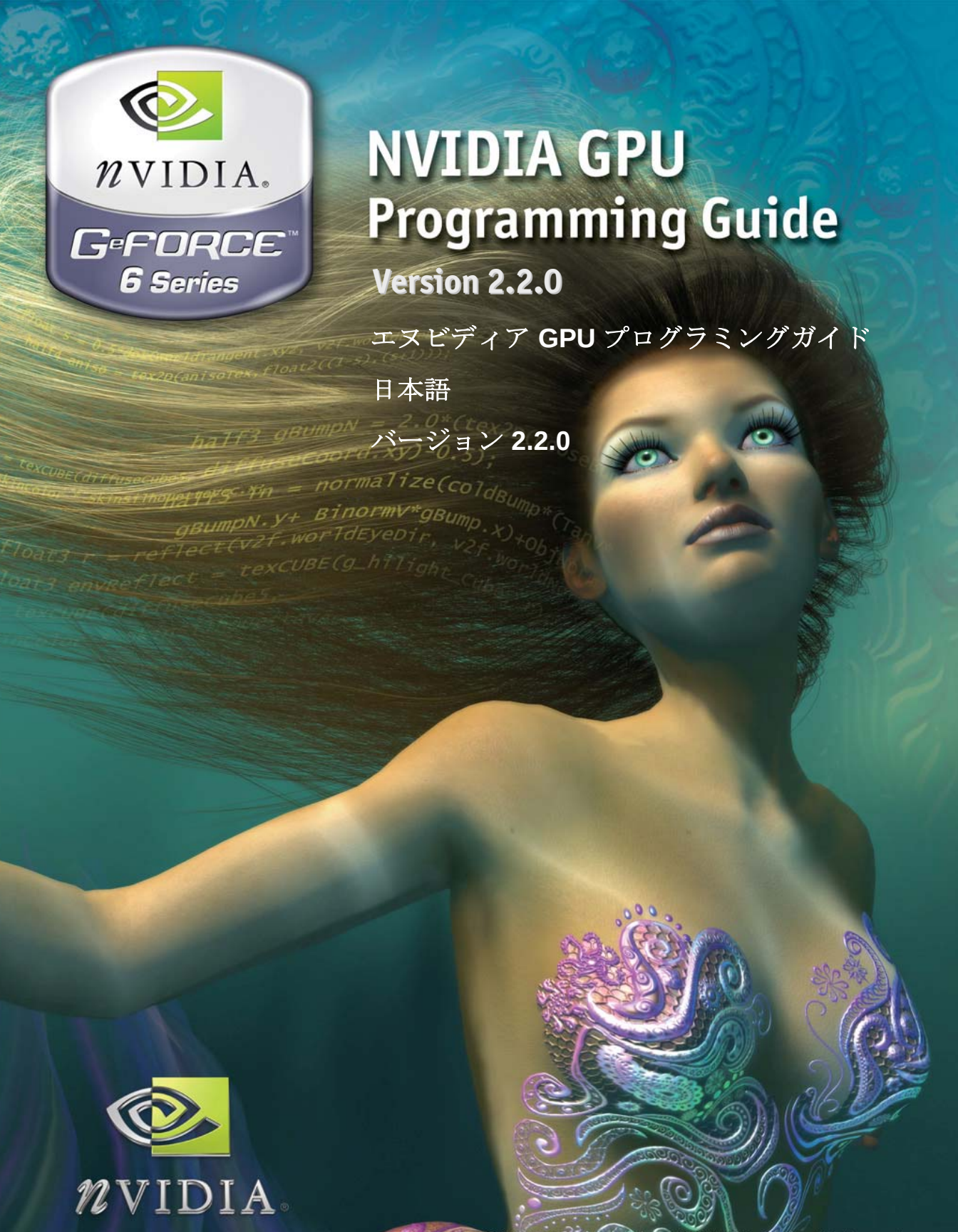

NVIDIA
GeFORCE
6 Series
NVIDIA GPU
Programming Guide
Version 2.2.0
エヌビディア GPU プログラミングガイド
日本語
バージョン 2.2.0
NVIDIA

HISTORY OF MAJOR REVISIONS

Version	Date	Changes
2.2.0	11/16/2004	Added normal map format advice Added ps_3_0 performance advice Added "General Advice" chapter
2.1.0	07/20/2004	Added Stereoscopic Development chapter
2.0.4	07/15/2004	Updated MRT section
2.0.3	06/25/2004	Added Multi-GPU Support chapter
2.0.0	06/01/2004	Added NV40 (GeForce 6 Series) chapters Renamed as "NVIDIA GPU Programming Guide"
1.0.0	07/14/2003	GeForce FX Programming Guide

目次

第1章
このドキュメントについて

1.1. はじめに

このガイドは、アプリケーション、グラフィックスAPI、およびグラフィックス処理ユニット（GPU）から最高のグラフィックス性能を引き出す手助けになるでしょう。このガイドにある情報を理解すれば、よりよいグラフィカルアプリケーションを書く上で役に立ちますが、お困りのときは、いつでも、devsupport@nvidia.com にeメールを送ってヘルプやアドバイスを求めてください。

このドキュメントは次のように構成されています。

- 第1章（本章）ではドキュメントの内容を簡単に説明します。
- 第2章では共通のボトルネックを見つけ、それに対処することによってアプリケーションを最適化する方法について説明します。

- 第 3 章では、ボトルネックを確認した後、それを解決するために役立つヒントを紹介します。ヒントは、最初に最も重要な最適化を行なえるように分類され優先順位が付けられています。
- 第 4 章では NVIDIA® GeForce™ 6 Seriesおよび NV4X-based Quadro FX に関するプログラミングに役立つヒントをいくつか提供します。これらのヒントは機能重視ですが、ケースによっては性能にも重点を置くものです。
- 第 5 章では、NVIDIA® GeForce™ FXおよびNV3X-based Quadro FX GPUに関するプログラミングに役立つヒントをいくつか提供します。これらのヒントは機能重視ですが、ケースによっては性能にも重点を置くものです。
- 第 6 章では NVIDIA GPU に関する全般的なアドバイスを提供します。性能、GPU 識別など種々様々なトピックをとりあげます。
- 第 7 章では、NVIDIA の Scalable Link Interface (SLI) 技術について説明しますが、これは複数の GPU を使用して性能を劇的に向上させることが可能なテクノロジです。
- 第 8 章では当社の立体ゲームサポートを活用する方法について説明します。うまく書かれた立体ゲームは躍動感があり、非立体ゲームよりもはるかに実体験を感じることができます。
- 第 9 章では、NVIDIA のパフォーマンスツールの概要を紹介します。
- 第 10 章では、参照しやすいように社内のコードネームと正式な製品名を付記して、当社の GPU を一覧表にします。

1.2. フィードバックの送信

このドキュメントに関してコメントや提案があれば、devsupport@nvidia.com宛に送ってください。

第2章
アプリケーションを最適化する方法

このセクションではグラフィックスアプリケーションで性能的なボトルネックを見つけ、それを取り除く代表的な対処法について検討します。

2.1. 正確に測定する

試験済の信頼できる性能インジケータとなる、性能測定ができる便利なツールが数多くあります。たとえば、 NVPerfHUDの黄色い線（詳細はNVPerfHUDのドキュメント参照）はフレームあたりの合計ミリ秒（ms）を示し、現在のフレームレートを表示します。

有効な性能比較を行なうには、次の手順を実行してください。

- **アプリケーションがクリーンに動作していることを確認。** たとえば、アプリケーションが Microsoft の DirectX Debug ランタイムを実行しているとき、エラーや警告が発生しないこと。

- **試験環境が有効であることを確認。**即ち、リリースバージョンのアプリケーションとその DLL、および最新版の DirectX のリリースランタイムを実行していることを確認します。
- **ソフトウェアはすべてリリースバージョン (デバッグビルドでなく) を使用。**
- **ディスプレイの設定がすべて正しいことを確認。**通常これは、デフォルト値設定になっているということです。特に、異方性フィルタリングとアンチエイリアシングの設定は性能に影響します。
- **垂直同期を無効にする。**これにより、フレームレートがモニターのリフレッシュレートによって制限されることがなくなります。
- **ターゲットハードウェアで実行。** 特定のハードウェア構成で十分な性能が出るかを調べる場合は、適正な量のシステムメモリを搭載して、正しい CPU および GPU で実行するようにしてください。ローエンドのシステムとハイエンドのシステムでは、ボトルネックが大きく変わる可能性があります。

2.2. ボトルネックを見つける

2.2.1. ボトルネックを知る

この時点では性能が悪いことを確認済みと仮定します。今度は性能上のボトルネックを見つける必要があります。一般にボトルネックはその状況の内容によって変化します。さらに複雑なことには、1 フレームの途中で変化してしまうことがよくあります。ですから、「ボトルネックを見つける」というのは、実際には、「このシナリオで最も障

害になるボトルネックを見つけよう」という意味になります。このボトルネックを除去すれば最高の性能を発揮できるのです。

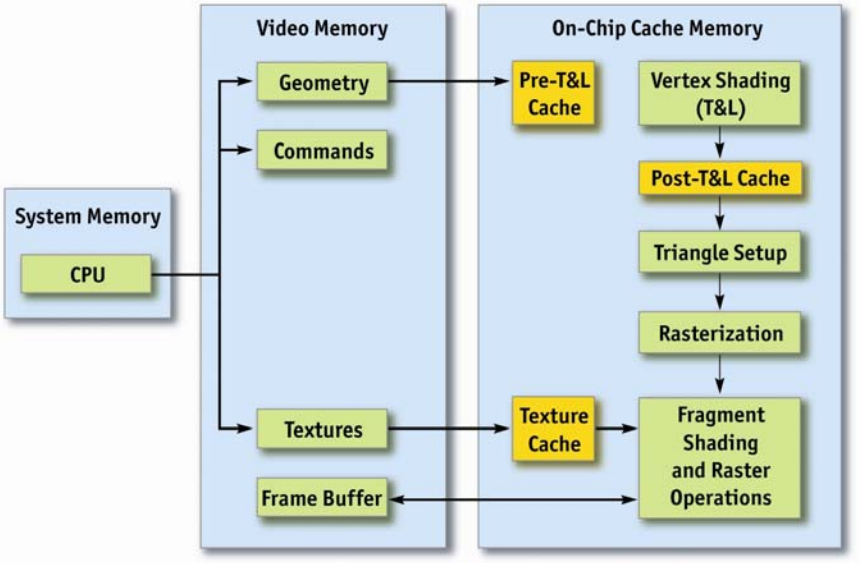


図 1. ボトルネックになりえるもの

理想的なケースでは、ボトルネックはひとつもないでしょう、即ち、CPU、AGP バス、および GPU パイプラインの段階がすべて均等にロードされています（図 1 参照）。残念ながらそのようなケースは現実世界のアプリケーションでは不可能で、実際には、何かが常に性能上の障害になっています。

