEXTコマンドによる入力セカンダリ カラー
FOGCOORD	FogCoordEXT コマンドによる入力フォグ座標
TEXCOORD0-TEXCOORD7	MultiTexCoord コマンドによる、入力テクスチャ座標（texcoord0-texcoord7）
ATTR0-ATTR15	VertexAttrib コマンドによる汎用属性 0-15
PSIZE、ATTR6	汎用属性 6

表 21 は、**arbvp1** プロファイルにおける varying 型出力パラメータで有効なバインディング セマンティクスをまとめたものです。これらのバインディング セマンティクスは、**ARB_vertex_program** 出力レジスタにマッピングされます。2 つのセットは、相互にエイリアスとして機能します。

表 21 arbvp1 の varying 型出力バインディング セマンティクス

バインディング セマンティクス名	対応するデータ
POSITION、HPOS	出力位置座標
PSIZE、PSIZ	出力ポイント サイズ
FOG、FOGC	出力フォグ座標
COLOR0、COL0	出力プライマリ カラー
COLOR1、COL1	出力セカンダリ カラー
BCOL0	出力背面プライマリ カラー
BCOL1	出力背面セカンダリ カラー
TEXCOORD0-TEXCOORD7、TEX0-TEX7	出力テクスチャ座標

注意： **arbvp1 プロファイルを使用する際に COLOR1 出力を有効にするためには、アプリケーションで glEnable(GL_COLOR_SUM_ARB) をコールする必要があります。**

このプロファイルでは、varying 型出力構造体のメンバに関するセマンティクスとして、**WPOS** を使用できます。ただしこれは、これらのバインディング セマンティクスに伴うメンバが参照されていない場合に限ります。このため Cg プログラムでは、**arbvp1** プロファイル プログラムの varying 型出力と、**fp30** プロファイル プログラムの varying 型入力を、同じ構造体で指定することが可能です。

OpenGL ARB フラグメント プログラム プロファイル（arbfp1）

OpenGL ARB フラグメント プログラム プロファイルは、Cg ソース コードを、OpenGL の **GL_ARB_fragment_program** 拡張のバージョン 1.0 と互換性のあるフラグメント プログラムにコンパイルする際に使用されます。[4]

- **プロファイル名**： **arbfp1**
- **起動方法**： コンパイラ オプション **-profile arbfp1** を使用。

arbfp1 プロファイルは、OpenGL ARB フラグメント プログラムの機能と一致するように Cg を制限します。この節では、**arbfp1** プロファイルを使用する場合の Cg の機能と制限について説明します。

メモリ

プログラム命令の制限

OpenGL ARB フラグメント プログラムでは、ARB フラグメント プログラムでの命令数に制限があります。

ARB フラグメント プログラムでは、**MAX_PROGRAM_INSTRUCTIONS_ARB** を使用して、基礎となる OpenGL 実装から問合せできる命令数に、最小値 72 という制限があります。テクスチャ命令の数（最小値 24）と、OpenGL 実装から問合せ可能な算術命令（最小値 48）にもそれぞれ制限があります。

プログラムのコンパイル時に最大許容数を超える命令を生成する必要がある場合、コンパイラはエラーをレポートします。

ベクトル レジスタの制限

同様に、ローカル プログラム パラメータを保持するレジスタ（最小値 24）と、一時的な結果を保持するレジスタ（最小値 16）を、OpenGL 実装から問合せできる数もそれぞれ制限があります。

プログラムのコンパイル時に使用可能な数よりも多くの一時またはローカル パラメータを必要とする場合、コンパイラはエラーを生成します。

4. OpenGL ARB フラグメント プログラムの機能と、コンパイラにより生成されるコードを理解するには、OpenGL Extensions ドキュメントで ARB フラグメント プログラム拡張を参照してください。

言語コンストラクトとサポート

データ型

このプロファイルでは、データ型の実装は次のとおりです。

- **float** データ型は、IEEE 32 ビットの単精度として実装されます。
- **half**、**fixed** および **double** データ型は、**float** として扱われます。
- **int** データ型は、浮動小数点の操作を使用してサポートされます。
- **sampler*** 型は、テクスチャの取出しに使用されるサンプラ オブジェクトを指定するためにサポートされます。

ステートメントと演算子

ARB フラグメント プログラム プロファイルを使用する場合、**while**、**do** および **for** ステートメントは、定義されるループがアンロール可能である場合のみ使用できます。ARB フラグメント プログラム 1 には、動的な分岐がないためです。

比較演算子（**>**、**<**、**>=**、**<=**、**==**、**!=**）とブール演算子（**||**、**&&**、**?:**）は使用可能です。ただし、論理演算子（**&**、**|**、**^**、**~**）は使用できません。

配列および構造体の使用

配列の変数インデクシングは許可されていません。配列および構造体データはパックされません。

バインディング

uniform 型データのバインディング セマンティクス

表 22 は、**arbfp1** プロファイルにおける uniform 型パラメータで有効なバインディング セマンティクスをまとめたものです。

表 22 arbfp1 の uniform 型入力バインディング セマンティクス

バインディング セマンティクス名	対応するデータ
register(s0)-register(s15) TEXUNIT0-TEXUNIT15	Texunit**画像ユニット*N*。*N*の値域は**[0..15]。**sampler* 型の uniform 型入力でのみ使用可能。**
register(c0)-register(c31) C0-C31	**ローカル パラメータ *N*。*N* の値域は** [0..31]。uniform **型入力でのみ使用可能。**

varying 型入力 / 出力データのバインディング セマンティクス

表 23 は、**arbfp1** プロファイルにおける varying 型入力パラメータで有効なバインディング セマンティクスをまとめたものです。

表 23 arbfp1 の varying 型入力バインディング セマンティクス

バインディング セマンティクス名	対応するデータ（型）
COLOR0	入力カラー 0（float4）
COLOR1	入力カラー 1（float4）
TEXCOORD0-TEXCOORD7	入力テクスチャ座標（float4）

表 24 は、**arbfp1** プロファイルにおける varying 型出力パラメータで有効なバインディング セマンティクスをまとめたものです。

表 24 arbfp1 の varying 型出力バインディング セマンティクス

バインディング セマンティクス名	対応するデータ
COLOR、COLOR0	出力カラー（float4）
DEPTH	出力深度（float4）

オプション

ARB フラグメント プログラム プロファイルには、プロファイル固有の次のオプションがあります。

NumTemps=<*n*>	（16<=***n***<=32。デフォルトは 32）
NumInstructionSlots=<*n*>	（72<=***n***<=1024。デフォルトは 1024）
NoDependentReadLimit=<*b*>	（***b***=0 または 1。デフォルトは 1）
NumTexInstructionSlots=<*n*>	（***n*** >=24）

実装での制限

現在、このプロファイルの実装には次のような制限があります。

- OpenGL ARB フラグメント プログラム プロファイルは、拡張機能が開発ベータ版の段階であり、そのサポートの一部はまだ使用できません。
- ARB フラグメント プログラムにおける OpenGL ステート アクセスは、未実装です。

OpenGL NV_vertex_program 2.0 プロファイル（vp30）

vp30 頂点プログラム プロファイルは、OpenGL の **NV_vertex_program2** 拡張で使用するために、Cg ソース コードを頂点プログラムにコンパイルする際に使用されます。

- **プロファイル名**： **vp30**
- **起動方法**： コンパイラ オプション **-profile vp30** を使用。

vp30 プロファイルは、**NV_vertex_program2** 拡張機能と一致するように Cg を制限します。この節では、**vp30** プロファイルを使用する場合の Cg の機能と制限について説明します。

位置座標の不変性

vp30 プロファイルでは、コア言語仕様に記載されているように、位置座標の不変性がサポートされます。

- モデルビュー射影行列は、**_GL_MVP**のバインディング セマンティクスを使用して指定する必要があります。**vp20** および **arbvp1** プロファイルとは異なり、このプロファイルでは、コンパイラはモデルビュー射影行列を使用して位置座標を変換する命令を発行します。
- 位置座標の計算は固定パイプラインの計算と比較して不変的であることがハードウェアによって保証されるため、アセンブリ コードで位置座標の不変性を指定するオプションは使用されません。

言語コンストラクト

データ型

このプロファイルでは、データ型の実装は次のとおりです。

- **float** データ型は、IEEE 32 ビットの単精度として実装されます。
- **half** データ型は、**float** として実装されます。
- **int** データ型は、浮動小数点の操作を利用してサポートされ、このとき浮動小数点型から除算、モジュロおよびキャストの適切な切捨てを行うための命令が追加されます。
- **fixed** または **sampler*** データ型はサポートされませんが、コア言語仕様によって各プロファイルではこれらのデータ型に必要な最小限の部分的なサポートが提供されます。つまり、変数に対して演算を実行しないかぎり、これらのデータ型を使用する変数を宣言することは可能です。

ステートメントと演算子

このプロファイルは、**vp20** プロファイルのスーパーセットです。**vp20** 用にコンパイルされるプログラムは、すべて **vp30** プロファイル用にもコンパイルする必要がありますが、その逆の場合は不要です。

vp20 プロファイルの機能に追加される **vp30** プロファイルの機能は、次のとおりです。

- **for**、**while** および **do** の各ループのサポートで、ループのアンロールは不要
- 非定数条件式が可能な **if/else** を完全サポート

バインディング

uniform 型データのバインディング セマンティクス

表 25 は、**vp30** プロファイルにおける uniform 型パラメータで有効なバインディング セマンティクスをまとめたものです。

表 25 vp30 の uniform 型入力バインディング セマンティクス

バインディング セマンティクス名	対応するデータ
register(c0)-register(c255) **C0-C255**	**定数レジスタ** [0..255]。 **エイリアスの** c0-c255（**小文字**）**も許可される。** **複数の定数レジスタを必要とする変数（行列など）で使用した場合、セマンティクスは使用される最初のレジスタを指定。**

varying 型入力 / 出力データのバインディング セマンティクス

表 26 は、**vp30** プロファイルにおける varying 型入力パラメータで有効なバインディング セマンティクスをまとめたものです。

TEXCOORD6 および **TEXCOORD7** のかわりに、**TANGENT** および **BINORMAL** も使用可能です。これらのバインディング セマンティクスは、**NV_vertex_program2** 入力属性パラメータにマッピングされます。2 つのセットは、相互にエイリアスとして機能します。

表 26 vp30 の varying 型入力バインディング セマンティクス

バインディング セマンティクス名	対応するデータ
POSITION、ATTR0	入力頂点、汎用属性 0
BLENDWEIGHT、ATTR1	入力頂点ウェイト、汎用属性 1
NORMAL、ATTR2	入力法線、汎用属性 2
COLOR0、DIFFUSE、ATTR3	入力プライマリ カラー、汎用属性 3
COLOR1、SPECULAR、ATTR4	入力セカンダリ カラー、汎用属性 4
TESSFACTOR、FOGCOORD、ATTR5	入力フォグ座標、汎用属性 5
PSIZE、ATTR6	入力ポイント サイズ、汎用属性 6
BLENDINDICES、ATTR7	汎用属性 7
TEXCOORD0-TEXCOORD7、ATTR8-ATTR15	入力テクスチャ座標 (texcoord0-texcoord7)、汎用属性 8-15
TANGENT、ATTR14	汎用属性 14
BINORMAL、ATTR15	汎用属性 15

表 27 は、**vp30** プロファイルにおける varying 型出力パラメータで有効なバインディング セマンティクスをまとめたものです。

これらのバインディング セマンティクスは、**NV_vertex_program2** 出力レジスタにマッピングされます。2 つのセットは、相互にエイリアスとして機能します。

表 27 vp30 の varying 型出力バインディング セマンティクス

バインディング セマンティクス名	対応するデータ
POSITION、HPOS	出力位置座標
PSIZE、PSIZ	出力ポイント サイズ
FOG、FOGC	出力フォグ座標
COLOR0、COL0	出力プライマリ カラー
COLOR1、COL1	出力セカンダリ カラー
BCOL0	出力プライマリ カラー
BCOL1	出力背面セカンダリ カラー

表 27 vp30 の varying 型出力バインディング セマンティクス （続き）

バインディング セマンティクス名	対応するデータ
TEXCOORD0-TEXCOORD7、TEX0-TEX7	出力テクスチャ座標
CLP0-CL5	出力クリップ座標

このプロファイルでは、varying 型出力構造体のメンバに関するセマンティクスとして、**WPOS** を使用できます。ただしこれは、これらのバインディング セマンティクスに伴うメンバが参照されていない場合に限ります。このため Cg プログラムでは、**vp30** プロファイル プログラムの varying 型出力と、**fp30** プロファイル プログラムの varying 型入力を、同じ構造体で指定することが可能です。

OpenGL NV_fragment_program プロファイル（fp30）

fp30 フラグメント プログラム プロファイルは、OpenGL の **NV_fragment_program** 拡張で使用するために、Cg ソース コードをフラグメント プログラムにコンパイルする際に使用されます。

- **プロファイル名**： **fp30**
- **起動方法**： コンパイラ オプション **-profile fp30** を使用。

この節では、**fp30** プロファイルを使用する場合の Cg の機能と制限について説明します。

言語コンストラクトとサポート

データ型

- **fixed** 型（s1.10 固定小数点数）がサポートされます。
- **half** 型（s10e5 浮動小数点数）がサポートされます。

パフォーマンスを重視する場合は、**fixed**、**half**、**float** の順で使用することをお薦めします。精度を重視する場合は、この逆になります。必要な精度を満たす中で、パフォーマンスが最も高いデータ型を使用してください。

ステートメントと演算子

- **if/else** を完全サポート
- コンパイラによりアンロールできる場合に限り、**for** および **while** ループが可能
- フレキシブル テクスチャ マッピングをサポート
- スクリーン空間導関数をサポート
- 配列の変数インデクシングは未サポート

バインディング

uniform 型データのバインディング セマンティクス

表 28 は、**fp30** プロファイルにおける uniform 型パラメータで有効なバインディング セマンティクスをまとめたものです。

表 28 fp30 の uniform 型入力バインディング セマンティクス

バインディング セマンティクス名	対応するデータ
register(s0)-register(s15) **TEXUNIT0-TEXUNIT15**	Texunit***N***。***N*** **の値域は** [0..15]。 **sampler* 型の** uniform **型入力でのみ使用可能。**
register(c0)-register(c31) **C0-C31**	**定数レジスタ** ***N***。***N*** **の値域は** [0..15]。 uniform **型入力でのみ使用可能。**

varying 型入力 / 出力データのバインディング セマンティクス

表 29 は、**fp30** プロファイルにおける varying 型入力パラメータで有効なバインディング セマンティクスをまとめたものです。

これらのバインディング セマンティクスは、**NV_fragment_program** 入力レジスタにマッピングされます。2 つのセットは、相互にエイリアスとして機能します。このプロファイルでは、varying 型入力構造体のメンバに関するバインディング セマンティクスとして、**POSITION**、**FOG**、**PSIZE**、**HPOS**、**FOGC**、**PSIZ**、**BCOL0**、**BCOL1** および **CLP0-LP5** も使用できます。ただしこれは、このバインディング セマンティクスに伴うメンバが参照されていない場合に限ります。このため Cg プログラムでは、**vp30** プロファイル プログラムの varying 型出力と、**fp30** プロファイル プログラムの varying 型入力を、同じ構造体で指定することが可能です。

表 29 fp30 の varying 型入力バインディング セマンティクス

バインディング セマンティクス名	対応するデータ（型）
COLOR0、COL0	**入力カラー** 0（**float4**）
COLOR1、COL1	**入力カラー** 1（**float4**）
TEXCOORD0-TEXCOORD7、 **TEX0-TEX7**	**入力テクスチャ座標（float4）**
WPOS	**ウィンドウ位置座標（float4）**

表 30 は、**fp2** プロファイルにおける varying 型出力パラメータで有効なバインディング セマンティクスをまとめたものです。

表 30 fp30 の varying 型出力バインディング セマンティクス

バインディング セマンティクス名	対応するデータ
COLOR、COLOR0、COL	出力カラー（float4）
DEPTH、DEPR	出力深度（float）

DirectX 頂点シェーダ 1.1 プロファイル（vs_1_1）

DirectX 頂点シェーダ 1.1 プロファイルは、Cg のソース コードを、DirectX 8.1 頂点シェーダおよび DirectX 9 VS 1.1 シェーダにコンパイルする際に使用されます。[5]

- **プロファイル名：** **vs_1_1**
- **起動方法：** コンパイラ オプション **-profile vs_1_1** を使用。

vs_1_1 プロファイルは、DirectX 頂点シェーダの機能と一致するように Cg を制限します。

この節では、**vs_1_1** プロファイルを使用する場合、開発者が記述する Cg のソース コードにどのような影響があるかを説明します。

メモリ上の制限

DirectX 8 頂点シェーダでは、命令とデータ用のメモリ量が限られています。

プログラム命令の制限

DirectX 8 頂点シェーダの命令は、128 個までに制限されています。プログラムのコンパイル時に 128 個を超える命令を生成する必要がある場合、コンパイラはエラーをレポートします。

ベクトル レジスタの制限

同様に、プログラム パラメータと一時的な結果を保持するレジスタ数も制限されています。具体的には、読取り専用ベクトル レジスタが 96 個、読取り / 書込みベクトル レジスタが 12 個です。プログラムのコンパイル時に使用可能な数よりも多くのレジスタを必要とする場合、コンパイラはエラーを生成します。

言語コンストラクトとサポート

データ型

このプロファイルでは、データ型の実装は次のとおりです。

- **float** データ型は、IEEE 32 ビットの単精度として実装されます。
- **half** および **double** データ型は、**float** として扱われます。
- **int** データ型は、浮動小数点の操作を利用してサポートされ、このとき浮動小数点型から除算、モジュロおよびキャストの適切な切捨てを行うための命令が追加されます。

5. DirectX VS 1.1 頂点シェーダと、コンパイラにより生成されるコードを理解するには、DirectX 8.1 SDK のドキュメントで頂点シェーダのリファレンスを参照してください。

- **fixed** または **sampler*** データ型はサポートされませんが、コア言語仕様によって各プロファイルではこれらのデータ型に必要な最小限の部分的なサポートが提供されます。つまり、変数に対して演算を実行しないかぎり、これらのデータ型を使用する変数を宣言することは可能です。

ステートメントと演算子

if、**while**、**do** および **for** ステートメントは、定義されるループがアンロール可能である場合のみ使用できます。VS 1.1 シェーダには、動的な分岐がないためです。

サブルーチン コールもないため、関数はすべてインラインです。比較演算子（**>**、**<**、**>=**、**<=**、**==**、**!=**）とブール演算子（**||**、**&&**、**?:**）は使用可能です。ただし、論理演算子（**&**、**|**、**^**、**~**）は使用できません。

配列の使用

配列の変数インデクシングは、配列が uniform 型定数であるかぎり認められます。互換性の理由から、変数式でインデックスされる配列は uniform 宣言のみ必要であり、const 宣言は不要です。ただし、後から変数式でインデックスされる配列に書込みを行った場合、その結果は予測できません。

頂点プログラムのインデクシングで許可されていないため、配列データはパックされません。配列の各要素は、1 つの 4-float プログラム パラメータレジスタを必要とします。たとえば、**float arr[10]**、**float2 arr[10]**、**float3 arr[10]** および **float4 arr[10]** は、いずれも 10 個のプログラム パラメータ レジスタを使用します。

行列の配列よりも、ベクトルの配列にアクセスする方が効率的です。行列にアクセスするには、floor 計算したうえで定数を乗算して、レジスタ インデックスを計算する必要があります。ベクトル（およびスカラ）は 1 つのレジスタを使用するため、floor 関数も乗算も不要です。行列の配列を使用するよりも、インデックスを事前に乗算したベクトルの配列を使用して行列スキニングを行う方が高速です。

定数

このプロファイルではリテラル定数を使用できますが、リテラル定数をプログラム自体に格納することはできません。かわりにコンパイラは、ロードする必要のあるプログラム パラメータ レジスタと定数のリストをコメントとして発行します。Cg ランタイム システムは、コンパイラの指示に従って定数のロードを処理します。

注意： Cg **ランタイム システムを使用しない場合は、定数が正しくロードされていることをプログラマが確認する必要があります。**

バインディング

uniform 型データのバインディング セマンティクス

表 31 は、**vs_1_1** プロファイルにおける uniform 型パラメータで有効なバインディング セマンティクスをまとめたものです。

表 31 vs_1_1 の uniform 型入力バインディング セマンティクス

バインディング セマンティクス名	対応するデータ
register(c0)-register(c95) **C0-C95**	**定数レジスタ** [0..95]。 **エイリアスの** c0-c95（**小文字**）**も許可される。** **複数の定数レジスタを必要とする変数（行列など）で使用した場合、セマンティクスは使用される最初のレジスタを指定。**

varying 型入力 / 出力データのバインディング セマンティクス

表 32 は、**vs_1_1** プロファイルにおける uniform 型パラメータで有効なバインディング セマンティクスをまとめたものです。これらは、DirectX 8.1 頂点シェーダにおいて入力レジスタにマッピングされます。

表 32 vs_1_1 の varying 型入力バインディング セマンティクス

バインディング セマンティクス名	対応するデータ
POSITION	**頂点シェーダの入力レジスタ： v0**
BLENDWEIGHT	**頂点シェーダの入力レジスタ： v1**
BLENDINDICES	**頂点シェーダの入力レジスタ： v2**
NORMAL	**頂点シェーダの入力レジスタ： v3**
PSIZE	**頂点シェーダの入力レジスタ： v4**
COLOR0、DIFFUSE	**頂点シェーダの入力レジスタ： v5**
COLOR1、SPECULAR	**頂点シェーダの入力レジスタ： v6**
TEXCOORD0-TEXCOORD7	**頂点シェーダの入力レジスタ： v7-v14**
TANGENT[i]	**頂点シェーダの入力レジスタ： v14**
BINORMAL	**頂点シェーダの入力レジスタ： v15**

i. **TANGENT** は **TEXCOORD7** のエイリアス。

表 33 は、**vs_1_x** プロファイルにおける varying 型出力パラメータで有効なバインディング セマンティクスをまとめたものです。これらは、DirectX 8.1 頂点シェーダにおいて出力レジスタにマッピングされます。

表 33 vs_1_1 の varying 型出力バインディング セマンティクス

バインディング セマンティクス名	対応するデータ
POSITION	出力座標： oPos
PSIZE	出力ポイント サイズ： oPts
FOG	出力フォッグ値： oFog
COLOR0-COLOR1	出力カラー値： oD0、oD1
TEXCOORD0-TEXCOORD7	出力テクスチャ座標： oT0-oT7

オプション

DirectX 9 で **vs_1_1** プロファイルを使用する場合は、varying 型入力を宣言する **dcl** ステートメントを生成するようコンパイラに指定する必要があります。オプション **-profileopts dcls** を指定すると、コンパイル出力に **dcl** ステートメントが追加されます。

DirectX ピクセル シェーダ 1.x プロファイル（ps_1_1、ps_1_2、ps_1_3）

DirectX ピクセル シェーダ 1_x プロファイルは、Cg のソース コードを、DirectX PS1.1、1.2、1.3 ピクセル シェーダのアセンブリにコンパイルする際に使用されます。

- **プロファイル名**:
  ps_1_1（DirectX PS 1.1 ピクセル シェーダ用）
  ps_1_2（DirectX PS 1.2 ピクセル シェーダ用）
  ps_1_3（DirectX PS 1.3 ピクセル シェーダ用）
- **起動方法**:
  次のコンパイラ オプションを使用。
  -profile ps_1_1
  -profile ps_1_2
  -profile ps_1_3

非推奨のプロファイル **dx8ps** も使用可能ですが、これは **ps_1_1** と同等です。

この節では、DirectX ピクセル シェーダ 1_x プロファイルを使用する際の機能と制限について説明します。

概要

DirectX PS 1.4 は現在、Cg プロファイルのサポート対象外です。これ以降の **ps_1_x** についての説明は、すべて **ps_1_1**、**ps_1_2** および **ps_1_3** を対象とします。