ボトルネックは CPU あるいは GPU に存在するかもしれません。NVPerHUD の緑の線は（NVPerHUD の詳細はセクション 9.1 参照) GPU が 1 フレーム中にアイドル状態になっている時間をミリ秒で示します。GPU がフレームあたり 1 ミリ秒でもアイドル状態があれば、そのアプリケーションは少なくとも部分的に CPU の制限を受けてい

ることを示します。GPU のアイドル状態がフレーム時間のうち大きな割合を占めている場合、即ち、すべてのフレームで 1 ミリ秒でもアイドル状態がありかつアプリケーションが CPU と GPU を同期化させていない場合は、CPU が最も大きなボトルネックです。GPU 性能を向上させても GPU のアイドル時間が増えるだけです。

2.2.2. 基礎テスト

いくつか簡単なテストを行なってアプリケーションのボトルネックを確認することができます。特別なツールやドライバは必要ありませんから、いつでの簡単に始めることができます。

- **すべてのファイルアクセスをなくす。**ハードディスクアクセスがあるとフレームレートが削られます。この状態はWindowsのPerfmonツール、AMDのCodeAnalyst(http://www.amd.com/us-en/Processors/DevelopWithAMD/0,,30_2252_3604,00.html)、あるいはIntelのVTune(http://www.intel.com/software/products/vtune/)を使って、ご使用のコンピュータの「使用中のハードディスク」ランプあるいはディスク性能モニター信号を調べるだけで簡単に検出することができます。ハードディスクアクセスは、メモリスワッピング（特にアプリケーションが大量のメモリを使用する場合に）によっても発生することがあることに留意してください。
- **異なる速度で同一 GPU を CPU で実行する。**CPU 速度を調整（すなわち、ダウンクロック）できるシステム BIOS を見つけると役に立つでしょう。なぜなら、ひとつのシステムだけでテスト可能だからです。フレームレートが CPU の速度に比例して変化する場合は、そのアプリケーションは CPU の制限を受けていることになります。

- **GPU のコアクロックを下げる。**Coolbits（第 6 参照）など公開されているユーティリティを使用してこれを行なうことができます。コアクロックを下げるとそれに比例して性能が落ちる場合は、アプリケーションは、バーテックスシェーダ、ラスタライズ、あるいはフラグメントシェーダの制限を受けていることになります（すなわち、シェーダの制限）。
- **GPU のメモリクロックを下げる。**Coolbits（第 6 参照）など公開されているユーティリティを使用してこれを行なうことができます。メモリクロックを下げると性能に影響する場合は、アプリケーションはテクスチャまたはフレームバッファ帯域幅の制限を受けていることになります（GPU 帯域幅の制限）。

一般に、CPU ボトルネックと GPU ボトルネックの判断を素早く行なうには、CPU 速度、GPU コアクロック、および GPU メモリクロックを変えるのが簡単な方法です。CPU のクロックを n パーセント下げると性能が n パーセント下がる場合は、そのアプリケーションは CPU の制限を受けています。GPU のコアクロックおよびメモリクロックを n パーセント下げると性能が n パーセント下がる場合は、そのアプリケーションは GPU の制限を受けています。

2.2.3. NVPerfHUD を使用する

CPU ボトルネックを確認するには、Null Hardware（Null HW）と呼ばれる、特殊なドライバでアプリケーションを実行します。このドライバは通常のドライバのように動作しますが（通常のドライバとすべて同じコードパスを通じて実行される）、実際には GPU に何も作業を渡さない点が異なります。したがって、Null HW ドライバは、無限

に高速な GPU をエミュレートします。Null HW ドライバの性能が通常のドライバの域を出ない場合は、そのアプリケーションは全面的に CPU の拘束を受けていることになります。

NVPerfHUD 2.0 には、アプリケーションのすべての描画呼び出しを抑え、したがって Null Hardware Driver をエミュレートする、特殊なモードがあります。しかし、すべての描画呼び出しを除外することは、CPU の負荷を軽くすることにもなります。各描画呼び出しの前に行なわれる状態変更を処理およびコミットする必要がなくなるからです。NVPerfHUD は他にも様々な有用な機能を備えており、それらは性能問題の確認に役立ちます。

NVPerfHUD を使って GPU がまったくアイドル状態にならない場合、そのアプリケーションは GPU の制限を受けています。NVPerfHUD の青い線は、ドライバが GPU の待ち状態に入っている時間をミリ秒で示しますから、GPU に拘束される性能を確認することができます。

NVPerfHUD のユーザガイドには、ボトルネックを確認してそれを取り除く詳細な方法、トラブルシューティングなどが記載されています。このガイドは次の URL から入手できます。

http://developer.nvidia.com/object/nvperfhud_home.html.

2.3. ボトルネック：CPU

アプリケーションが CPU の拘束を受けている場合は、プロファイリングを使用して何が CPU 時間を消費しているのかを特定してください。次のモジュールは通常、相当量の CPU 時間を使います。

- アプリケーション（実行ファイルと関連 DLL）
- ドライバ（nv4disp.dll、nvoglnt.dll）

- DirectX ランタイム（d3d9.dll）
- DirectX Hardware Abstraction Layer（hal32.dll）

この段階での目標は、CPU のオーバーヘッドを減少して、CPU がボトルネックにならないようにすることですから、何が CPU 時間を最も消費しているかは比較的重要な問題です。有用なアドバイスとして、マイナーな最適化よりもアルゴリズムの改善があります。そして、もちろん、最大の性能を得るために、CPU を最も消費しているものを突き止めます。

次に、アプリケーションコードを入念に調べて、コードモジュールを除去あるいは小さくできないかをチェックします。アプリケーションが、hal32.dll、d3d9.dll または nvoglnt で大量に CPU を消費している場合は、API を乱用している印です。もしドライバが大量に CPU を消費しているなら、ドライバに対する呼び出しの数を減らせばいいでしょう。バッチサイズを改善することもドライバ呼び出しの減少に役立ちます。次のプレゼンテーションにバッチングに関する詳細情報があります。

http://developer.nvidia.com/docs/IO/8230/BatchBatchBatch.ppt

http://download.nvidia.com/developer/presentations/GDC_2004/Dx9Optimization.pdf

NVPerfHUD は、ドライバのオーバーヘッドの確認にも役立ちます。フレームあたりドライバで消費された時間を表示できますし（赤の線で表す）、フレームあたり描画されたバッチの数をグラフ化します。

性能が CPU の拘束を受けている場合にチェックする領域は他に次のものがあります。

- **フレームバッファやテクスチャなど、アプリケーションがリソースをロックしているか？** リソースをロックすると CPU および GPU をシリアライズする可能性があり、結果的に GPU が口ックを返す用意ができるまで、CPU を待たせることになります。したがって CPU はアクティブに待ち状態に入り、アプリケーションコードの処理ができなくなります。したがって、口ックは CPU のオーバーヘッドの原因になります。
- **アプリケーションが CPU を使って GPU をプロテクトしているか？**小セットのトライアングルをカリングすると、CPU の作業が発生し、作業が GPU に退避されますが、GPU はすでにアイドル状態です。これらの CPU 側の最適化を行なえば、実際に、CPU 拘束時には性能の向上が見られます。
- **CPU の作業負荷を GPU に移すことを考える。**アルゴリズムを構成しなおして GPU のバーテックスまたはピクセルプロセッサに適合するようにできませんか？
- **シェーダを使用してバッチサイズを増やし、ドライバのオーバーロードを減少させる。**たとえば、2 つのマテリアルをひとつのシェーダに一体化し、2 つのバッチをそれぞれのシェーダで描画するのではなく、ひとつのバッチでジオメトリを描画するようにできます。Shader Model 3.0 は様々な状況で有用であり、複数のバッチをひとつに圧縮し、バッチおよび描画の両方のオーバーヘッドを減少させることができます。Shade Model 3.0 の詳細については、 セクション **Error! Reference source not found.** を参照してください。

2.4. ボトルネック：GPU

GPUは深くパイプライン化されたアーキテクチャを持っています。GPUがボトルネックの場合、どの段階のパイプラインが最大のボトルネックになっているかを突き止める必要があります。グラフィックスパイプラインの様々な段階の概要については、次を参照してください。
http://developer.nvidia.com/docs/IO/4449/SUPP/GDC2003_PipelinePerformance.ppt.

NVPerfHUD は、様々な GPU およびドライバの機能をオン、オフさせることによって、仕組みを簡略化しています。たとえば、ミップマップ LOD バイアスにすべてのテクスチャを 2 × 2 にさせることができます。これで性能が大きく向上する場合は、テクスチャキャッシュミスがボトルネックです。NVPerfHUD は、同様にして、シェーダの全部または一部を 1 サイクルで実行させることによって、ピクセルシェーダの実行時間に対するコントロールを許可します。

GPUがアプリケーションのボトルネックになっていることを確認した場合は、**Error! Reference source not found.**に示されているヒントを使用して性能を改善してください。

第3章
一般的な GPU 性能に関するヒント

この章では、GeForce FX および GeForce 6 シリーズ GPU で最適な性能を実現するのに役立つ、最上位の性能に関するヒントを示します。便利のため、ヒントはパイプラインの段階ごとに構成してあります。各サブセクション内では、まず何に重点を置けばよいかわかるように、ヒントを重要度にそって並べてあります。

最新の GPU パイプライン性能の概要を知る上ですばらしいのは、「GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics」という本の「Graphics Pipeline Performance」章です。この章では、グラフィックスパイプラインのあらゆる部分の性能上の問題の解決方法はもちろん、ボトルネックの特定についても解説されています。

Graphics Pipeline Peformanceは次のURLから自由に入手できます。
http://developer.nvidia.com/object/gpu_gems_samples.html.