基礎となる命令セットとマシン アーキテクチャにより、これらのプロファイルでのプログラマビリティは、Cg のコンストラクトで許可されているものに比べて制限されています。[6] そのため、これらのプロファイルでは、Cg プログラムで可能な操作の内容に制限が加えられています。

これらのプロファイルと Cg との主な相違は、**ps_1_2** および **ps_1_3** で追加のテクスチャ処理演算が公開されていることと、**ps_1_3** では（制限された形で）深度値出力を利用できることです。

DirectX ピクセル シェーダ 1_x プロファイルでの演算は、テクスチャ処理演算と算術演算の 2 つのカテゴリに分類できます。テクスチャ処理演算はテクスチャ処理命令を生成する演算であり、算術演算は算術命令を生成する演算です。これらのプロファイルでの Cg プログラムは、生成できる命令数がテクスチャ処理命令は最大 4 つ、算術命令は最大 8 つまでに制限されています。最大数がかなり小さいため、これらのプロファイルで Cg コードを記述している間、ユーザはこの制限を強く意識する必要があります。

6. 基礎となる命令セット、その機能および制限事項の詳細は、DirectX ピクセル シェーダ 1.1、1.2 および 1.3 に関する MSDN のドキュメントを参照してください。

特定の単純な算術演算は、テクスチャ処理演算の入力に利用でき、算術演算の入力および出力にも、算術命令を生成せずに利用できます。ここからは、これらの演算を入力修飾子および出力修飾子と呼びます。

ps_1_x プロファイルでは、テクスチャ処理演算または算術演算をプログラム内でいつ使用できるかということも制限されています。テクスチャ処理演算は、以下の場合にかぎって算術演算の出力に依存します。

- 算術演算が、テクスチャ処理演算に対して有効な入力修飾子である。
- 算術演算が、複合テクスチャ処理演算の一部である（これについてのまとめは、補助テクスチャ関数を参照）。

修飾子

入力修飾子および出力修飾子を利用すると、算術命令を生成することなく単純な算術演算を実行できます。この場合、算術演算がその適用対象であるアセンブリ命令またはソース レジスタを修飾します。たとえば、

```
z = (x - 0.5 + y) / 2
```

という Cg の式では本来、次のようなピクセル シェーダ命令が生成されます（ただし **x** は **t0** 内、**y** は **t1** 内、**z** は **r0** 内にあるとします）。

```
add_d2 r0, t0_bias, t1
```

表 34 は、DirectX ピクセル シェーダ 1_x の各命令セット修飾子と、Cg プログラム内での表現の対応をまとめたものです。各修飾子を使用できるコンテキストと、修飾子の組合せ方法の詳細は、DirectX ピクセル シェーダ 1_x のドキュメントを参照してください。

表 34 ps_1_x の命令セット修飾子

命令 / レジスタ修飾子	Cg の式
instr_x2	**2*x**
instr_x4	**4*x**
instr_d2	**x/2**
instr_sat	**saturate(x)（すなわち min(x, max(x, 1), 0)）**
reg_bias	**x-0.5**
1-**reg**	**1-x**
-reg	**-x**
reg_bx2	**2*(x-0.5)**

言語コンストラクトとサポート

データ型

ps_1_x プロファイルでは、演算は **MaxPixelShaderValue** から **MaxPixelShaderValue** の範囲で、符号付クランプ浮動小数点値に対して行われます。**MaxPixelShaderValue** は、DirectX の実装によって決定されます。このプロファイルではすべてのデータ型を使用できますが、演算はすべてこの範囲内で実行されます。

詳細は、DirectX ピクセル シェーダ 1_x のドキュメントを参照してください。

ステートメントと演算子

DirectX ピクセル シェーダ 1_x プロファイルでは、次の例外を除いて Cg 言語のすべてのコンストラクトがサポートされます。

- 任意のスウィズルはサポートされていません（任意の書込みマスクはサポート）。許可されているスウィズルは、次のとおりです。
  .x/.r .y/.g .z/.b .w/.a
  .xy/.rg .xyz/.rgb .xyzw/.rgba
  .xxx/.rrr .yyy/.ggg .zzz/.bbb .www/.aaa
  .xxxx/.rrrr .yyyy/.gggg .zzzz/.bbbb .wwww/.aaaa
- 行列のスウィズルはサポートされていません。**<**、**<=**、**>** および **>=** 以外のブール演算はサポートされていません。さらに **<**、**<=**、**>** および **>=** がサポートされるのも、**?:** 演算子での条件としてのみです。
- ビットごとの整数演算子は、サポートされていません。
- **/** がサポートされるのは、除数が 0 以外の定数である場合、または **ps_1_3** で深度出力を計算する場合のみです。
- **%** はサポートされていません。
- 3 項演算子 **?:** は、ブール テスト式がコンパイル時のブール型定数であるか、uniform 型スカラのブール、または値域 [-0.5, 1.0] にある定数値とのスカラ比較である場合（たとえば、**a > 0.5 ? b : c**）にサポートされます。
- **do**、**for**および**while**各ループは、完全にアンロール可能な場合にのみサポートされます。
- 配列、ベクトルおよび行列は、コンパイル時の定数値によって、または完全にアンロール可能なループ内でのインデックス変数によってのみインデクシングできます。
- **discard** ステートメントはサポートされていません。これと類似の、ただし一般性の低い **clip()** 関数はサポートされます。
- 入力または出力の **struct** に対する **allocation-rule-identifier** の使用はオプションです。

標準ライブラリ関数

DirectX ピクセル シェーダ 1_x プロファイルには機能上の制限があるため、Cg 標準ライブラリ関数のすべてはサポートされません。表 35 は、このプロファイルでサポートされる Cg 標準ライブラリ関数のリストです。これらの関数の説明は、標準ライブラリのドキュメントを参照してください。

表 35 サポートされる標準ライブラリ関数

dot(floatN, floatN)
lerp(floatN, floatN, floatN)
lerp(floatN, floatN, float)
tex1D(sampler1D, float)
tex1D(sampler1D, float2)
tex1Dproj(sampler1D, float2)
tex1Dproj(sampler1D, float3)
tex2D(sampler2D, float2)
tex2D(sampler2D, float3)
tex2Dproj(sampler2D, float3)
tex2Dproj(sampler2D, float4)
texRECT(samplerRECT, float2)
texRECT(samplerRECT, float3)
texRECTproj(samplerRECT, float3)
texRECTproj(samplerRECT, float4)
tex3D(sampler3D, float3)
tex3Dproj(sampler3D, float4)
texCUBE(samplerCUBE, float3)
texCUBEproj(samplerCUBE, float4)

注意： **非射影のテクスチャ ルックアップ関数は、実際にはハードウェア上で射影ルックアップとして実行されます。そのため、アプリケーションまたは頂点プログラムからこれらの関数に渡されるテクスチャ座標の w 成分は、値 1 を含む必要があります。**

射影テクスチャ ルックアップ関数のテクスチャ座標パラメータには、生成されるテクスチャ処理命令によって行われるスウィズルと一致するスウィズルを指定する必要があります。これは煩雑なようですが、**ps_1_x** プロファイルのプログラムが他のピクセル シェーダ プロファイルでも正常に動作するための措置です。

表 36 は、射影テクスチャ ルックアップ関数のテクスチャ座標パラメータで必要なスウィズルのリストです。

表 36 射影テクスチャ ルックアップで必要なスウィズル

テクスチャ ルックアップ関数	テクスチャ座標のスウィズル
tex1Dproj	.xw/.ra
tex2Dproj	.xyw/.rga
texRECTproj	.xyw/.rga
tex3Dproj	.xyzw/.rgba
texCUBEproj	.xyz/.rgba

バインディング

バインディングの手動割当て

Cg コンパイラは、テクスチャ ユニットと、uniform 型サンプラ パラメータ / テクスチャ座標入力との間のバインディングを自動的に判断できます。この自動割当ては、uniform 型サンプラ パラメータとテクスチャ座標入力が併用されるコンテキストに基づいて行われます。

テクスチャ ユニットと uniform 型パラメータ / テクスチャ座標の間のバインディングを、アプリケーションと一致するよう指定するためには、プログラムで使用する uniform 型サンプラ パラメータとテクスチャ座標入力のバインディング セマンティクスが、すべて一致している必要があります。つまり、**TEXUNIT<*n*>** は **TEXCOORD<*n*>** とのみ併用できます。

部分的に指定されたバインディング / セマンティクスは、すべての場合に動作するとはかぎりません。基本的にこの制限は、DirectX ピクセル シェーダ 1_x のテクスチャ サンプラとテクスチャ座標が密接に結び付いていることが原因です。

uniform 型データのバインディング セマンティクス

uniform 型パラメータのバインディング セマンティクスが指定されない場合、コンパイラが自動的にバインディング セマンティクスを割り当てます。uniform 型スカラ パラメータは、Cg プログラム内での使用方法に応じて、定数レジスタの **xyz** 部分または **w** 部分に割り当てられます。コンパイラの出力を、Cg ランタイムなしに使用する場合は、**x** 成分だけでなく uniform 型スカラのすべての値を必要なスカラ値に設定する必要があります。

表 37 は、**ps_1_x** プロファイルにおける uniform 型パラメータで有効なバインディング セマンティクスをまとめたものです。

表 37 ps_1_x の uniform 型入力バインディング セマンティクス

バインディング セマンティクス名	対応するデータ
register(s0)-register(s3) **TEXUNIT0-TEXTUNIT3**	**テクスチャ ユニット *N*。*N* の値域は** [0..3]。**sampler* 型の uniform 型入力でのみ使用可能。**
register(c0)-register(c7) **C0-C7**	**定数レジスタ** [0..7]。

varying 型入力 / 出力データのバインディング セマンティクス

ps_1_xプロファイルのvarying型入力バインディング セマンティクスは、**vs_1_1** プロファイルの varying 型出力バインディング セマンティクスと同じです。

ps_1_x プロファイルでの varying 型入力バインディング セマンティクスは、**COLOR0**、**COLOR1**、**TEXCOORD0**、**TEXCOORD1**、**TEXCOORD2** および **TEXCOORD3** から構成されています。これらは、DirectX 頂点シェーダにおいて出力レジスタにマッピングされます。

表 38 は、**ps_1_x** プロファイルにおける varying 型入力パラメータで有効なバインディング セマンティクスをまとめたものです。

表 38 ps_1_x の varying 型入力バインディング セマンティクス

バインディング セマンティクス名	対応するデータ
COLOR、COLOR0 COL、COL0	入力カラー値 v0
COLOR1 COL1	入力カラー値 v1
TEXCOORD0-TEXCOORD3 TEX0-TEX3	入力テクスチャ座標 t0-t3

さらに、**ps_1_x** プロファイルでは、**POSITION**、**FOG**、**PSIZE**、**TEXCOORD4**、**TEXCOORD5**、**TEXCOORD6** および **TEXCOORD7** を varying 型入力に指定できます。ただしこれは、これらの入力が参照されていない場合に限ります。このため Cg プログラムでは、**vs_1_1** プロファイル プログラムの varying 型出力と、**ps_1_x** プロファイル プログラムの varying 型入力を、同じ構造体で指定することが可能です。

表 39 は、**ps_1_x** プロファイルにおける varying 型出力パラメータで有効なバインディング セマンティクスをまとめたものです。

表 39 ps_1_x の varying 型出力バインディング セマンティクス

バインディング セマンティクス名	対応するデータ
COLOR、COLOR0 COL、COL0	出力カラー（float4）
DEPTH DEPR	出力深度（float）

出力深度値は、**ps_1_3** プロファイルでのみ割当てが可能という点で特殊であり、次のような形をとる必要があります。

```
...
float4 t = <texture addressing operation>;
float z = dot(texCoord<n>, t.xyz);
float w = dot(texCoord<n+1>, t.xyz);
depth = z / w;
...
```

補助テクスチャ関数

DirectX ピクセル シェーダ 1_x ではテクスチャ処理命令の機能が制限されているため、これらのプロファイルでは、より複雑なテクスチャ処理命令の機能を持つ補助関数のセットが用意されています。これは、ps_1_x の Cg プログラムを記述する際の便宜に過ぎません。各関数の拡張形式を直接記述しても、同じ結果を得ることができます。拡張形式を使用すると、他のプロファイルでもサポートされるという利点もあります。

表 40 は、これらの関数をまとめたものです。

表 40 ps_1_x の補助テクスチャ関数

テクスチャ関数 説明
`offsettex2D(uniform sampler2D tex, float2 st, float4 prevlookup, uniform float4 m)` `offsettexRECT(uniform samplerRECT tex, float2 st, float4 prevlookup, uniform float4 m)`

次の処理を実行します。
`float2 newst = st + m.xy * prevlookup.xx + m.zw * prevlookup.yy;`
`return tex2D/RECT(tex, newst);`
ここで
`st` はサンプラ `tex` に関連付けられたテクスチャ座標です。
`prevlookup` は以前のテクスチャ演算の結果です。
`m` は 2-D バンプ環境マッピング行列です。
すべての `ps_1_x` プロファイルで、この関数を使用して `texbem` 命令を生成できます。

表 40 ps_1_x の補助テクスチャ関数 （続き）

テクスチャ関数 説明

offsettex2DScaleBias(uniform sampler2D tex, float2 st,
float4 prevlookup, uniform float4 m,
uniform float scale, uniform float bias)
offsettexRECTScaleBias(uniform samplerRECT tex, float2 st,
float4 prevlookup, uniform float4 m,
uniform float scale, uniform float bias)
 	
次の処理を実行します。
float2 newst = st + m.xy * prevlookup.xx + m.zw * prevlookup.yy;
float4 result = tex2D/RECT(tex, newst);
return result * saturate(prevlookup.z * scale + bias);
ここで
st はサンプラ tex に関連付けられたテクスチャ座標です。
prevlookup は以前のテクスチャ演算の結果です。
m は 2-D バンプ環境マッピング行列です。
scale は 2-D バンプ環境マッピングのスケール係数です。
bias は 2-D バンプ環境マッピングのオフセットです。
すべての ps_1_x プロファイルで、この関数を使用して texbem1 命令を生成できます。

tex1D_dp3(sampler1D tex, float3 str, float4 prevlookup) 次の処理を実行します。 return tex1D(tex, dot(str, prevlookup.xyz)); ここで str はサンプラ tex に関連付けられたテクスチャ座標です。 prevlookup は以前のテクスチャ演算の結果です。 ps_1_2 および ps_1_3 プロファイルで、この関数を使用して texdp3tex 命令を生成できます。

表 40 ps_1_x の補助テクスチャ関数 （続き）

テクスチャ関数 説明

tex2D_dp3x2(uniform sampler2D tex, float3 str,
float4 intermediate_coord, float4 prevlookup)
texRECT_dp3x2(uniform samplerRECT tex, float3 str,
float4 intermediate_coord, float4 prevlookup)
次の処理を実行します。
float2 newst = float2(dot(intermediate_coord.xyz, prevlookup.xyz),
dot(str, prevlookup.xyz));
return tex2D/RECT(tex, newst);
ここで
str はサンプラ tex に関連付けられたテクスチャ座標です。
prevlookup は以前のテクスチャ演算の結果です。
intermediate_coord は以前のテクスチャ ユニットに関連付けられたテクスチャ座標です。
すべての ps_1_x プロファイルで、この関数を使用して texm3x2pad/texm3x2tex 命令の組合せを生成できます。

tex3D_dp3x3(sampler3D tex, float3 str,
float4 intermediate_coord1,
float4 intermediate_coord2, float4 prevlookup)
texCUBE_dp3x3(samplerCUBE tex, float3 str,
float4 intermediate_coord1,
float4 intermediate_coord2, float4 prevlookup)
次の処理を実行します。
float3 newst = float3(dot(intermediate_coord1.xyz, prevlookup.xyz),
dot(intermediate_coord2.xyz, prevlookup.xyz),
dot(str, prevlookup.xyz));
return tex3D/CUBE(tex, newst);
ここで
str はサンプラ tex に関連付けられたテクスチャ座標です。
prevlookup は以前のテクスチャ演算の結果です。
intermediate_coord1 は n-2 テクスチャ ユニットに関連付けられたテクスチャ座標です。
intermediate_coord2 は n-1 テクスチャ ユニットに関連付けられたテクスチャ座標です。
すべての ps_1_x プロファイルで、この関数を使用して texm3x3pad/ texm3x3pad/texm3x3tex 命令の組合せを生成できます。

表 40 ps_1_x の補助テクスチャ関数 （続き）

テクスチャ関数 説明
texCUBE_reflect_dp3x3(uniform samplerCUBE tex, float4 strq, float4 intermediate_coord1, float4 intermediate_coord2, float4 prevlookup)

次の処理を実行します。
float3 E = float3(intermediate_coord2.w, intermediate_coord1.w,
strq.w);
float3 N = float3(dot(intermediate_coord1.xyz, prevlookup.xyz),
dot(intermediate_coord2.xyz, prevlookup.xyz),
dot(strq.xyz, prevlookup.xyz));
return texCUBE(tex, 2 * dot(N, E) / dot(N, N) * N - E);
ここで
strq はサンプラ tex に関連付けられたテクスチャ座標です。
prevlookup は以前のテクスチャ演算の結果です。
intermediate_coord1 は n-2 テクスチャ ユニットに関連付けられたテクスチャ座標です。
intermediate_coord2 は n-1 テクスチャ ユニットに関連付けられたテクスチャ座標です。
すべての ps_1_x プロファイルで、この関数を使用して texm3x3pad/texm3x3pad/texm3x3vspec 命令の組合せを生成できます。

表 40 ps_1_x の補助テクスチャ関数 （続き）

テクスチャ関数 説明

texCUBE_reflect_eye_dp3x3(uniform samplerCUBE tex,
float3 str, float4 intermediate_coord1,
float4 intermediate_coord2,
float4 prevlookup, uniform float3 eye)
 	
次の処理を実行します。
float3 N = float3(dot(intermediate_coord1.xyz, prevlookup.xyz),
dot(intermediate_coord2.xyz, prevlookup.xyz),
dot(coords.xyz, prevlookup.xyz));
return texCUBE(tex, 2 * dot(N, E) / dot(N, N) * N - E);
ここで
strq はサンプラ tex に関連付けられたテクスチャ座標です。
prevlookup は以前のテクスチャ演算の結果です。
intermediate_coord1 は n-2 テクスチャ ユニットに関連付けられたテクスチャ座標です。
intermediate_coord2 は n-1 テクスチャ ユニットに関連付けられたテクスチャ座標です。
eye はアイレイ ベクトルです。
すべての ps_1_x プロファイルで、この関数を使用して texm3x3pad/texm3x3pad/texm3x3spec 命令の組合せを生成できます。

tex_dp3x2_depth(float3 str, float4 intermediate_coord,
float4 prevlookup)
 	
次の処理を実行します。
float z = dot(intermediate_coord.xyz, prevlookup.xyz);
float w = dot(str, prevlookup.xyz);
return z / w;
ここで
str は *n* 番目のテクスチャ ユニットに関連付けられたテクスチャ座標です。
intermediate_coord は n-1 テクスチャ ユニットに関連付けられたテクスチャ座標です。
prevlookup は以前のテクスチャ演算の結果です。
ps_1_3 プロファイルでは、この関数と varyring 型出力セマンティクス DEPTH を使用して、texm3x2pad/texm3x2depth 命令の組合せを生成できます。

例

次の例では、開発者が Cg を使用して DirectX ピクセル シェーダ 1_x 機能を実現する方法を示しています。

例 1

```
struct VertexOut {
  float4 color     : COLOR0;
  float4 texCoord0 : TEXCOORD0;
  float4 texCoord1 : TEXCOORD1;
};
float4 main(VertexOut IN,
            uniform sampler2D diffuseMap,
            uniform sampler2D normalMap) : COLOR
{
  float4 diffuseTexColor = tex2D(diffuseMap, IN.texCoord0.xy);
  float4 normal = 2 * (tex2D(normalMap, IN.texCoord1.xy)-0.5);
  float3 light_vector = 2 * (IN.color.rgb - 0.5);
  float4 dot_result = saturate(dot(light_vector,
                               normal.xyz).xxxx);
  return dot_result * diffuseTexColor;
}
```

例 2

```
struct VertexOut {
    float4 texCoord0 : TEXCOORD0;
    float4 texCoord1 : TEXCOORD1;
    float4 texCoord2 : TEXCOORD2;
    float4 texCoord3 : TEXCOORD3;
};

float4 main(VertexOut IN,
            uniform sampler2D normalMap,
            uniform sampler2D intensityMap,
            uniform sampler2D colorMap) : COLOR
{
  float4 normal = 2 * (tex2D(normalMap, IN.texCoord0.xy)-0.5);
  float2 intensCoord = float2(
                         dot(IN.texCoord1.xyz, normal.xyz),
                         dot(IN.texCoord2.xyz, normal.xyz));
  float4 intensity = tex2D(intensityMap, intensCoord);
  float4 color = tex2D(colorMap, IN.texCoord3.xy);
  return color * intensity;
}
```

OpenGL NV_vertex_program 1.0 プロファイル（vp20）

vp20 頂点プログラム プロファイルは、OpenGL の **NV_vertex_program** 拡張で使用するために、Cg ソース コードを頂点プログラムにコンパイルする際に使用されます。[7]

- **プロファイル名： vp20**
- **起動方法：** コンパイラ オプション **-profile vp20** を使用。

この節では、**vp20** プロファイルを使用する場合の Cg の機能と制限について説明します。

概要

vp20 プロファイルは、**NV_vertex_program** 拡張機能と一致するように Cg を制限します。**NV_vertex_program** の機能は DirectX 8 頂点シェーダの機能と同等であるため、プログラマが記述する Cg ソース コードに関する制限事項は、DirectX VS 1.1 シェーダ プロファイルの場合と同じです。[8]

コンパイラ出力の構文を除き、**vp20** 頂点シェーダ プロファイルと DirectX VS 1.1 プロファイルの違いは、**vp20** プロファイルが出力を 2 つ多くサポートしていることのみです。頂点からフラグメントへのコネクタは、追加のフィールド **BFC0**（バックフェース プライマリ カラー用）および **BFC1**（バックフェース セカンダリ カラー用）を持つことができます。

位置座標の不変性

- **vp20** プロファイルでは、コア言語仕様に記載されているように、位置座標の不変性がサポートされます。
- モデルビュー射影行列は、**_GL_MVP**のバインディング セマンティクスを使用して指定する必要があります。

7. NV_vertex_program と、vp20 プロファイルを使用してコンパイラが生成するコードを理解するには、GL_NV_vertex_program 拡張機能のドキュメントを参照してください。
8. このプロファイルでサポートされるデータ型、ステートメントおよび演算子の詳細な説明は、222 ページの「DirectX 頂点シェーダ 1.1 プロファイル（vs_1_1）」を参照してください。

データ型

このプロファイルでは、データ型の実装は次のとおりです。

- **float** データ型は、IEEE 32 ビットの単精度として実装されます。
- **half** および **double** データ型は、**float** として実装されます。
- **int** データ型は、浮動小数点の操作を利用してサポートされ、このとき浮動小数点型から除算、モジュロおよびキャストの適切な切捨てを行うための命令が追加されます。
- **fixed** または **sampler*** データ型はサポートされませんが、コア言語仕様によって各プロファイルではこれらのデータ型に必要な最小限の部分的なサポートが提供されます。つまり、変数に対して演算を実行しないかぎり、これらのデータ型を使用する変数を宣言することは可能です。

バインディング

uniform 型データのバインディング セマンティクス

表 41 は、**vs20** プロファイルにおける uniform 型パラメータで有効なバインディング セマンティクスをまとめたものです。

表 41 vp20 の uniform 型入力バインディング セマンティクス

バインディング セマンティクス名	対応するデータ
register(c0)-register(c95) **C0-C95**	**定数レジスタ** [0..95]。 **エイリアスの c0-c95（小文字）も許可される。** **複数の定数レジスタを必要とする変数（行列など）で使用した場合、セマンティクスは使用される最初のレジスタを指定。**

varying 型入力 / 出力データのバインディング セマンティクス

表 42 は、**vp20** プロファイルにおける varying 型入力パラメータで有効なバインディング セマンティクスをまとめたものです。

TEXCOORD6 および **TEXCOORD7** のかわりに、**TANGENT** および **BINORMAL** も使用可能です。2 場面のバインディング セマンティクス **ATTR0-ATTR15** のセットも使用できます。2 つのセットは、相互にエイリアスとして機能します。