3.1. ヒントのリスト

正しく使用すれば、GeForce FX および GeForce 6 シリーズ GPU は最高レベルの性能を発揮することができます。このリストは性能上のヒントの概要を示します。詳細はそれ以降のセクションで説明します。

- **バッチングの不具合が CPU ボトルネックの原因**
  - バッチをあまり使わない
    - テクスチャアトラスを使用してテクスチャ状態変化を回避します。
      http://developer.nvidia.com/object/nv_texture_tools.html
    - DirectX では、Instancing API を使用して SetMatrix および類似のインスタンス生成状態変化を回避します。
- **バーテックスシェーダが GPU ボトルネックの原因**
  - インデックス付きプリミティブ呼び出しを使用
    - DirectX 9 のメッシュ最適化呼び出しを使う
      [`ID3DXMesh:OptimizeInplace()` または `ID3DXMesh:Optimize()`]
    - インデックス付きリストの効果がない場合は、NVTriStripユーティリティを使う
      http://developer.nvidia.com/object/nvtristrip_library.html
- **ピクセルシェーダが GPU ボトルネックの原因**
  - 機能を満たす最小限度のピクセルシェーダバージョンを選択

- 自分のシェーダを開発するときは、もっと高いバージョンを使用しても OK です。まずそれを動作させ、ピクセルシェーダバージョンを下げてそれを最適化できるか調べます。
- ps_2_*機能が必要な場合は、ps_2_a プロファイルを使用します。
- 機能を満たす範囲で最下位のデータ精度を選択。
  - float よりも half の方を選びます。
- 可能な限りすべてにこの型を使用します。
  - 変動パラメータ
  - 一定パラメータ
  - 変数
  - 定数
- バーテックスシェーダとピクセルシェーダをバランスさせる。
- ピクセルシェーダが制約になっている場合は、線形化可能な計算をバーテックスシェーダにまかせる。
- ピクセルシェーダのライフにわたって変化しない定数に一定パラメータを使用しない。
- 代数を使って計算を軽減できないか調べる。
- 複雑な関数をテクスチャ参照で置き換える
  - ピクセル単位のスペキュラーライティング
  - FX Composer を使用してプログラムで生成したテクスチャをファイルにする
  - ただし、sincos、log、exp はネイティブな命令であり、テクスチャ参照で置き換える必要はない

- **テクスチャ作成が GPU ボトルネックの原因**
  - ミップマッピングを使用
  - 手堅くトライリニア型および異方性フィルタリングを使用
    - 異方性フィルタリングのレベルをテクスチャの複雑度合いに合わせる。
    - 当社のPhotoshopプラグインを使用して異方性フィルタリングのレベルを変化させ、どのように見えるかを調べる。
      http://developer.nvidia.com/object/nv_texture_tools.html

経験から得た単純な規則に従う。テクスチャのノイズが多い場合は、異方性フィルタリングをオンにすること。

- **ラスタライズが GPU ボトルネックの原因**
  - 倍速 Z-Only およびステンシルレンダリング
  - Early Z-cull（早期 z カリング）による最適化
- **アンチエイリアシング**
  - アンチエイリアシングの利用方法

3.2. バッチング

3.2.1. あまりバッチを使用しない

「バッチング」とは、トライアングルごとに API を 1 回ずつ呼び出さないで、多くのトライアングルを 1 回の API 呼び出しで描画できるように、ジオメトリを集めてグループ化することを指します。API を呼び出すごとにドライバのオーバーヘッドが発生しますから、API

呼び出しをできるだけ少なくしてこのオーバーヘッドを減少させるのが最もよい方法です。言い換えれば、数千のトライアングルを一度に描画して描画呼び出しの数を減少させます。もっと大きいバッチを少数使うのも、性能向上には、いい方法です。GPU は常にパワフルになっていますから、最適なレンダリング速度を達成するためには、効率的なバッチングがますます重要になります。

3.3. バーテックスシェーダ

3.3.1. インデックス付きプリミティブ呼び出しを使う

インデックス付きプリミティブ呼び出しを使うことによって、GPU はその変換/ライティング後頂点キャッシュを利用することができます。頂点を調べてそれが変換済であることがわかれば、2 度変換することはありません。キャッシュを使うだけです。

DirectX では、ID3DXMesh クラスの OptimizeInPlace()あるいは Optimize()関数を使用してメッシュを最適化し、バーテックスキャッシュにもっと向いたものにすることができます。

また、NVTriStripユーティリティを使用して最適化された、キャッシュ向きのメッシュを作成することもできます。NVTriStripは標準プログラムであり、次のURLから入手できます。
http://developer.nvidia.com/object/nvtristrip_library.html.

3.4. シェーダ

高レベルのシェーディング言語はパワフルで柔軟性のあるメカニズムを提供し、シェーダを簡単に書くことが可能になります。残念ながら、

このことは言い換えれば、低速なシェーダを書くのも以前より簡単になった、ということになります。注意しないと、低速なシェーダの自然破裂によってアプリケーションの停止という事態に陥る可能性があります。下記は、単純な効果のために非効率的なシェーダを書かないために役立つヒントです。さらに、GPU の計算能力を最大限活用する方法も知ることができるでしょう。正しく使用すれば、ハイエンドの GeForce FX GPU はクロックサイクルあたり 20 以上の演算が可能です。最新の GeForce 6 シリーズ GPU はその何倍もの性能をもたらすことができます。

3.4.1. 機能を満たす最小限度のピクセルシェーダバージョンを選択

機能を満たす最小限度のピクセルシェーダバージョンを選択します。たとえば、要素あたり 8 ビット程度のテクスチャで単純なテクスチャフェッチやブレンド演算をしている場合は、ps_2_0 以上のシェーダを使う必要はありません。

3.4.2. ps_2_a プロファイルを使用してピクセルシェーダをコンパイル

Microsoft の HLSL コンパイラ (fxc.exe) は、コンパイル対象のプロファイルに基づいてチップ固有の最適化を追加します。GeForce FX GPU を使用しており、シェーダが ps_2_0 以上を要求する場合は、ps_2_a プロファイルを使ってください。これは、そのまま GeForce FX ファミリーに相当する ps_2_0 機能のスーパーセットです。ps_2_a プロファイル向けにコンパイルする方が汎用 ps_2_0 プロファイル向けにコンパイルするよりおそらく性能がよくなるでしょう。2003 年 7 月に HLSL のリリースが開始された時点では ps_2_a プロファイルしかありませんでした。

一般に、最新版の fxc を使用してください（DirectX 9.0c 以降）。Microsoft はリリースごとにコンパイル性能を上げ、バグフィックスをしているからです。GeForce 6 シリーズ GPU の場合は、適切なプロファイルと最新のコンパイラでコンパイルするだけで十分です。

3.4.3. 機能を満たす範囲で最下位のデータ精度を選択

性能と品質両方に影響するもうひとつの要因は、演算およびレジスタに使用される精度です。GeForce FX および GeForce 6 シリーズ GPU は 32 ビットおよび 16 ビットの浮動小数点フォーマット（それぞれ float 型、half 型と呼ばれる）、および 12 ビット固定小数点フォーマット（fixed と呼ばれる）をサポートします。float データ型は IEEE と非常に似ており、s23e8 フォーマットです。half も s10e5 フォーマットである点で IEEE に似ています。12 ビット fixed 型は、[-2,2)の範囲をカバーし、ps_2_0 以降のプロファイルでは使用できません。fixed 型は、DirectX では ps_1_0 から ps_1_4 までのプロファイルで使用でき、OpenGL では NV_fragment_program extension 拡張または Cg のいずとも使用できます。

これらの型の性能は精度によって変わります。

- fixed 型は最も高速であり、カラー計算などの低精度の計算に使用します。
- 浮動小数点精度が必要な場合は、half 型の方が float 型より高性能です。half 型を慎重に使用すれば、ほとんどのアプリケーションで、フレームレートは 3 倍になり、32 ビットフルの結果の最下位ビット（LSB）ひとつで、レンダリングしたピクセルの 99%以上を表現できるのです。
- 最高可能精度が必要な場合は、float 型を使います。

/Gpp フラッグ（2003 年 7 月の HLSL アップデートから使用可能）を使用してシェーダ内ですべてを強制的に half 精度にすることができます。シェーダを動作状態にした後、このセクションのヒントにしたがって、このフラッグを有効にし、性能および品質に対する影響を調べます。何もエラーが出なければ、このフラッグをそのまま有効にしておきます。他の方法として、その方が有利な場合は、手動で half 精度に下げます（/Gpp は可能な最高性能限度を規定します）。

half または fixed 型を使用する場合は、必ず変動パラメータ、一定パラメータ、変数および定数に使用するようにしてください。DirectX のアセンブリ言語を ps_2_0 プロファイルと一緒に使用する場合は、_pp 修飾子を使用して計算の精度を下げてください。

OpenGLのARB_fragment_program言語を使用している場合、できるだけ精度を下げて実行時間を最小限に抑えたいときはARB_precision_hint_fastestオプションを使い、命令ごとに精度をコントロールしたいときはNV_fragment_programオプションを使います（http://www.nvidia.com/dev_content/nvopenglspecs/GL_NV_fragment_program_option.txt参照）。

多くのカラーベースのオペレーションが、精度を損なわずに、fixed または half データ型で実行できます（たとえば、tex2D*diffuseColor 演算）。

OpenGL では、GeForce FX ハードウェアで、ほとんどが浮動小数点動作からなるシェーダを、固定小数点精度でオペレーション（正規化ベクトルの内積など）を実行することによって、スピードアップすることができます。

たとえば、正規化の結果は、カラー同様に、half 精度でかまいません。位置も half 精度でかまいませんが、関係値をほぼゼロにするためにバーテックスシェーダでスケールを変える必要があるかもしれません。

たとえば、値を局所接空間に移動し、それから位置をスケールダウンすると、非常に大きな位置を half 精度に変換するときに見られる縞状のアーチファクトをなくすことができます。

3.4.4. 代数を使って計算を軽減

シェーダを動作状態にしたら、計算について調べ、数学的特性を使用して計算を圧縮できないか考えます。これは複数のシェーダにわたって共有されるライブラリ関数の場合は特に当てはまります。例を示します。