表 42 vp20 の varying 型入力バインディング セマンティクス

バインディング セマンティクス名	対応するデータ
POSITION、ATTR0	入力頂点、汎用属性 0
BLENDWEIGHT、ATTR1	入力頂点ウェイト、汎用属性 1
NORMAL、ATTR2	入力法線、汎用属性 2
COLOR0、DIFFUSE、ATTR3	入力プライマリ カラー、汎用属性 3
COLOR1、SPECULAR、ATTR4	入力セカンダリ カラー、汎用属性 4
TESSFACTOR、FOGCOORD、ATTR5	入力フォグ座標、汎用属性 5
PSIZE、ATTR6	入力ポイント サイズ、汎用属性 6
BLENDINDICES、ATTR7	汎用属性 7
TEXCOORD0-TEXCOORD7、ATTR8-ATTR15	入力テクスチャ座標 (texcoord0-texcoord7)、汎用属性 8-15
TANGENT、ATTR14	汎用属性 14
BINORMAL、ATTR15	汎用属性 15

表 43 は、**vp20** プロファイルにおける varying 型出力パラメータで有効なバインディング セマンティクスをまとめたものです。

これらのバインディング セマンティクスは、**NV_vertex_program** 出力レジスタにマッピングされます。2つのセットは、相互にエイリアスとして機能します。

表 43 vp20 の varying 型出力バインディング セマンティクス

バインディング セマンティクス名	対応するデータ
POSITION、HPOS	出力位置座標
PSIZE、PSIZ	出力ポイント サイズ
FOG、FOGC	出力フォグ座標

表 43 vp20 の varying 型出力バインディング セマンティクス （続き）

バインディング セマンティクス名	対応するデータ
COLOR0、COL0	出力プライマリ カラー
COLOR1、COL1	出力セカンダリ カラー
BCOL0	出力プライマリ カラー
BCOL1	出力背面セカンダリ カラー
TEXCOORD0-TEXCOORD3, TEX0-TEX3	出力テクスチャ座標

このプロファイルでは、varying 型出力構造体のメンバに関するセマンティクスとして、**WPOS** を使用できます。ただしこれは、これらのバインディング セマンティクスに伴うメンバが参照されていない場合に限ります。このため Cg プログラムでは、**vp20** プロファイル プログラムの varying 型出力と、**fp30** プロファイル プログラムの varying 型入力を、同じ構造体で指定することが可能です。

OpenGL NV_texture_shader および NV_register_combiners プロファイル（fp20）

OpenGL の NV_texture_shader および NV_register_combiners プロファイルは、OpenGL 拡張の NV_texture_shader および NV_register_combiners 族のために、Cg のソース コードを **nvparse** テキストフォーマットにコンパイルする際に使用されます。[9]

- **プロファイル名**： **fp20**
- **起動方法**： コンパイラ オプション **-profile fp20** を使用。

この節では、**fp20** プロファイルを使用する場合の Cg の機能と制限について説明します。

概要

fp20 プロファイルでの演算は、テクスチャ シェーダ演算と算術演算の 2 つのカテゴリに分類できます。テクスチャ シェーダ演算はテクスチャ シェーダ命令を生成する演算であり、算術演算はレジスタ結合命令を生成する演算です。

基礎となる命令セットとマシン アーキテクチャにより、このプロファイルでのプログラマビリティは、Cg のコンストラクトで許可されているものに比べて制限されています。そのため、このプロファイルでは、Cg プログラムで可能な操作の内容に制限が加えられています。

制限

これらのプロファイルでの Cg プログラムは、生成できる命令数がテクスチャ シェーダ命令は最大 4 つ、レジスタ結合命令は最大 8 つまでに制限されています。最大数がかなり小さいため、これらのプロファイルで Cg コードを記述している間、ユーザはこの制限を強く意識する必要があります。

fp20 プロファイルでは、テクスチャ シェーダ演算または算術演算をプログラム内でいつ使用できるかということも制限されています。テクスチャ シェーダ演算は、以下の場合にかぎって算術演算の出力に依存します。

- 算術演算が、テクスチャ シェーダ演算に対して有効な入力修飾子である。
- 算術演算が、複合テクスチャ シェーダ演算の一部である（これについてのまとめは、250 ページの「補助テクスチャ関数」を参照）。

9. 基礎となる命令セット、その機能および制限事項の詳細は、NV_texture_shader および NV_register_combiners 拡張に関する OpenGL 拡張のドキュメントを参照してください。

修飾子

特定の単純な算術演算は、テクスチャ シェーダ演算の入力に利用でき、算術演算の入力および出力にも、レジスタ結合命令を生成せずに利用できます。これらの演算を入力修飾子および出力修飾子と呼びます。

レジスタ結合命令を生成するかわりに、算術演算がその適用対象であるアセンブリ命令またはソース レジスタを修飾します。たとえば、

```
z = (x - 0.5 + y) / 2
```

という Cg の式では本来、次のようなレジスタ結合命令が生成されます（ただし **x** は **tex0** 内、**y** は **tex1** 内、**z** は **col0** 内にあるとします）。

```
rgb
  {
    discard = half_bias(tex0.rgb);
    discard = tex1.rgb;
    col0 = sum();
    scale_by_one_half();
  }
alpha
  {
    discard = half_bias(tex0.a);
    discard = tex1.a;
    col0 = sum();
    scale_by_one_half();
  }
```

表 44 は、NV_texture_shader および NV_register_combiners の命令セット修飾子と、Cg プログラム内での表現の対応をまとめたものです。各修飾子を使用できるコンテキストと、修飾子の組合せ方法の詳細は、NV_texture_shader および NV_register_combiners のドキュメントを参照してください。

表 44　NV_texture_shader および NV_register_combiners の命令セット修飾子

命令 / レジスタ修飾子	Cg の式
scale_by_two()	**2*x**
scale_by_four()	**4*x**
scale_by_one_half()	**x/2**
bias_by_negative_one_half()	**x-0.5**
bias_by_negative_one_half_scale_by_two()	**2*(x-0.5)**
unsigned(**reg**)	**saturate(x)** **（すなわち min(x, max(x, 1), 0)）**

表 44 NV_texture_shader および NV_register_combiners の命令セット修飾子 （続き）

命令 / レジスタ修飾子	Cg の式
unsigned_invert(**reg**)	**1-saturate(x)**
half_bias(**reg**)	**x-0.5**
-**reg**	**-x**
expand(**reg**)	**2*(x-0.5)**

言語コンストラクトとサポート

データ型

fp20 プロファイルでは、演算は -1 から 1 の範囲で、符号付クランプ浮動小数点値に対して行われます。このプロファイルではすべてのデータ型を使用できますが、演算はすべてこの範囲内で実行されます。詳細は、NV_texture_shader および NV_register_combiners のドキュメントを参照してください。

ステートメントと演算子

fp20 プロファイルでは、次の例外を除いて Cg 言語のすべてのコンストラクトがサポートされます。

- 任意のスウィズルはサポートされていません（任意の書込みマスクはサポート）。許可されているスウィズルは、次のとおりです。
  .x/.r .y/.g .z/.b .w/.a
  .xy/.rg .xyz/.rgb .xyzw/.rgba
  .xxx/.rrr .yyy/.ggg .zzz/.bbb .www/.aaa
  .xxxx/.rrrr .yyyy/.gggg .zzzz/.bbbb .wwww/.aaaa
- 行列のスウィズルはサポートされていません。
- **<**、**<=**、**>** および **>=** 以外のブール演算はサポートされていません。さらに **<**、**<=**、**>** および **>=** がサポートされるのも、**?:** 演算子での条件としてのみです。
- ビットごとの整数演算子は、サポートされていません。
- **/** がサポートされるのは、除数が 0 以外の定数である場合、または深度出力を計算する場合のみです。
- **%** はサポートされていません。
- 3 項演算子 **?:** は、ブール テスト式がコンパイル時のブール型定数であるか、uniform 型スカラのブール、または値域 [-0.5, 1.0] にある低数値とのスカラ比較である場合（たとえば、**a > 0.5 ? b : c**）にサポートされます。
- **do**、**for** および **while** 各ループは、完全にアンロール可能な場合にのみサポートされます。

- 配列、ベクトルおよび行列は、コンパイル時の定数値によって、または完全にアンロール可能なループ内でのインデックス変数によってのみインデクシングできます。
- **discard** ステートメントはサポートされていません。これと類似の、ただし一般性の低い **clip()** 関数はサポートされます。
- 入力または出力の **struct** に対する **allocation-rule-identifier** の使用はオプションです。

標準ライブラリ関数

fp20 プロファイルは機能が制限されているため、Cg の標準ライブラリ関数のすべてはサポートされません。

表 45 は、このプロファイルでサポートされている Cg 標準ライブラリ関数のリストです。これらの関数の説明は、標準ライブラリのドキュメントを参照してください。

表 45 サポートされる標準ライブラリ関数

dot(floatN, floatN)
lerp(floatN, floatN, floatN)
lerp(floatN, floatN, float)
tex1D(sampler1D, float)
tex1D(sampler1D, float2)
tex1Dproj(sampler1D, float2)
tex1Dproj(sampler1D, float3)
tex2D(sampler2D, float2)
tex2D(sampler2D, float3)
tex2Dproj(sampler2D, float3)
tex2Dproj(sampler2D, float4)
texRECT(samplerRECT, float2)
texRECT(samplerRECT, float3)
texRECTproj(samplerRECT, float3)
texRECTproj(samplerRECT, float4)
tex3D(sampler3D, float3)
tex3Dproj(sampler3D, float4)

表 45 サポートされる標準ライブラリ関数 （続き）

texCUBE(samplerCUBE, float3)
texCUBEproj(samplerCUBE, float4)

注意： 非射影のテクスチャ ルックアップ関数は、実際にはハードウェア上で射影ルックアップとして実行されます。そのため、アプリケーションまたは頂点プログラムからこれらの関数に渡されるテクスチャ座標の w 成分は、値 1 を含む必要があります。

射影テクスチャ ルックアップ関数のテクスチャ座標パラメータには、生成されるテクスチャ シェーダ命令によって行われるスウィズルと一致するスウィズルを指定する必要があります。これは煩雑なようですが、**fp20** プロファイルのプログラムが他のピクセル シェーダ プロファイルでも正常に動作するための措置です。

表 46 は、射影テクスチャ ルックアップ関数のテクスチャ座標パラメータで必要なスウィズルのリストです。

表 46 射影テクスチャ ルックアップで必要なスウィズル

テクスチャ ルックアップ関数	テクスチャ座標のスウィズル
tex1Dproj	.xw/.ra
tex2Dproj	.xyw/.rga
texRECTproj	.xyw/.rga
tex3Dproj	.xyzw/.rgba
texCUBEproj	.xyz/.rgba

バインディング

バインディングの手動割当て

Cg コンパイラは、テクスチャ ユニットと、uniform 型サンプラ パラメータ / テクスチャ座標入力との間のバインディングを自動的に判断できます。この自動割当ては、uniform 型サンプラ パラメータとテクスチャ座標入力が併用されるコンテキストに基づいて行われます。

テクスチャ ユニットと uniform 型パラメータ / テクスチャ座標の間のバインディングを、アプリケーションと一致するよう指定するためには、プログラムで使用する uniform 型 sampler パラメータとテクスチャ座標入力のバインディング セマンティクスが、すべて一致している必要があります。たとえば、**TEXUNIT<n>** は **TEXCOORD<n>** とのみ併用できます。部分的に指定されたバインディング / セマンティクスは、すべての場合に動作するとはかぎりません。基本的にこの制限は、NV_texture_shader 拡張のテクスチャ サンプラとテクスチャ座標が密接に結び付いていることが原因です。

uniform 型データのバインディング セマンティクス

uniform 型パラメータのバインディング セマンティクスが指定されない場合、コンパイラが自動的にバインディング セマンティクスを割り当てます。uniform 型スカラ パラメータは、Cg プログラム内での使用方法に応じて、定数レジスタの **xyz** 部分または **w** 部分に割り当てられます。コンパイラの出力を、Cg ランタイムなしに使用する場合は、**x** 成分だけでなく uniform 型スカラのすべての値を必要なスカラ値に設定する必要があります。