- 汎用球面マップ投影はたいてい次の式で表されます。

  $p = \mathrm{sqrt}(Rx^2 + Ry^2 + (Rz + 1)^2)$

  これは次のように展開されます。

  $p = \mathrm{sqrt}(Rx^2 + Ry^2 + Rz^2 + 2Rz + 1)$

  反射ベクトルが正規化されていることがわかっている場合（セクション Error! Reference source not found. および Error! Reference source not found.参照）、はじめの 3 項の合計は 1.0 になることが保証されます。この式は次のように書き直すことができます。

  $p = \mathrm{sqrt}(2 * (Rz + 1)) = 1.414*\mathrm{sqrt}(Rz + 1)$

- 1.414 の掛け算を別の定数に畳み込む（セクション Error! Reference source not found.参照）、内積を軽減します。
- ドット（正規化（N）、正規化（L））ははるかに効率的に計算できます。

- これは通常(N/|N|) dot (L/|L|)として計算され、2 つの高価な逆数平方根 (rsq) が必要です。
- 次のちょっとした代数で下記が得られます。
- (N/|N|) dot (L/|L|)
- = (N dot L) / (|N| * |L|)
- = (N dot L) / (sqrt((N dot N) * (L dot L))
- = (N dot L) * rsq((N dot N) * (L dot L))
- 高価な rsq 演算はひとつしか必要ありません。

3.4.5. ベクトル値を複数補間式のスカラー成分にパックしない

あまり多くの情報を計算にパックすると、コンパイラがコードを効率的に最適化するのが難しくなります。たとえば、接線マトリクスを渡す場合、3q 成分にビューベクトルを含めないことです。この間違いは次のように示されます。

```
// 間違った習慣
tangent = float4(tangentVec, viewVec.x)
binormal = float4(binormalVec, viewVec.y)
normal = float4(normalVec, viewVec.z)
```

こうしないで、ビューベクトルを 4 次補間式に入れます。

3.4.6. オーバーレイ汎用ライブラリ関数を書かない

複数のシェーダにわたって共有される関数は、非常に汎用的に書かれることがよくあります。たとえば、反射はたいてい次のように計算されます。

```
float3 reflect(float3 I, float3 N) {
       return (2.0*dot(I,N)/dot(N,N))*N – I;
}
```

このように書けば、反射ベクトルは法線ベクトルや入射ベクトルの長さと無関係に計算できます。しかし、シェーダの作成者は、ライティング計算を行なうために、少なくとも正規化された法線ベクトルが欲しい場合がよくあります。この場合は、内積、逆数、およびスカラー乗算は reflect()から外すことができます。このような最適化は性能を劇的に向上さることができます。

3.4.7. 正規化されたベクトルの長さを計算しない

オーバーレイ汎用ライブラリ関数に共通な（かつ不経済な）例として、インプットベクトルを局所的に計算するということがあります。しかし、ベクトルはたいていは関数を呼び出す前に正規化されています。コンパイラはそれを検知しませから、1.0 を計算するためにかなりのピクセル単位の計算がなされています。

ライブラリ関数がベクトルの長さと無関係に正しく動作しなければならない場合、関数へのスカラーパラメータを長さにしてみてください。そうすれば、関数を呼び出す前にベクトルを正規化するシェーダは 1.0 という定数の値を渡すことができ（長さを計算しないという利益をもたらし）、ベクトルを正規化しないシェーダは長さを計算できるのです。

3.4.8. 一定の定数式を畳み込む

多くの開発者がピクセルシェーダで動的定数を含む式を計算しています。2 つ以上の一定の定数（または一定かつインラインの定数）を式に使用する場合、それらの定数を畳み込んで性能を向上させる方法があります。例を示します。

```
half4 main(float2 diffuse : TEXCOORD0,
           uniform sampler2D diffuseTex,
```

```
            uniform half4 g_OverbrightColor) {
  return tex2D(diffuseTex, diffuse) * g_OverbrightColor * 2.0;
}
```

g_OverbrightColor は CPU であらかじめ 2.0 を乗じて、フレームあたり数百万におよぶピクセルのピクセル単位の乗算を回避することが可能です。

できるだけ多くの定数式を畳み込むためには、式を分散あるいは因数に分解することが必要になる場合もあります。さらに、HLSL プリシェーダを使用してシェーダ実行前にあらかじめ CPU で計算しておくことも可能です。

もうひとつの共通の例は、頂点ごとに materialColor * lightColor を計算していることです。この式では任意のバッチの頂点はすべて同じ値を持ちますから、CPU で計算すべきです。

また、逆マトリクスや転置マトリクスは GPU でなく、CPU で計算すべきです。これらは、頂点ごとあるいはフラグメントごとにではなく、一度計算すればよいからです。/Zpr（行順パック）および/Zpc（列順パック）コンパイラオプションを使えばマトリクスを望みの形で保存するのに役立つでしょう。

3.4.9. ピクセルシェーダのライフにわたって変化しない定数に一定パラメータを使用しない

開発者は一定パラメータを使用して 0、1、255 などよく使われる定数を渡すことがあります。この習慣は避けるべきです。コンパイラが定数とシェーダパラメータを区別するのが難しくなり、性能の低下につながります。

3.4.10. バーテックスシェーダとピクセルシェーダをバランスさせる

高性能の実現はボトルネックを除去できるかにかかっています。それは実際には、パイプラインのあらゆる部分（CPU、AGP バス、およびグラフィックスパイプラインの段階）をバランスさせる必要があるということです。バーテックスシェーダを使うかピクセルシェーダを使うかの判断は次のいくつかの要素によって変わります。

- **オブジェクトをどのようにテセレーションしていますか？** フレームあたり数百万の頂点がある場合、バーテックスシェーダの負荷を軽くするのがよいでしょう。これはマルチパスアルゴリズムを使用している場合に、特に当てはまります。
- **ターゲットにしている解像度は？**アプリケーションを高解像度で実行するつもりであれば、ピクセルシェーダの方がボトルネックになる可能性が高い。したがって、バーテックスシェーダにもっと計算をまかせるのがよいでしょう。
- **ピクセルシェーダの期間は？** 複雑なシェーディングを行っている場合は、おそらくピクセルシェーダがボトルネックになるでしょう。ピクセルシェーダのコンパイルが（平均）20 サイクル以上になり、スクリーンの半分以上を占める場合は、アプリケーションはおそらく、GeForce FX ハードウェアで、ピクセルシェーダの制約を受けるでしょう。計算をバーテックスシェーダに移せないか調べてください。（例については、セクション **Error! Reference source not found.** 参照）。当社の NVShaderPerf ツールを使用してシェーダが何サイクル使っているか知ることができます。また、GeForce 6 シリーズなどもっと新しいハードウェア

を使えば、ピクセルシェーダが制約になることなく、もっと複雑なシェーディングが可能でしょう。

3.4.11. ピクセルシェーダが制約になっている場合は、線形化可能な計算をバーテックスシェーダにまかせる

ラスタライザは頂点ごとに値をとり、パースペクティブコレクションを実行しながら、フラグメントごとにそれらを補間します。線形計算をバーテックスシェーダに移すことによってすでにこれを実現しているハードウェアを利用します。こうすればピクセルシェーダでは、わずかな頂点の計算ですむし、また、補間された結果を受け取ることもできるでしょう。

たとえば、減衰のためにワールド空間からライト空間に移ることができます。あるいは、パーピクセル反射をキューブマップにマッピングしていない限り、バンプマッピングをしている場合は、移動先を接線空間パーバーテックスにできます。

3.4.12. mul()標準ライブラリ関数を使う

手動でマトリクス乗算を行わずに、mul()標準ライブラリ関数を使います。そうすれば、アプリケーションが補間式にマトリクスを渡すときに発生する可能性のある行順/列順の問題が回避されます。

3.4.13. 従属テクスチャ座標には、saturate()ではなくD3DTADDRESS_CLAMP（または GL_CLAMP_TO_EDGE）を使用

saturate()を使用すると一部の GPU では余分なコストがかかります。テクスチャ座標として、クランプされた結果を使用する場合は、シェーダで行うよりも、テクスチャハードウェアの能力を利用してテクスチャ座標を[0..1]の範囲にクランプする方がよいでしょう。

3.4.14. 最初に低い番号の補間を使う

低い番号のテクスチャ座標セット（TEXCOORD セット）を最初に使えば、高い性能が得られるでしょう。TEXCOORD0 からはじめて、TEXCOORD1、TEXCOORD2 などに上げていきます。

3.5. テクスチャリング

3.5.1. ミップマッピングを使う

縮小したテクスチャで「スパークリング」アーチファクトが発生しないように、アプリケーションでは常にミップマッピングを使います。画像品質の向上、テクスチャキャッシュの挙動の改善、および高性能が実現されるでしょう。メモリ使用量をわずか 30%増やすだけでこれをすべて実現するのですから、すばらしい効果といえます。3D テクスチャは、特に、ミップマッピングから大きな恩恵を受けることができ、ミップマッピングを有効にしたところ、30%から 40%も性能が向上したことがあります。

ミップマップを作成するときは、簡単にボックスフィルタを使ってミップマップをますます小さくすることのないようにしてください。その代わりに、ガウスフィルタまたはミッチェルフィルタを使用してもっとサンプルを取ります。それによってより高品質の結果がもたらされます。しかし、もう少し前処理に時間を使ってミップマップを作成すれば、アプリケーション実行時の安定性がよくなるでしょう。当社のPhotoshopプラグイン（NVIDIA Texture Toolsセットに含まれている）は高品質のミップマップを素早く作成できます。このセットは、次のURLから入手できます。
http://developer.nvidia.com/object/nv_texture_tools.html

3.5.2. 手堅くトライリニア型および異方性フィルタリングを使用

トライリニア型および異方性フィルタリングは、どちらも、画像品質の向上に役立ちますが、性能上の犠牲を伴います。トライリニア型および異方性フィルタリングは、必要な場合のみ使用するようにしましょう。一般に、高コントラスト細部が多いテクスチャで使うことが多いでしょう。異方性フィルタリングの場合は、テクスチャの方位も考慮することがあります。テクスチャがビューアに対して傾斜していることがわかっている場合は（たとえば、フロアテクスチャ）、そのテクスチャのために異方性フィルタリングのレベルを高くします。マルチテクスチャサーフェスの場合は、異なるレイヤーごとに適切なフィルタリングレベルを設定してください。

使用する異方性フィルタリングのレベルを決めるのに、当社のAdobe Photoshopプラグインが役に立ちます。このツールを使えば、様々なフィルタリングレベルを試して、視覚効果を見ることができます。このツールは次のURLから入手できます。

http://developer.nvidia.com/object/nv_texture_tools.html. アーチストにとっては、どのテクスチャに異方性フィルタリングあるいはトライリニア型フィルタリングが必要かを判断するために、このツールを使うとよいかもしれません。

3.5.3. 複雑な関数をテクスチャ参照で置き換える

テクスチャは複雑な関数をエンコードするいい方法です。オンザフライで参照できる多次元配列と考えてください。GeForce FX ファミリーはテクスチャに効率的にアクセスでき、たいてい、コストは算術演算と同じ程度です。FX Composer ツールを使用してこの種の最適化のプロトタイプを作ることが可能です。FX Conposer は次の URL から入手できます。

http://developer.nvidia.com/FXComposer

複雑なシーケンスの算術演算をテクスチャでエンコードできる場合は常に、性能を向上させることができます。log や exp など一部の複雑な関数は、ps_2_0 以降のプロファイルではマイクロ命令であり、したがって、最適性能のためにテクスチャでエンコードする必要はありません。

3.5.3.1. ピクセル単位のライティング

2D テクスチャの使用

テクスチャの有用性が発揮される共通の状況として、ピクセル単位のライティングがあります。一方の軸では (N dot L) 、他方の軸では (N dot H) で参照する 2D テクスチャを使うことができます。各 (u, v) 位置で、テクスチャは次のようにエンコードします。

```
max(N dot L,0) + Ks*pow((N dot L>0) ? max(N dot H,0) : 0), n)
```

これは標準的な Blinn ライティングモデルであり、ディフューズおよびスペキュラー項のクランピングを含みます。

1D ARGB テクスチャの使用

ひとつの役に立つ秘訣として、(N dot H) によって参照される 1D ARGB テクスチャを使用する方法があります。テクスチャは各チャンネルの様々な指数に応じて (N dot H) をエンコードします。例えば、次のようにエンコードします。

$$((N \cdot H)^4, (N \cdot H)^8, (N \cdot H)^{12}, (N \cdot H)^{16})$$

それから、各マテリアルに、これらの値をブレンドする 4 要素の加重定数が割り当てられ、シェーディング用にモノクロのスペキュラー値が与えられます。この方法のすばらしい点は、GeForce 4 クラスのハードウェアで機能することと、様々な外観を可能にする柔軟性を備えていることです。

3D テクスチャの使用

3D テクスチャを使うことによってミックスにスペキュラー指数演算を追加することもできます。最初の 2 軸は前のセクションで説明した 2D テクスチャ手法を使用し、3 番目の軸はスペキュラー指数 (鏡面) をエンコードします。

ただし、テクスチャが大きすぎるとキャッシュ性能が落ちることがあることを忘れないでください。最も使用頻度の高い指数だけをエンコードするとよいかもしれません。

3.5.3.2. ベクトルの正規化

ps_1_*シェーダを書いている場合は、正規化キューブマップを使用してベクトルを素早く正規化します。より高品質にするために、2 つの 16 ビット符号付キューブマップを使うことができます、ひとつは x と y 用で、もうひとつは z 用です。

実際には、正規化されていないベクトル V は、補間あるいはフィルタされた法線であるため、1 に近いのが普通であるという事実に基づく、もうひとつの最適化があります。つまり、1 / sqrt(x) at x = 1 というテイラー展開によって 1 /||V||の近似を求めることができるということです。

$$1 / \mathrm{sqrt}(x) \sim 1 + \tfrac{1}{2}(1 - x)$$

次のようになります。

$$V / ||V|| = V / \mathrm{sqrt}(||V||^2) = V + \tfrac{1}{2} V (1 - ||V||^2)$$

この公式は 2 つのアセンブリ命令で書くことができます。

```
dp3_sat r1, r0, r0
mad_d2 r1, r0, 1-r1, r0_x2
```

ここで r0 は V を含み、r1 の最終値は $V / ||V||$を含む。

このコードは、_x2 レジスタ修飾子であるため、ps_1_4 に対してのみ有効です。もっと下位のピクセルシェーダバージョンの場合は、代わりに、次の公式を使うことができます。

```
dp3_sat r1, r0_bx2, r0_bx2
mad r1, r0_bias, 1-r1, r0_bx2
```

これは r0 は ½ (V + 1)を含むと仮定しており、 V はたいてい[-1, 1]から[0, 1]までの圧縮範囲でピクセルシェーダに渡される必要があるので、制約になることはまずありません。

GeForce 6 シリーズ GPU は特別な half 精度の正規化ユニットを持っており、1 シェーディングサイクル中に無償で fp16 を正規化することができます。この機能を利用しましょう、正規化をひとつの fp16 で行なうだけで、コンパイラは nrmh 命令を生成します。

正規化に関して詳細は、次の URL から入手できる、当社の Normalization Heuristics and Bump Map Compression ホワイトペーパーを参照してください。

http://developer.nvidia.com/object/normalization_heuristics.html
http://developer.nvidia.com/object/bump_map_compression.html

3.5.3.3. sincos()関数

ここまでのアドバイスにもかかわらず、GeForce FX ファミリーおよびそれ以降の GPU は、ハードウェアに備わった複雑な数学関数をいくつかサポートしています。そのような関数の中で便利なのが sincos 関数であり、これによってある値のサインとコサインを同時に計算することが可能となります。

3.6. 性能

3.6.1. 倍速 Z-Only およびステンシルレンダリング

GeForce FX および GeForce 6 シリーズ GPU は深度あるいはステンシル値をレンダリングする場合にだけ、倍速でレンダリングします。この特別なレンダリングモードを有効にするには、次のルールに従う必要があります。

- カラー書き込みを無効にする
- アクティブな深度/ステンシルサーフェスをマルチサンプルしない
- Texkill をどのフラグメントにも適用していないこと
- 深度リプレイス（oDepth、texm3x2depth、texdepth）をどのフラグメントにも適用していないこと
- アルファテストを無効にする
- アクティブテクスチャでカラーキーを使用しない
- ユーザクリップ面を有効にしない

3.6.2. Early-Z 最適化

Early-Z 最適化（「z カリング」と呼ばれることもある）は隠蔽サーフェスのレンダリングを回避することによって性能の向上をはかります。隠蔽サーフェスにコストのかかるシェーダが適用される場合、z カリングによって大量の計算が軽減されます。z カリングを利用するには、次のガイドラインに従ってください。

- 穴付きのトライアングルを作成しない（即ち、アルファテストや texkill を避ける）
- 深度を修正しない（即ち、GPU が補間された深度値を使えるようにする）

これらのルールを守らないと、GPU が早期最適化に使うデータが無効になり、深度バッファが再度クリアされるまで z カリングができなくなる可能性があります。

3.6.3. 最初に深度を規定する

前述の 2 つの性能上の機能を利用する最適な方法は、「最初に深度を規定する」ことです。つまり、ファーストパスとして、倍速深度レンダリングを使用して（シェーディングなしで）シーンを描画します。これでビューアに最も近いサーフェスが確立されます。シーンを再度レンダリングすることができますが、今度はフルシェーディングです。z カリングは自動的に目に見えないフラグメントをカリングしますから、シェーディング計算を軽減することになります。

最初に深度を規定するには、それ自身のレンダリングパスが必要ですが、多くの隠蔽サーフェスにコストのかかるシェーディングが適用される場合は、性能的に有利です。倍速レンダリングはトライアングルが小さくなると効率が落ちます。また、トライアングルが小さいと z カリングの効率も落ちる可能性があります。

もうひとつ関連する手法として、NVSDK 7.1 以降で提供されている遅延シェーディング（Deferred Shading）があります。

3.6.4. メモリ割り当て

アプリケーションによるビデオメモリのスラッシングの可能性を最小限に抑えるために、シェーダおよびレンダーターゲットを割り当てる最もよい方法は次の通りです。

1. 最初にレンダーターゲットを割り当てる
    - ピッチごとに割り当て順序をソート（width * bpp）。
    - 使用頻度に基づき、異なるピッチグループをソート。最も頻繁にレンダリングされるサーフェスを最初に割り当てる。

2. バーテックスおよびピクセルシェーダを作成
3. 残りのテクスチャをロード

3.7. アンチエイリアシング

GeForce FX ファミリーおよび GeForce 6 シリーズは強力なアンチエイリアシングエンジンを備えています。アンチエイリアシングを有効にしておけば最高の性能を発揮しますから、アプリケーションでアンチエイリアシングを有効にすることを勧めます。

アンチエイリアシングと連携しない技術を使用する必要がある場合は、当社に連絡してください。皆様と問題を検討し、ソリューションを見つけるお手伝いをします。

DirectX 9.0b 以降では解決されていますが、ひとつの問題として、ポストプロセス効果と一緒にアンチエイリアシングを使うという問題があります。StretchRect()呼び出しは、マルチサンプリングに合わせて、バックバッファをオフスクリーンテクスチャにコピーすることができます。

たとえば、4x マルチサンプリングを、100 x 100 バックバッファで有効にする場合、ドライバは実際には、アンチエイリアシングを行うために、200 x 200 バックバッファと深度バッファを作成します。アプリケーションが 100 x 100 オフスクリーンテクスチャを作成する場合、それは StretchRect()でバックバッファ全体をオフスクリーンサーフェスに描画でき、GPU はアンチエイリアシングを施されたバッファをオフスクリーンバッファにフィルタをかけて渡します。

それからグローおよびその他のポストプロセス効果が 100 x 100 テクスチャで実施でき、さらにメインのバックバッファに適用し直されます。

この実質的なバックバッファサイズ（200 x 200）とアプリケーションのビュー（100 x 100）の解像度の不一致が、マルチサンプル z バッファを非マルチサンプルレンダーターゲットに貼り付けられない理由です。

第4章
GeForce 6 シリーズのプログラミングヒント

この章では、GeForce 6 シリーズおよび Quadro FX 4xxx GPU の機能をフルに生かすのに役立つ有用なヒントをいくつか紹介します。これらはほとんど機能に重点を置いていますが、一部は性能にも影響します。