表 47 は、**fp20** プロファイルにおける uniform 型パラメータで有効なバインディング セマンティクスをまとめたものです。

表 47 fp20 の uniform 型バインディング セマンティクス

バインディング セマンティクス名	対応するデータ
register(s0)-register(s3) **TEXUNIT0-TEXTUNIT3**	**テクスチャ ユニット *N*。*N* の値域は** [0..3]。**sampler* 型の** uniform **型入力でのみ使用可能。**

ps_1_x プロファイルでは、**C<n>**/**register(c<n>)** バインディング セマンティクスを指定することによって、uniform 型変数がどの定数レジスタに格納されるかをプログラマが決定できます。**NV_register_combiners** 拡張には定数レジスタの単一バンクがないため、これは、**fp20** プロファイルでは許可されていません。**NV_register_combiners** 拡張では定数レジスタについて言及されていませんが、定数レジスタは結合ステージごとであるため、プログラムでそれにバインディングを指定することは、コンパイラを過剰に制限することになります。

varying 型入力 / 出力データのバインディング セマンティクス

fp20 プロファイルの varying 型入力バインディング セマンティクスは、**vp20** プロファイルの varying 型出力バインディング セマンティクスと同じです。

fp20 プロファイルでの varying 型入力バインディング セマンティクスは、**COLOR0**、**COLOR1**、**TEXCOORD0**、**TEXCOORD1**、**TEXCOORD2** および **TEXCOORD3** から構成されています。これらは、頂点シェーダにおいて出力レジスタにマッピングされます。

表 48 は、**fp20** プロファイルにおける varying 型入力パラメータで有効なバインディング セマンティクスをまとめたものです。

表 48 fp20 の varying 型入力バインディング セマンティクス

バインディング セマンティクス名	対応するデータ
COLOR、COLOR0 COL、COL0	入力カラー値（v0）
COLOR1 COL1	入力カラー値 v1
TEXCOORD0-TEXCOORD3 TEX0-TEX3	入力テクスチャ座標（t0-t3）
FOGP FOG	入力フォッグおよび係数

さらに、**fp20** プロファイルでは、varying 型入力に **POSITION**、**PSIZE**、**TEXCOORD4**、**TEXCOORD5**、**TEXCOORD6** および **TEXCOORD7** を指定できます。ただし、これらの入力が参照されていない場合にかぎります。このため Cg プログラムでは、**vp20** プロファイル プログラムの varying 型出力と、**fp20** プロファイル プログラムの varying 型入力を、同じ構造体で指定することが可能です。

表 49 は、**fp20** プロファイルにおける varying 型出力パラメータで有効なバインディング セマンティクスをまとめたものです。

表 49 fp20 の varying 型出力バインディング セマンティクス

バインディング セマンティクス名	対応するデータ
COLOR、COLOR0 COL、COL0	出力カラー（float4）
DEPR DEPTH	出力深度（float4）

出力深度値は、次の形でのみ値を代入できるという点で特殊です。

```
...
float4 t = <texture shader operation>;
float z = dot(texCoord<n>, t.xyz);
float w = dot(texCoord<n+1>, t.xyz);
depth = z / w;
...
```

補助テクスチャ関数

NV_texture_shaderではテクスチャ シェーダ命令の機能が制限されているため、これらのプロファイルでは、より複雑なテクスチャ シェーダ命令の機能を持つ補助関数のセットが用意されています。これは、**fp20** の Cg プログラムを記述する際の便宜に過ぎません。各関数の拡張形式を直接記述しても、同じ結果を得ることができます。拡張形式を使用すると、他のプロファイルでもサポートされるという利点もあります。

表 50 は、これらの関数をまとめたものです。

表 50 fp20 の補助テクスチャ関数

テクスチャ関数 説明
`offsettex2D(uniform sampler2D tex, float2 st, float4 prevlookup, uniform float4 m)` `offsettexRECT(uniform samplerRECT tex, float2 st, float4 prevlookup, uniform float4 m)`

次の処理を実行します。
`float2 newst = st + m.xy * prevlookup.xx + m.zw * prevlookup.yy;`
`return tex2D/RECT(tex, newst);`
ここで
`st` はサンプラ `tex` に関連付けられたテクスチャ座標です。
`prevlookup` は以前のテクスチャ演算の結果です。
`m` はオフセット テクスチャ行列です。
この関数を使用して `offset_2d` または `offset_rectangle` という NV_texture_shader 命令を生成できます。

表 50 fp20 の補助テクスチャ関数 （続き）

テクスチャ関数 説明

`offsettex2DScaleBias(uniform sampler2D tex, float2 st, float4 prevlookup, uniform float4 m, uniform float scale, uniform float bias)`
`offsettexRECTScaleBias(uniform samplerRECT tex, float2 st, float4 prevlookup, uniform float4 m, uniform float scale, uniform float bias)`
 	
次の処理を実行します。
`float2 newst = st + m.xy * prevlookup.xx + m.zw * prevlookup.yy;`
`float4 result = tex2D/RECT(tex, newst);`
`return result * saturate(prevlookup.z * scale + bias);`
ここで
`st` はサンプラ `tex` に関連付けられたテクスチャ座標です。
`prevlookup` は以前のテクスチャ演算の結果です。
`m` はオフセット テクスチャ行列です。
`scale` はオフセット テクスチャ スケールです。
`bias` はオフセット テクスチャ バイアスです。
この関数を使用して `offset_2d_scale` または `offset_rectangle_scale` という NV_texture_shader 命令を生成できます。

`tex1D_dp3(sampler1D tex, float3 str, float4 prevlookup)`
 	
次の処理を実行します。
`return tex1D(tex, dot(str, prevlookup.xyz));`
ここで
`str` はサンプラ `tex` に関連付けられたテクスチャ座標です。
`prevlookup` は以前のテクスチャ演算の結果です。
この関数を使用して `dot_product_1d` という NV_texture_shader 命令を生成できます。

表 50 fp20 の補助テクスチャ関数 （続き）

テクスチャ関数 説明

`tex2D_dp3x2(uniform sampler2D tex, float3 str,`
`float4 intermediate_coord, float4 prevlookup)`
`texRECT_dp3x2(uniform samplerRECT tex, float3 str,`
`float4 intermediate_coord, float4 prevlookup)`
次の処理を実行します。
`float2 newst = float2(dot(intermediate_coord.xyz, prevlookup.xyz),`
`dot(str, prevlookup.xyz));`
`return tex2D/RECT(tex, newst);`
ここで
`str` はサンプラ `tex` に関連付けられたテクスチャ座標です。
`prevlookup` は以前のテクスチャ演算の結果です。
`intermediate_coord` は以前のテクスチャ ユニットに関連付けられたテクスチャ座標です。
この関数を使用して `dot_product_2d` または `dot_product_rectangle` という NV_texture_shader 命令の組合せを生成できます。

`tex3D_dp3x3(sampler3D tex, float3 str,`
`float4 intermediate_coord1,`
`float4 intermediate_coord2, float4 prevlookup)`
`texCUBE_dp3x3(samplerCUBE tex, float3 str,`
`float4 intermediate_coord1,`
`float4 intermediate_coord2, float4 prevlookup)`
次の処理を実行します。
`float3 newst = float3(dot(intermediate_coord1.xyz, prevlookup.xyz),`
`dot(intermediate_coord2.xyz, prevlookup.xyz),`
`dot(str, prevlookup.xyz));`
`return tex3D/CUBE(tex, newst);`
ここで
`str` はサンプラ `tex` に関連付けられたテクスチャ座標です。
`prevlookup` は以前のテクスチャ演算の結果です。
`intermediate_coord1` は `n-2` テクスチャ ユニットに関連付けられたテクスチャ座標です。
`intermediate_coord2` は `n-1` テクスチャ ユニットに関連付けられたテクスチャ座標です。
この関数を使用して `dot_product_3d` または `dot_product_cube_map` という NV_texture_shader 命令の組合せを生成できます。

表 50 fp20 の補助テクスチャ関数 （続き）

テクスチャ関数	説明

`texCUBE_reflect_dp3x3(uniform samplerCUBE tex, float4 strq, float4 intermediate_coord1, float4 intermediate_coord2, float4 prevlookup)`	次の処理を実行します。
	`float3 E = float3(intermediate_coord2.w, intermediate_coord1.w, strq.w);`
	`float3 N = float3(dot(intermediate_coord1.xyz, prevlookup.xyz), dot(intermediate_coord2.xyz, prevlookup.xyz), dot(strq.xyz, prevlookup.xyz));`
	`return texCUBE(tex, 2 * dot(N, E) / dot(N, N) * N - E);`
	ここで
	`strq` はサンプラ `tex` に関連付けられたテクスチャ座標です。
	`prevlookup` は以前のテクスチャ演算の結果です。
	`intermediate_coord1` は `n-2` テクスチャ ユニットに関連付けられたテクスチャ座標です。
	`intermediate_coord2` は `n-1` テクスチャ ユニットに関連付けられたテクスチャ座標です。
	この関数を使用して `dot_product_reflect_cube_map_eye_from_qs` という NV_texture_shader 命令の組合せを生成できます。

表 50 fp20 の補助テクスチャ関数 （続き）

テクスチャ関数 説明

texCUBE_reflect_eye_dp3x3(uniform samplerCUBE tex,
float3 str,
float4 intermediate_coord1,
float4 intermediate_coord2,
float4 prevlookup,
uniform float3 eye)
 	
次の処理を実行します。
float3 N = float3(dot(intermediate_coord1.xyz, prevlookup.xyz),
dot(intermediate_coord2.xyz, prevlookup.xyz),
dot(coords.xyz, prevlookup.xyz));
return texCUBE(tex, 2 * dot(N, E) / dot(N, N) * N - E);
ここで
strq はサンプラ tex に関連付けられたテクスチャ座標です。
prevlookup は以前のテクスチャ演算の結果です。
intermediate_coord1 は n-2 テクスチャ ユニットに関連付けられたテクスチャ座標です。
intermediate_coord2 は n-1 テクスチャ ユニットに関連付けられたテクスチャ座標です。
eye はアイレイ ベクトルです。
この関数を使用して dot_product_reflect_cube_map_const_eye という NV_texture_shader 命令の組合せを生成できます。

tex_dp3x2_depth(float3 str, float4 intermediate_coord,
float4 prevlookup)
 	
次の処理を実行します。
float z = dot(intermediate_coord.xyz, prevlookup.xyz);
float w = dot(str, prevlookup.xyz);
return z / w;
ここで
str は *n* 番目のテクスチャ ユニットに関連付けられたテクスチャ座標です。
intermediate_coord は n-1 テクスチャ ユニットに関連付けられたテクスチャ座標です。
prevlookup は以前のテクスチャ演算の結果です。
この関数と varying 型出力セマンティクス DEPTH を使用して、dot_product_depth_replace という NV_texture_shader 命令の組合せを生成できます。

例

次の例では、開発者が Cg を使用して NV_texture_shader および NV_register_combiners の機能を実現する方法を示しています。

例 1

```
struct VertexOut {
    float4 color     : COLOR0;
    float4 texCoord0 : TEXCOORD0;
    float4 texCoord1 : TEXCOORD1;
};

float4 main(VertexOut IN,
            uniform sampler2D diffuseMap,
            uniform sampler2D normalMap) : COLOR
{
  float4 diffuseTexColor = tex2D(diffuseMap, IN.texCoord0.xy);
  float4 normal = 2 * (tex2D(normalMap, IN.texCoord1.xy)-0.5);
  float3 light_vector = 2 * (IN.color.rgb - 0.5);
  float4 dot_result = saturate(
                        dot(light_vector, normal.xyz).xxxx);
  return dot_result * diffuseTexColor;
}
```

例 2

```
struct VertexOut {
    float4 texCoord0 : TEXCOORD0;
    float4 texCoord1 : TEXCOORD1;
    float4 texCoord2 : TEXCOORD2;
    float4 texCoord3 : TEXCOORD3;
};

float4 main(VertexOut IN,
            uniform sampler2D normalMap,
            uniform sampler2D intensityMap,
            uniform sampler2D colorMap) : COLOR
{
  float4 normal = 2 * (tex2D(normalMap, IN.texCoord0.xy)-0.5);
  float2 intensCoord = float2(
                        dot(IN.texCoord1.xyz, normal.xyz),
                        dot(IN.texCoord2.xyz, normal.xyz));
  float4 intensity = tex2D(intensityMap, intensCoord);
  float4 color = tex2D(colorMap, IN.texCoord3.xy);
  return color * intensity;
}
```

付録 C
Cg のパフォーマンスを向上させる 9 つのステップ

効率的なプログラムにコンパイルされる Cg コードを記述するためには、C や C++、Java の効率的なプログラムとは異なるテクニックとアプローチが必要です。基本的な訓練の一部（効果的なアルゴリズムの使用など）は同じでも、現在の GPU のハードウェア プログラミング モデルは、CPU のプログラミング モデルとは大幅に異なります。このことが、作成したシェーダで期待どおりのパフォーマンスを得られないという落とし穴につながることもありますが、それは同時に、慎重なプログラミングによって GPU を限界まで効率化するチャンスでもあるのです。

Cg 言語では、GPU ハードウェアの低レベルな作業のほとんどが隠蔽されているため、GPU レベルでの命令セットを用いるよりより高度なレベルでシェーダに専念することができます。しかし、C や C++ のコードを書くときに現在のコンピュータ アーキテクチャ（キャッシュやメモリ階層の問題など）を理解することが重要であるのと同様に、Cg のコードを書く場合にも GPU に関する多少の理解が役に立ちます。ここでは、Cg で記述され、NVIDIA の GeForce FX アーキテクチャ上（具体的には **vp30**、**fp30**、**arbfp1**、**ps_2_0**、**ps_2_x**、**vs_2_0** および **vs_2_x** の各プロファイル）で動作する頂点プログラムとフラグメント プログラムのパフォーマンスを最大限に向上させるテクニックを説明しますが、原則の多くはより広範囲に適用が可能です。

1. ベクトル化のためのプログラム

GPU では通常、1 つの演算を実行するのと同じ速度で 4 つの算術演算を実行できます。したがって、4 つの浮動小数点値を持つ 2 つのベクトル、

```
float4 a, b;
```

がある場合、この 2 つのベクトルを次のように加算しても、

```
float4 c = a+b;
```

次のように、その 2 つの要素を加算するよりも計算の負荷が高くなることはありません。

```
float d = a.x + b.x;
```

これは、効率的なプログラミングにとって 2 つの意味があります。まず、これらのベクトル演算に自然にマッピングされるコードを書く必要があるということです。**float4** の 2 つの変数を加算する場合、

```
float4 c = float4(a.x + b.x, a.x + b.y, a.z + b.z,
                  a.w + b.w);
```

と書くより、次のようなコードの方が効率的です。

```
float4 c = a+b;
```

コンパイラは、可能なかぎりプログラム内でベクトル化の検出を試行しますが、元々のコードでのベクトル化の程度が高いほど、コンパイラの負担は少なくなります。

もっと具体的な例を、接空間バンプ マッピングについて行われる一般的な計算で見てみます。タンジェント方向のオフセットを **x** に、従法線方向のオフセットを **y** に、法線方向のオフセットを **z** にそれぞれ格納することによってバンプをエンコードするテクスチャ マップがあるとすると、バンプマッピング後の法線は、タンジェント、従法線および法線を適切にスケーリングすることで計算されます。C または C++ であれば、この計算は次のように書くのが一般的でしょう。

```
// Tangent, binormal, normal.  Passed in from vertex program.
Float3 T, B, N;
Float3 Nbump;    // Bump-mapped normal
Float3 bump = tex2D(bumpSampler, uv);
Nbump.x = bump.x * T.x + bump.y * B.x + bump.z * N.x;
Nbump.y = bump.x * T.y + bump.y * B.y + bump.z * N.y;
Nbump.z = bump.x * T.z + bump.y * B.z + bump.z * N.z;
```

この例では、浮動小数点値のペアを一度に加算および乗算する一連の計算が記述されています。代数学を考えると、次のように **float3** と **float** の乗算を 3 回、**float3** どうしの加算を 2 回行うように書き換えることができて実行速度は元のコードの数倍になります。

```
Nbump = bump.x * T + bump.y * B + bump.z * N;
```

2. ベクトル化を最大限に活用するためにスウィズルを使用する

GPU がベクトル内の値をスウィズルする際、パフォーマンス上の負荷はありません（スウィズルを利用してベクトルの要素を並べ替えられることを思い出してください）。次のようなベクトルがあるとします。

```
float3 a = float3(0, 1, 2);
```

スウィズルによって、次のような新しいベクトルが作られます。

```
a.xxx = float3(0, 0, 0);
a.yzz = float3(1, 2, 2);
a.zy  = float2(2, 1);
```

これがスウィズルの例です。データを慎重にスウィズルすれば、計算の両辺にある 2 つのベクトルの同じ成分を使わない場合でも、ベクトル化の利点を利用できます。たとえば、外積の計算を考えてみましょう。2 つの 3 次元ベクトルがあり、その外積はこの 2 つのベクトルに直交する新たなベクトルを返します。これは次のように計算されます。

```
float3 a, b;
float3 c = float3(a.y*b.z - a.z*b.y, a.z*b.x - a.x*b.z,
                  a.x*b.y - a.y*b.x);
```

この例でもやはり、**float** 値のペアに関する算術演算が何度も繰り返されています。これを、ベクトル化演算で単純化してみます。次に示すのは、Cg 標準ライブラリにある **cross()** 関数の実装ですが、2 回のベクトル乗算と 1 回のベクトル減算だけになっています。

```
float3 cross(float3 a, float3 b) {
  return  a.yzx * b.zxy - a.zxy * b.yzx;
}
```

外積の計算結果が等しくなることを、自分でも確認してみてください。ベクトル化計算を多くすることによって、GPU の処理効率が向上していることにも注意してください。

3. Cg 標準ライブラリを使用する

Cg 標準ライブラリに用意されている関数は、効率と精度のどちらの点からも綿密に作成されています。標準ライブラリ関数を適宜使用すると、GPU 上での高速なコードへのコンパイルが自動的に保証されるため、作成するシェーダにおける問題の解決に専念することができます。

特に高速な標準ライブラリ関数として、2つのベクトルの内積を計算する **dot()**、変数の絶対値を計算する **abs()**、値を 0 と 1 の間にクランプする **saturate()**、1 組の値の最小と最大を返す **min()** および **max()** などがあります。これらの関数の多くは、GPU のアセンブリ言語命令に直接コンパイルされるため、標準ライブラリで提供されている関数を使うのが最も効率的です。次のような内積関数を独自に作成したとします。

```
float mydot(float3 a, float3 b) {
  return a.x*b.x + a.y*b.y + a.z*b.z;
}
```

このコードをコンパイルすると数個の命令になりますが、組込みの **dot()** 関数をコンパイルすれば、内積に特化した 1 つの命令になります。このような命令を生成するには、標準ライブラリを使うしかありません。

特に注意が必要な 2 つの関数があります。**abs()** 関数は通常、他の命令の実行中に GPU によって計算されるため、頂点プログラムでもフラグメント プログラムでも負荷とはなりません。同様に、**saturate()** 関数もフラグメント プログラ

ムにおいて負荷になりません。これらの関数は、必要であれば積極的に使用してください。

4. 複雑な関数のエンコードにテクスチャ マップを使用する

テクスチャ マップをサポートする各プロファイルでは、テクスチャ マップ ルックアップが非常に効率的です。一定数以上の算術演算が必要な複雑な関数の場合、それをテクスチャ マップにエンコードします。たとえば、作成した関数 **f(*x*,*y*)** がシェーダでボトルネックになっているとします。ここでは、この関数がコールされるとき ***x*** と ***y*** は常に 0 から 1 の間の値をとり、**f(*x*,*y*)** の計算結果も常に 0 から 1 の間であると仮定します。この関数が適度に円滑であり、それほど高い精度での計算が必要ない場合は、アプリケーションで事前計算してテクスチャ マップに計算結果を格納しておくことができます。そのためには、

```
float val = f(x,y);
```

を次のようなコードに置き換えます。

```
float val = tex2D(fSampler, float2(x, y)).x;
```

この手法は、1D テクスチャ マップおよび 3D テクスチャ マップを使用して、1 次元および 3 次元の関数にも適用できます。

また一般的には、関数に渡す値が **[0,1]** の範囲にないこともあり、関数の戻り値が **[0,1]** の範囲にないこともあります。この場合、2 つのユーティリティ関数を使用します。**remapTo01()** は、範囲 **[*low*,*high*]** を **[0,1]** にリマップし、**remapFrom01()** はこの逆を行います。

```
float4 remapTo01(float4 v, float4 low, float4 high) {
  return saturate((v - low)/(high-low));
}

float4 remapFrom01(float4 v, float4 low, float4 high) {
  return lerp(low, high, v);
}
```

ここでは、ベクトル化も忘れないようにします。2 つの **float** 値関数の定義域と値域が同じであれば、1 枚のテクスチャの 2 つの成分にパックできます。1 つのテクスチャ ルックアップだけで両方をロードでき、**remap*()** のベクトル化バージョンを使用して、リマップをさらに効率化することもできます。

5. 必要最小限の精度のデータ型を使用する

複数の精度をサポートする各プロファイルでは、一般的に、**fixed** 精度の変数を使用して計算できる場合には **half** を使用するより高速であり、**half** を使用する方が **float** を使用するより高速です。**half** や **float** が提供する範囲や精度が必要な場合もありますが、必要がなければこれらを使用することは回避すべきです。

6. シェーディング計算には適切な標準ライブラリルーチンを使用する

シェーディング モデル（Lambertian、Blinn、Phong など）を実装する場合、スペキュラ成分の計算には内積ルーチンの実行、負の結果の 0 へのクランプ、値のべき乗などが伴います。このプロセスを高速化できる、いくつかのコツがあります。

- 前述したように、内積の計算には必ず **dot()** 関数を使用します。
- フラグメント プログラムで、内積計算の結果を **[0,1]** の範囲にクランプする必要がある場合は、**max()** 関数ではなく **saturate()** 関数を使用します。これを **max(0,dot(N,L))** と書くことがよくありますが、正規化されたベクトルの内積は 1 を超えることがないので、**N** および **L** ベクトルが正規化されているかぎり、これは **saturate(dot(N,L))** と書くのと等価です。フラグメント プログラムでは **saturate()** による負荷はないため（259 ページの「3.Cg 標準ライブラリを使用する」を参照）、こうすることでコードが効率的になります。
- 可能であれば、標準ライブラリ関数 **lit()** を使用します。**lit()** 関数は、拡散と光沢を持つ Blinn シェーディング モデルを実装します。この関数のパラメータは、次の 3 つです。
  - 正規化された表面法線とライト ベクトルの内積
  - ハーフアングル ベクトルと法線の内積
  - スペキュラ指数

  戻り値は 4 ベクトルですが、このとき、
  - **x** および **w** 成分は常に 1 です。
  - **y** 成分は、ディフューズ内積に等しいか、あるいは内積が 0 未満の場合には 0 です。
  - **z** 成分は、スペキュラ内積に指定の指数をべき乗した値に等しいか、あるいはディフューズ内積が 0 未満の場合には 0 です。

  これらはすべて、等価な演算を Cg コードで記述するより、はるかに効率的に実行されます。

7. 計算頻度の違いを活用する

フラグメント プログラムは、一般的に頂点プログラムより実行回数がはるかに多いということを覚えておいてください。そのため、可能であれば必ず、計算をフラグメント プログラムから頂点プログラムに移行します。前述したように、頂点プログラムからの varying 型出力は、フラグメント プログラムに渡される前に自動的に線形補間されます。