4.1. シェーダモデル 3.0 のサポート

Microsoft DirectX 9.0 は高度なバーテックスおよびピクセルシェーダ技術をささえるためにいくつか新しいスタンダードを導入しました、バージョン 2.0 とバージョン 3.0 がそれです。シェーダモデル 2.0 ハードウェアは 2002 年後半から使用可能で、現在出荷されている大部分の GPU がシェーダモデル 2.0 以降をサポートしています。シェーダモデル 2.0 は高度なライティング及びアニメーション技法のために

有用な技術を含んでいますが、シェーダプログラムの長さと複雑さに制限があり、それが、実現可能な効果の忠実度の足かせとなっています。

開発者はピクセルシェーダ 2.0 およびバーテックスシェーダ 2.0 に固有の限界に対して立ち向かい、より新しい、より進んだシェーダモデル 3.0 を採用しはじめています。このシェーダモデルは、ピクセルおよびバーテックスシェーダ処理の両方で、いくつかの領域で先進的な技術を提供しています。

4.1.1. ピクセルシェーダ 3.0

ピクセルシェーダ 2.0 と 3.0 の機能上の主な差異を次に示します。

ピクセルシェーダの機能	シェーダ 2.0	シェーダ 3.0	説明
シェーダの長さ	96	65535+	より複雑なシェーディング、ライティング、プロシージャルマテリアルが可能。
ダイナミック分岐	No	Yes	無関係なピクセルの複雑なシェーディングをスキップすることによって、性能を無駄にしない。
シェーダアンチエイリアシング	非サポート	内臓デリバティブ命令	開発者はどの関数の画面空間デリバティブでも計算でき、アーチファクトを除去するためにシェーディング周波数またはオーバーサンプリングを調整可能。
バックフェイスレジス	No	Yes	シングルパスで両面ライティングが可能。

タ			
補間カラーフォーマット	最低 8 ビット整数	最低 32 ビット固定小数点	広範囲、高精度カラーにより頂点レベルで高ダイナミックレンジライティングが可能。
複数レンダーターゲット	任意	4 つ必要	進んだライティングアルゴリズムによりフィルタリングおよびバーテックス作業負荷を軽減、結果として最小のコストでより多くのライトが使える。
フォグおよびスペキュラー	最低 8 ビット固定関数	カスタム fp16-fp32 シェーダプログラム	シェーダモデル 3.0 は開発者に、スペキュラーおよびフォグ計算の完全かつ精度の高いコントロールを可能にする、以前は固定機能。
テクスチャ座標カウント	8	10	ピクセル単位のインプットが増え、よりリアルなレンダリングが可能、特にスキン。

4.1.2. バーテックスシェーダ 3.0

開発者がバーテックスシェーダモデル 2.0 から 3.0 に移ると使用できる主な機能の同様なリストを示します。

バーテックスシェーダの機能	**シェーダ 2.0**	**シェーダ 3.0**	**説明**
シェーダの長さ	256 命令	65535 命令	命令が増えてより詳細なキャラクタのライティングやアニメーションが可能
ダイナミック分岐	No	Yes	無関係な頂点のアニメーションや計算をスキップすることによって、性能を無駄にしない
頂点テクスチ	No	最大 4 テクスチャ	ディスプレートメントマッピング、パーテ

ヤ		から参照数任意	ィクル効果が可能
インスタンス生成サポート	No	必要	多くの多様なオブジェクトを 1 コマンドで描画可能

4.1.3. ダイナミック分岐

シェーダ 3.0 モデル (バーテックスとピクセル) のひとつの重要な機能は、ダイナミック分岐 (Dynamic Branching)です。これを追加するだけで、シェーダの作成者はシェーダプログラムにループと条件分岐を作成することができます。たとえば、ある人は、ある数の頂点ライトまでループするシェーダを書いて、どれが特定の頂点に影響するかを確認し、各関連ライトのインデックスをピクセルシェーダに渡すことができました。それからピクセルシェーダはこのライトインデックスを使ってどのライトパラメータを適用すべきか判断できました。このピクセルシェーダはアクティブなライト全体にわたってループし、すべてのライトの処理が済んだら、ダイナミック分岐を使ってループを抜けることができるのです。

ほとんどのライトのタイプがオブジェクトの正面 (ライトと向かい合う面) にのみ当たります。したがって、頂点およびピクセル分岐の両方を使って、シェーダがライトから見て外を向いていると感知するライトの処理をスキップすることができます (新しい「バックフェイス」レジスタを使用)。これによって大幅な処理時間の軽減になり、シェーダのスピードアップにつながります。同様のスピードアップを使って、キャラクタボーンアニメーションの処理や多くの類似のアルゴリズムをスキップすることができます。

4.1.4. コードのメンテナンスが容易

ゲームエンジンはますます複雑になっており、ピクセルシェーダ 2.0 のプログラムの長さ制限に対処するために、シェーダごとに多くの様々なバージョンを作ることが多くなっています。実行時に貴重なシステムメモリを占有するだけでなく、コードのメンテナンス、シェーダのコンパイル時間、レベルロード時間なども増加することになります。シェーダ 3.0 はこの問題を取り払いました。その包括的なループと分岐により、エンジンは、実行時に正しい実行パスを選択するために、適切な静的および動的分岐を含むシングルバーテックスシェーダとシングルピクセルシェーダを書くだけでよくなり、シェーダの組み合わせによる破裂問題が大幅に単純化されました。

4.1.5. インスタンス生成

シェーダモデル 3.0 のもうひとつの主要な機能は、Microsoft DirectXR Instancing API のサポートです。現在、ゲームは画面に表示できるオブジェクトの数の制限に直面しています。これはグラフィックスのパワーでなく、たいていは、CPU 側でわずかに異なるだけで多くのバリエーションがある同一のオブジェクトを保存あるいはサブミットする際のオーバーヘッドが原因です。たとえば、森はたいてい、互いに似ている木から構成されますが、それぞれは異なる位置にあり、高さや葉の色なども異なります。望みのバリエーションを追加するために、開発者は、互いにわずかに異なる木の数多くのコピーを保存するか、あるいは、それぞれの木を回転、スケール、色付けおよび配置するためにコストのかかるレンダー状態変化をさせるか、どちらかを選択する必要があります。

インスタンス生成を使えば、プログラマはひとつの木を保存し、それからいくつか他の頂点データストリームを保存して、インスタンス単

位で色、高さ、枝のサイズなどを指定することができます。たとえば、単一の 1000 頂点の木のモデルは頂点位置と法線を含み、200 要素の頂点ストリームは位置、カラー、および高さを含みます。インスタンス生成により、プログラマはひとつの描画呼び出しをサブミットし、木の基本形状の同じデータを使用して、200 本の木のそれぞれをレンダリングすることが可能です。ただし、インスタンス単位のストリームを通じてそれを変化させます。

当社のインスタンス生成コードのサンプルはつぎのURLから入手できます。
http://download.nvidia.com/developer/SDK/Individual_Samples/samples.html

4.1.6. 要約

要約すれば、DirectX 9.0 シェーダモデル 3.0 は、使いやすさ、性能、およびシェーダの複雑さの点で大幅な進歩があります。ダイナミック分岐は、早期退職の可能性のある多くのアルゴリズムにスピードアップをもたらし、同時に、グラフィックスエンジンやツールのシェーダコードパスを簡略化します。最後に、インスタンス生成により、CPU およびメモリのオーバーヘッドを低く抑えて、きわめて複雑な処理が可能になります。

4.2. sRGB エンコーディング

sRGB エンコーディングはガンマ変換を使用してより精度の高い二アゼロを提供し、人間の視覚体系を真似る、ひとつのフォーマットです。sRGB はまた、DXT 圧縮形式と組み合わせれば、アプリケーション

のカラー忠実度を増し、同時に記憶サイズを小さくすることができます。

GeForce 6 シリーズ GPU では、状況に応じて浮動小数点フォーマットを使って、次のような利点が得られます。

- 範囲全体にわたる精度の高さ。
- かなり大きく、リニアでダイナミックな範囲
- フレームバッファのよりリニアなブレンディング

テクスチャの場合、sRGB はほとんどのケースで浮動小数点の方がはるかに適しています。メモリ使用量と帯域幅要求が少ないからです。

4.3. 個別のアルファブレンディング

GeForce 6 シリーズ GPU では、カラーおよびアルファに対して個別のブレンディング関数を指定することが可能です。これにより、たとえば、テクスチャのアルファチャンネルに保存する値に関して柔軟性が増します。アルファを確保しながら、カラーチャンネルを変調するのは、このひとつの使用例です。

4.4. サポートされるテクスチャフォーマット

The following table shows the texture formats supported by GeForce 6 Series GPUs.

整数フォーマット	2D	キューブ	3D	MIP	フィルタ	sRGB	レンダー	ブレンド	頂点
R8G8B8	N	N	N	N	N	N	N	N	N

A8R8G8B8	Y	Y	Y	Y	Y	Y	Y	Y	N
X8R8G8B8	Y	Y	Y	Y	Y	Y	Y	Y	N
R5G6B5	Y	Y	Y	Y	Y	Y	Y	Y	N
X1R5G5B5	Y	Y	Y	Y	Y	Y	Y	Y	N
A1R5G5B5	Y	Y	Y	Y	Y	Y	N	N	N
A4R4G4B4	Y	Y	Y	Y	Y	Y	N	N	N
R3G3B2	N	N	N	N	N	N	N	N	N
A8	Y	Y	Y	Y	Y	N/A	N	N	N
A8R3G3B2	N	N	N	N	N	N	N	N	N
X4R4G4B4	N	N	N	N	N	N	N	N	N
A2B10G10R10	N	N	N	N	N	N	N	N	N
A8B8G8R8	N	N	N	N	N	N	N	N	N
X8B8G8R8	N	N	N	N	N	N	N	N	N
G16R16	Y	Y	Y	Y	Y	N	N	N	N
A2R10G10B10	N	N	N	N	N	N	N	N	N
A16B16G16R16	N	N	N	N	N	N	N	N	N
A8P8	N	N	N	N	N	N	N	N	N
P8	N	N	N	N	N	N	N	N	N
L8	Y	Y	Y	Y	Y	N	N	N	N
L16	Y	Y	Y	Y	Y	N	N	N	N
A8L8	Y	Y	Y	Y	Y	N	N	N	N
A4L4	N	N	N	N	N	N	N	N	N

Float フォーマット	2D	キューブ	3D	MIP	フィルタ	sRGB	レンダー	ブレンド	頂点
R16F	N	N	N	N	N	N/A	N	N	N
G16R16F	Y	Y	Y	Y	Y	N/A	Y	N	N
A16B16G16R16F	Y	Y	Y	Y	Y	N/A	Y	Y	N
R32F	Y	Y	Y	Y	N	N/A	Y	N	Y
G32R32F	N	N	N	N	N	N/A	N	N	N
A32B32G32R32F	Y	Y	Y	Y	N	N/A	Y	N	Y

シャドウマップ	2D	キューブ	3D	MIP	フィルタ	sRGB	レンダー	ブレンド	頂点
D24X8	Y	N	N	Y	Y	N/A	Y	N/A	N

D24S8	Y	N	N	Y	Y	N/A	Y	N/A	N
D16	Y	N	N	Y	Y	N/A	Y	N/A	N

4.5. 浮動小数点テクスチャ

GeForce 6 シリーズ GPU は浮動小数点テクスチャのサポートを強化しました。次の表に、成分当たり 16 ビット (16fp) と成分当たり 32 ビット (fp32) 浮動小数点テクスチャの両方についてサポートされている種々の機能を示します。

テクスチャ成分タイプ	ニアレストフィルタリング	バイリニアおよびトライリニアフィルタリング	異方性フィルタリング	ミップマップサポート	3D テクスチャ	Cube Maps	2 の累乗でないテクスチャ
16 ビット	Yes	Yes	Yes	Yes	Yes	Yes	Yes
32 ビット	Yes	No	No	Yes	Yes	Yes	Yes

4.5.1. 制限

R16F フォーマットはサポートしていません。代わりに G16R16F を使ってください。さらに、ブレンドできるのは A16B16G16R16F サーフェスだけで、G16R16F や R32F サーフェスはブレンドできません。ただし、G16R16F テクスチャのフィルタリングはサポートされています。

4.6. 複数レンダーターゲット（MRT）

GeForce 6 シリーズ GPU は MRT をサポートし、これによってピクセルシェーダは最大 4 つの異なるターゲットにデータを書き出すことができます。MRT はピクセルシェーダが 5 つ以上の float 値を計算し、これらの中間結果をテクスチャに保存する必要があるときはいつも便利です。

MRT の使用例として、位置と速度を同時に計算する素粒子物理学やそれに類似した GPGPU アルゴリズムがあります。遅延シェーディングは複数の float4 値を同時に計算する別の技法です。これは、たとえば、サーフェス法線、拡散およびスペキュラーマテリアルプロパティなどすべてのマテリアルプロパティを計算して個別のテクスチャに保存する別の技法です。これらのプロパティは、それに続くパスで複数のライトでシーンを照明するときに使用されるようになります。

DirectX のキャップスビット NumSimultaneousRT は、グラフィックスデバイスが同時にレンダリングできるレンダ-ターゲットの数を示します。GeForce 6 シリーズ GPU の場合、キャップスビットは 4 です。MRT を有効にするには、SetRenderTarget(index, pRenderTarget) API 呼び出しを使います。この呼び出しは、レンダーターゲットテクスチャに渡されたものを特定のインデックスにバインドします。それからピクセルシェーダがバインドされたレンダーターゲットテクスチャに、oC0、oC1、oC2、および oC3 レジスタを使って出力します。1 から 3 までのインデックスのレンダーターゲット

を NULL にリセットして MRT レンダリングをオフにするのを忘れないでください。

MRT は他の GPU 機能を制限します。最も重要なのは、ハードウェアアンチエイリアシングが MRT レンダーターゲットに適用できないことです。さらに、すべてのレンダーターゲットは同じ幅、高さおよびビット深度を持たなければなりません。GeForce 6 シリーズの場合、それは、32 ビットフォーマット内 (すなわち、A8R8G8B8、X8R8G8B8、G16R16F および R32F) で自由に組み合わせられる、すなわち、各 64 ビットで (すなわち、A16R16G16B16F フォーマットを使用) 最大 4 つのレンダーターゲットを使える、あるいは各 128 ビットで (すなわち、A32R32G32B32F フォーマットを使用) 最大 4 つのレンダーターゲットを使える、ということになります。

さらに、ポストピクセルシェーダブレンド演算アルファブレンディング、アルファテスト、フォギング、およびディザリングは、D3DMISCCAPS_MRTPOSTPIXELSHADERBLENDING キャプスビットがセットされ、USAGE_QUERY_POSTPIXELSHADERBLENDING のレンダーフォーマットクエリーが Yes を返す場合は、MRT でしか使用できません。

GeForce 6 シリーズのチップは、R32F、G16R16F および A32R32G32B32F を除き、すべての MRT フォーマットについてこの機能をサポートします。したがって、GeForce 6 シリーズ GPU (GeForce 6200 は除く) は A16R16G16B16F 浮動小数点レンダーターゲットのポストブレンド演算をサポートしているということです。DirectX 仕様はさらにこれらの MRT ポストブレンド演算の動作について修正されています。MRT レンダリングはディザリング状態を無視し、レンダーターゲットゼロについてのみフォグ状態を優先し (1 から 3 までのレンダーターゲットはフォグが無効になっているかのよ

うに動作する）、アルファテストは oC0.a の値だけを使用してすべての 4 つのレーダーターゲットピクセルを放棄すべきかどうか判断します。

最後に、MRT を使用することは、性能上の含みがあります。MRT は関連フレームバッファ帯域幅の点で大きなコストがかかります、特に、幅の広いビット深度を使うときです。たとえば、4 つの A32R32G32B32F サーフェスのレンダリングは、ひとつの A8R8G8B8 のレンダリングに比べ、16 倍もフレームバッファ帯域幅を消費します。さらに、GeForce 6 シリーズは、3 つ以下のレンダーターゲットを同時に使うとき、性能上のスイートスポットがあります。

そのため、次の一般的な性能に関するアドバイスが適用されます。MRT は必要なときのみ使用する、すなわち、MRT が複数のパスを保存するとき。レンダーターゲットの数とそのビット深度を最小限にする、たとえば、データを密にパックし、アルファチャンネルを無駄にしない。必ず早期に MRT レンダーターゲットを割り当てる（セクション 3.6.4 メモリ割り当て、参照）。

また、MRTの出力を、パスの総数が増えない範囲で、3 つ以下のグループに分割します。たとえば、アプリケーションがひとつの周辺のパスをレンダリングし、その後に、4 つのMRTに出力されるパスが続く場合、周辺のパスのレンダリング中はターゲットの内ひとつを出力し、それから 3 つのMRTだけに出力することを考えます。この方法はターゲットのひとつが他より低精度で保存できる場合、特に有効であり、他のターゲットと無関係に簡単に計算できます（たとえば、マテリアル拡散テクスチャマップ）。当社のSDKのバージョン 7.1 の遅延シェーディングについて詳細は次のURLから入手できます、また、遅延シ

ェーディングのデモクリップもダウンロードできます。
ftp://download.nvidia.com/developer/Movies/NV40-LowRes-Clips/Deferred_Shading.avi.

4.7. 頂点テクスチャリング

GeForce 6 シリーズ GPU は頂点テクスチャリングをサポートしますが、頂点テクスチャを定数 RAM として扱うことはありません。頂点テクスチャは、実際の定数読み込みと違って、データをフェッチする際に待ち時間が発生します。したがって、頂点テクスチャを使用する最もよい方法は、テクスチャをフェッチし、その後に多くの算出演算を続け、テクスチャフェッチの結果を使う前に、待ち時間を隠蔽することです。

頂点テクスチャは大きな定数配列の代替ではありません。少数の頂点単位のデータ向けに設計されていますから、頂点ごとの頂点テクスチャフェッチの数はほんの少数にすぎません。

詳しくは、当社の「Using Vertex Textures」ホワイトペーパーをお読みください。これはつぎのURLから入手できます。
http://developer.nvidia.com/object/using_vertex_textures.html.

4.8. 一般的な性能上のアドバイス

GeForce 6 シリーズアーキテクチャには、効率を上げ、用途を広げる多数の改善点が含まれています。ここでは、その能力を活用するのに役に立つヒントをいくつか取り上げます。

- **書き込みマスクとスウィズルを使う。**GeForce 6 シリーズシェーダのアーキテクチャは 4 成分ベクトルの部分を異なる単位でスケ

ジュールすることができ（co-issue と dual-issue を通じて）、これによってシェーダ稼働率が改善されます。書き込みマスクとスウィズルを使うことによって、コンパイラがこれらのタイプのスケジュール機会を確定する役に立てられます。

- **可能な場合はいつも中間精度を使用。**GeForce 6 シリーズ GPU で中間精度を使う理由は 2 つあります。ひとつは、GeForce 6 シリーズは特別なフリーの fp16 正規化ユニットをシェーダに持っており、これによって 16 ビット浮動小数点正規化を他の計算と平行して非常に効率的に実行することができます。これを利用するには、プログラムで適切な場合はいつでも中間精度を使うようにするだけです。もうひとつの理由は、中間精度はレジスタの負担を軽減する役に立ち、高性能につながる可能性があることです。
- **分岐が十分論理的な場合はダイナミック分岐を使う。**セクション Error! Reference source not found.で触れたように、ダイナミック分岐はコードを高速にでき、実装も容易です。しかし、それを最適に動作させるには、分岐が十分論理的でなければなりません（たとえば、ラフに 30 x 30 ピクセルの領域以上）。

4.9. 法線マップ

アプリケーションで法線マップの保存に問題がある場合は、半球の再マッピングとして知られる手法を使って成分の内のひとつを保存しなくてもよいようにして、単位長さ接線空間法線の法線マップ圧縮を行うことができます。

```
N.z = sqrt( 1 – N.x*N.x – N.y*N.y );
```

これは 3~5 のピクセルシェーダ命令にコンパイルされますから、この手法はこれらの命令が他の（前に存在している）シェーダ命令と共同発行（co-issue）できるか、およびテクスチャのフェッチがボトルネックになっているかいないかによって、性能向上に対する効果が変わります。

半球の再マッピングを行うことにした場合、GeForce 6 GPU で好ましいテクスチャフォーマットは、DirectX では D3DFMT_V8U8、OpenGL では GL_LUMINANCE8_ALPHA8 です。これらのフォーマットは 16 ビット/ピクセルで、2:1 の可逆圧縮が可能です。

半球の再マッピングは、正の単位長さの法線だけが生成されるため、柔軟性が少し損なわれます。負の値（オブジェクト空間法線マッピングなど）や非単位法線（ http://developer.nvidia.com/object/mipmapping_normal_maps.htmlに説明のあるアンチエイリアシング手法など）に依存する手法は不可能です。

法線マップの詳細については、「Bump Map Compression」ホワイトペーパーをご覧ください。これは次のURLから入手できます。 http://developer.nvidia.com/object/bump_map_compression.html.