フラグメント プログラムから頂点プログラムへ計算を移行できるのは、主に次の 3 つの場合です。

- 結果がすべてのフラグメントで一定の場合。
  頂点シェーダによって計算される値がすべての頂点について等しいために、補間後にすべてのフラグメントが受け取る値も等しい場合、そのような値のみに基づいてフラグメント シェーダが行う計算は、頂点シェーダに移行できます（ただし、テクスチャ マップ ルックアップなど、フラグメントでのみ可能な演算を必要としない場合に限ります）。
- 結果が三角形全体で線形になる場合。
  フラグメント シェーダによって計算される値が、三角形の表面全体にわたって線形に変化する場合（フラグメントから光源までの距離を減衰に利用する場合など）、その値は各頂点の頂点シェーダで計算してからフラグメント シェーダに渡すことができます。補間は、その途中で GPU によって自動的に行われます。
- 結果が三角形全体でほぼ線形になる場合。
  フラグメント シェーダによって計算される値が三角形全体にわたって徐々に変化する場合、その値を各頂点で計算してから、フラグメント シェーダで線形補間しても、許容範囲の近似値となります。たとえば、通常の Gourand シェーディング アルゴリズムでも、この利点を活用してピクセルごとではなく頂点ごとにライティングを計算しています。

同様に、頂点シェーダの計算が uniform 型パラメータの値にのみ依存するのであれば、それを CPU に移行して、計算の結果を別の uniform 型パラメータで頂点シェーダに渡すという活用も可能です。たとえば頂点シェーダが、離れた光源の方向を指定する **`float3`** ベクトルに渡される場合、このベクトルを CPU 上で正規化してから頂点シェーダに渡します。こうすると、頂点シェーダで **`normalize(lightvector)`** を何度も再計算する必要がなくなります。

8. 乗算のためだけの行列転置は避ける

行列の転置の計算は、多くの場合回避できます。転置した **float3x3** 行列 ***m*** に **float3** の **v** を乗算する場合であれば、

```
mul(v, m);
```

とする方が、等価でありながら次の式より効率的です。

```
mul(transpose(m), v);
```

9. フラグメント プログラムでは条件コードを最小限にする

GPU は現在、フラグメント プログラムでの分岐をサポートしていません。条件的に実行されるコード（**if/else** 式など）の行数が多いプログラムは、条件にかかわらずすべてを実行したのと同程度の実行速度になる傾向があります。したがって、条件コードの行数が多く、かつそれを CPU で評価できるのであれば、シェーダ ソース コードを複数バージョン作成して、その 1 つを実行時に適切なコード パスにバインドする方が有利です。

この状況の一例が、汎用の光源モデルのシェーディングをサポートするフラグメント シェーダです。パラメータの設定方法に応じて、ポイント ライト、スポットライト、あるいはテクスチャ マップにより光の寄与を決定する光源が実装されます。使用するライティング モデルの決定に一連の **if/else** によるテストを作成するのではなく、光源ごとに異なるバージョンのシェーダを用意する方が、通常は効率的です。

付録 D
Cg コンパイラ オプション

この付録では、Cg コンパイラのコマンドライン オプションについて説明します。次に、Cg コンパイラ **cgc.exe** のコマンドライン オプションを示します。

- **-profile *prof***
  prof プロファイルに対してコンパイルします。
- **-profileopts *profopts***
  プロファイル固有オプションのカンマ区切りのリストを指定します。有効なオプションについては、プロファイルの仕様を参照してください。
- **-entry *fname***
  main 関数名を ***fname*** として指定します。
- **-o *fname***
  ファイル ***fname*** に出力を書き込みます。
- **-D*macro*[=*value*]**
  macro を定義します。オプションは ***value*** で指定します。
- **-I*pathname***
  インクルード ディレクトリのパスを指定します。
- **-l *filename***
  コンパイラ メッセージを標準出力ではなく ***filename*** に書き込みます。
- **-strict**
  厳密な型チェックを強制的に実行させます。
- **-nofx**
  CgFX のキーワードを予約語として扱わないようにします。
- **-quiet**
  stdout へのヘッダーの出力を抑止します。
- **-nocode**
  コンパイルしますが、コードは生成しません。
- **-nostdlib**
  コンパイルの前に **stdlib.h** ヘッダー ファイルをインクルードしません。

- **-longprogs**
  プロファイルの制限よりも長いコードの生成を許可します。
- **-debug**
  debug() 関数を有効にします。
- **-v**
  コンパイラのバージョンを **stdout** に出力します。
- **-h**
  短いヘルプ メッセージを出力します。
- **-maxunrollcount *N***
  ループの最大アンロール回数を ***N*** に設定します。***N*** 回を超えて反復するループは、アンロールされません。デフォルトは、**256** です。
- **-posinv**
  位置座標の不変性が現在のプロファイルでサポートされている場合に、位置固定の頂点プログラムを生成します。

付録 E
カラー図

この付録には、このマニュアル全体を通して次のシェーダ例で提供されている図のカラー バージョンを記載しています。各カラー図の下の参照ページは、対応するコードの記載ページを示しています。

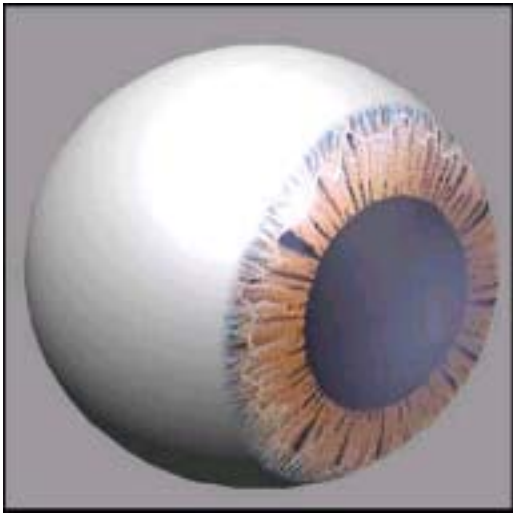
- 「拡張プロファイルのサンプル シェーダ」の図
  - 268 ページの「改善された水面の例」
  - 268 ページの「融解ペイントの例」
  - 268 ページの「マルチペイントの例」
  - 268 ページの「レイ トレーシングによる屈折の例」
  - 269 ページの「皮膚の例」
  - 269 ページの「薄膜エフェクトの例」
  - 269 ページの「カー ペイント 9 の例」
- 「基本プロファイルのサンプル シェーダ」の図
  - 269 ページの「異方性ライティングの例」
  - 270 ページの「バンプ dot3x2 ディフューズおよびスペキュラの例」
  - 270 ページの「バンプ反射マッピングの例」
  - 270 ページの「フレネルの例」
  - 270 ページの「草の例」
  - 271 ページの「屈折の例」
  - 271 ページの「シャドウ ボリューム掃引の例」
  - 271 ページの「サイン波の例」
  - 271 ページの「行列パレット スキニングの例」

改善された水面の例
（101 ページの「改善された水面」を参照）

融解ペイントの例
（105 ページの「融解ペイント」を参照）

図 23 カラー図 - 改善された水面と融解ペイント

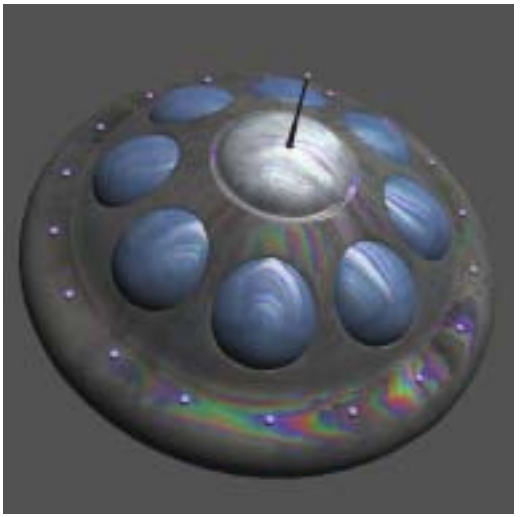
マルチペイントの例
（109 ページの「マルチペイント」を参照）



レイ トレーシングによる屈折の例
（114 ページの「レイ トレーシングによる屈折」を参照）

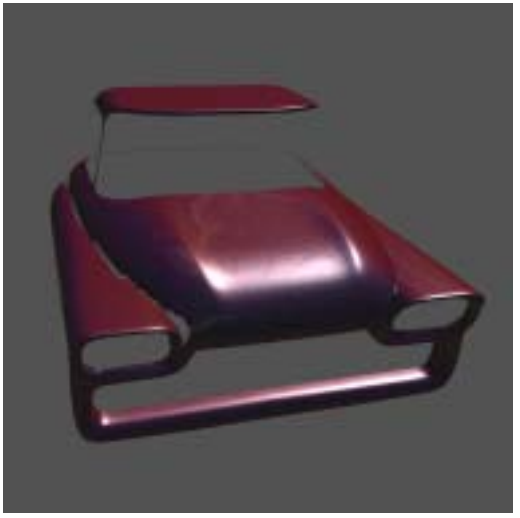
図 24 カラー図 - マルチペイントとレイ トレーシングされた屈折

皮膚の例
（119 ページの「皮膚」を参照）

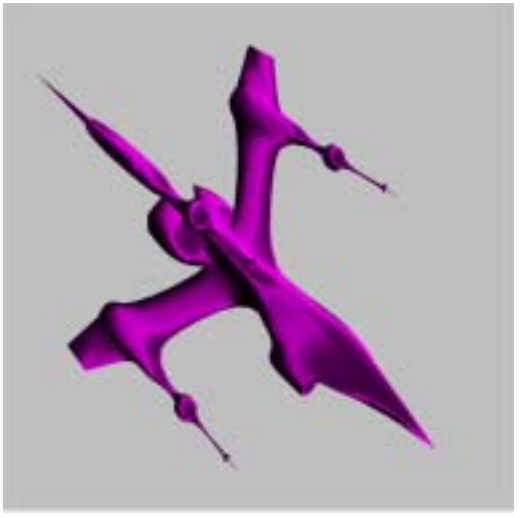


薄膜エフェクトの例
（124 ページの「薄膜エフェクト」を参照）

図 25 カラー図 - 皮膚と薄膜エフェクト



カー ペイント 9 の例
（127 ページの「カー ペイント 9」を参照）



異方性ライティングの例
（134 ページの「異方性ライティング」を参照）

図 26 カラー図 - カーペイント 9 と異方性ライティング

バンプ dot3x2 ディフューズおよびスペキュラの例
（136 ページの「バンプ dot3x2 ディフューズおよびスペキュラ」を参照）

バンプ反射マッピングの例
（140 ページの「バンプ反射マッピング」を参照）

図 27 カラー図 - バンプ dot3x2 ディフューズおよびスペキュラと、バンプ反射マッピング

フレネルの例
（144 ページの「フレネル」を参照）

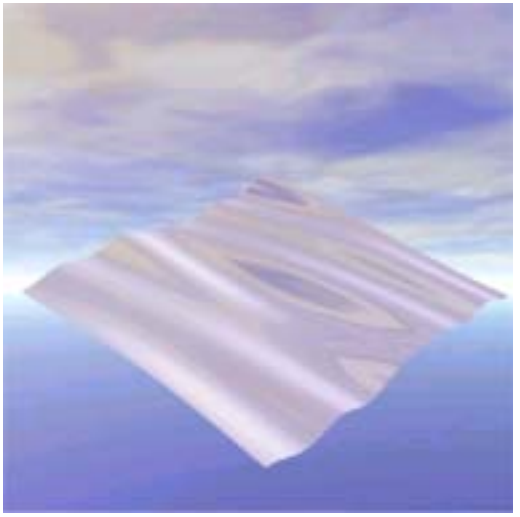
草の例
（146 ページの「草」を参照）

図 28 カラー図 - フレネルと草

屈折の例
（149 ページの「屈折」を参照）

シャドウ ボリューム掃引の例
（155 ページの「シャドウ ボリューム掃引」を参照）

図 29 カラー図 - 屈折とシャドウ ボリューム押出し



サイン波の例
（158 ページの「サイン波デモ」を参照）

行列パレット スキニングの例
（161 ページの「行列パレット スキニング」を参照）

図 30 カラー図 - サイン波と行列パレット スキニング

索引

よ

ら

り

れ

わ