ローポリモデルをハイポリモデルのように見せる高品質の法線マップを作成するには、NVIDIA Melodyを使用してください。ローポリワーキングモデルをロードしてから、ハイポリ参照モデルをロードし、「Generate Normal Map（法線マップ生成）」ボタンをクリックして、あとはMelodyがうまくやるのを見るだけです。Melodyは次のURLから入手できます。 http://developer.nvidia.com/object/melody_home.html.

第5章
GeForce FX プログラミングのヒント

この章では GeForce FX ファミリーの機能を 100%活用するために役に立つヒントをいくつか提供します。これらは、一部は性能にも影響しますが、ほとんどは機能重視のヒントです。

5.1. バーテックスシェーダ（Vertex Shaders）

強力な GeForce FX バーテックスエンジンでは 1 秒あたり 200 万を上回るトライアングルを処理できます。フルダイナミック分岐がサポートされたており、シェーダの作成者は、光、骨などの数と種類に基づいて分岐することが可能です。

アクティブな各分岐スレッドは、プログラム全体の実行速度を落とすことになります。そのため、大きなバーテックス計算を回避したり、

あるいは API を通じてプリミティブバッチサイズを大きくするためにのみ分岐を行ってください。

5.2. ピクセルシェーダ（Pixel Shader）の長さ

GeForce FX はもともと、パスあたり Direct3D では 512、OpenGL では 1,024 のピクセル命令を処理できます。DirectX では、ps_2_a あるいは ps_2_x プロファイル向けにコンパイルしてください。OpenGL では、GLSL、Cg を使用でき（arbfp1 または fp30 プロファイル向けにコンパイルすることによって）、あるいは ARB_fragment_program 拡張を直接使用できます。

Quadro FX カードはパスあたり 2,048 のピクセル命令を処理することができます。

5.3. DirectX 固有のピクセルシェーダ

最新の DirectX 9 ps_2_0 およびそれ以降のシェーディングモデルでは、デフォルトで 24 ビット以上の精度で数値計算および一時計算を行う必要があります（GeForce FX ファミリーはこのケースでは 32 ビット float 型を使用しています）。アプリケーションからアセンブリ言語で_pp 修飾子を指定して 16 ビット浮動小数点精度にすることができます。

HLSL または Cg を使用する場合は、非常に簡単にこれを行うことができます。32 ビット精度は float 型として、16 ビット精度は half 型として変数を宣言します。

先ず一番便利な方法でシェーダを書き、それから必要に応じてレジスタの使い方や half 精度の最適化を行うことをお勧めします。

固定小数点ブレンディングは、主に、固定機能テクスチャブレンディングパイプラインやシェーダモデル ps_1_0 から ps_1_4 のテクスチャ演算セクションに使用されます。

DirectX では、ps_2_0 を通じて固定小数点精度を要求することはできません。プログラムを ps_1_1 から ps_1_4 に適合させられる場合は、固定小数点シェーディングハードウェアの使用頻度が増えるため、たいていは実行速度が上昇します。

5.4. OpenGL 固有のピクセルシェーダ

ARB_fragment_program 拡張ではデフォルトで最低 24 ビット浮動小数点精度が必要です。ARB_fragment_program ソースコードの先頭に種々のフラッグを立てて精度をコントロールすることができます。

- NV_fragment_program。half (16 ビット浮動小数点) および fixed (12 ビット固定小数点) フォーマットを明示的に使用してコントロールと性能を最大化できます。
- ARB_precision_hint_fastest。統合コンパイラ (Unified Compiler) が実行時に各シェーダに適した精度を判断し、アーチファクトを最小に抑え、全体性能を向上します。

- ARB_precision_hint_nicest。プログラム全体を強制的に float 精度で実行させます。

5.5. 16 ビット浮動少数点を使う

多くの開発者が以前に half を扱った経験がなく、また float は「ちょっと役に立つ」ひとつのフォーマットと見なし、その範囲と精度にはほとんど注意を払っていません。half は 10 ビットの仮数と 5 ビットの指数を持ちます。float は 23 ビットの仮数と 8 ビットの指数を持ちます。仮数ビットはそれぞれ定規の目盛りと見ることができます。この目盛りによってそのフォーマットの最大精度が決定されます。half の場合、各仮数ビット間の精度は 0.1%です。指数値は定規の長さとしてみることができます。指数が大きくなって定規が長くなると、定規の各目盛りの長さの単位が大きくなります。このようにして浮動小数点フォーマットは自動的に精度と範囲の交換をします。

half では、小数ビットは残されず、-2048 から 2048 までの整数と正確に表すことしかできません。ピクセルシェーダでビュー座標あるいはワールド座標で計算する場合、精度から外れることがあります。例えば、キャラクタの位置が 4096、光の位置が 4097 の場合、両方のキャラクタは同じ 16 ビット浮動小数点数で表されます。これらの値を減算すると、ゼロになります。それから、その結果を 2 乗して正規化すると、無限大となります。回避方法は単純、簡潔で、オリジナルのシェーダよりも高速でさえあります。簡単で、マトリクスおよびベクトル減算演算をバーテックスシェーダに移すだけです。

最初に GeForce FX CPU を使うとき照明計算の問題によって half フォーマットの誤用につながることがよくあります。これは驚くことではありません。half 型は、カラー用のフィルムには広く使用されていますが、ゲームでは一般的ではないからです。

バーテックスシェーダは、大きなワールド座標およびビュー座標スペースを処理できるように、最低限 float をサポートする必要があります。また、そもそもなぜリニア計算がバーテックスシェーダにふさわしいのかは簡単にわかります。それが頂点でしか計算できず、CPU に負担をかけずに反復できる場合、なぜピクセルごとに何かを再計算するのでしょうか？

したがって、バーテックスシェーダで、頂点を照明空間または接空間に移動し、その結果の位置をピクセルシェーダに渡すことを勧めます。ひとつのよい方法として、バーテックスクスシェーダでの頂点の位置から照明位置を減算する方法があります。それから、均一にベクトルに定数を掛けて、その値をゼロに近づけ、フラグメントシェーダでベクトルを正規化します。（均等なスケールは正規化された結果には影響しません）。

接空間でなんとか照明しようとすることがよくあります。これはそんな場合に特に有用です。なぜなら、座標系の中心を頂点におくことになるからです。ほぼゼロにすることによって符号ビットも使えるようになり、使える仮数ビットがひとつ増えます。

一般的には、CPU で通常計算を行い、リニア計算はバーテックスシェーダで、非リニア計算はピクセルシェーダで行います。

5.6. サポートされるテクスチャフォーマット

整数フォーマット	2D	キューブ	3D	MIP	フィルタ	sRGB	レンダー	ブレンド	頂点
R8G8B8	N	N	N	N	N	N	N	N	N
A8R8G8B8	Y	Y	Y	Y	Y	Y	Y	Y	N
X8R8G8B8	Y	Y	Y	Y	Y	Y	Y	Y	N
R5G6B5	Y	Y	Y	Y	Y	Y	Y	Y	N
X1R5G5B5	Y	Y	Y	Y	Y	Y	Y	Y	N
A1R5G5B5	Y	Y	Y	Y	Y	Y	N	N	N
A4R4G4B4	Y	Y	Y	Y	Y	Y	N	N	N
R3G3B2	N	N	N	N	N	N	N	N	N
A8	N	N	N	N	N	N/A	N	N	N
A8R3G3B2	N	N	N	N	N	N	N	N	N
X4R4G4B4	N	N	N	N	N	N	N	N	N
A2B10G10R10	N	N	N	N	N	N	N	N	N
A8B8G8R8	N	N	N	N	N	N	N	N	N
X8B8G8R8	N	N	N	N	N	N	N	N	N
G16R16	Y	Y	Y	Y	Y	N	N	N	N
A2R10G10B10	N	N	N	N	N	N	N	N	N
A16B16G16R16	N	N	N	N	N	N	N	N	N
A8P8	N	N	N	N	N	N	N	N	N
P8	N	N	N	N	N	N	N	N	N
L8	Y	Y	Y	Y	Y	N	N	N	N
L16	Y	Y	Y	Y	Y	N	N	N	N
A8L8	Y	Y	Y	Y	Y	N	N	N	N
A4L4	N	N	N	N	N	N	N	N	N

Float フォーマット	2D	Cube	3D	MIP	フィルタ	sRGB	レンダー	ブレンド	頂点
R16F	N	N	N	N	N	N/A	N	N	N
G16R16F	Y	Y	Y	Y	N	N/A	Y	N	N

A16B16G16R16F*	Y	N	N	N	N	N/A	Y	N	N
R32F	Y	Y	Y	Y	N	N/A	Y	N	N
G32R32F	N	N	N	N	N	N/A	N	N	N
A32B32G32R32F*	Y	N	N	N	N	N/A	Y	N	N

シャドウマップ	2D	Cube	3D	MIP	フィルタ	sRGB	レンダー	ブレンド	頂点
D24X8	Y	N	N	Y	Y	N/A	Y	N/A	N
D24S8	Y	N	N	Y	Y	N/A	Y	N/A	N
D16	Y	N	N	Y	Y	N/A	Y	N/A	N

*ラッピングまたはミップマッピング機能のない DX9.0c で照射

5.7. DirectX で ps_2_x と ps_2_a を使用する

GeForce FX は導関数計算（DDX と DDY による）、長いシェーダ、プレディケーション（predication）サポートを含む、ps_2_0 機能を超えた機能をサポートします。この機能は高レベルシェーディング言語から、2 通りの方法で利用できます。ひとつは、ps_2_x プロファイルと共に HLSL または Cg を使用してコンパイルする方法で、上記機能を使いますが、特定機能セットのビットをチェックしません。もっとよい方法は、ps_2_a プロファイルを使うことです、これは GeForce FX ファミリーのシェーディング機能と正確に適合するプロファイルで、より最適化されたコードを生成します。

5.8. 浮動小数点レンダーターゲットを使用する

GeForce ファミリーは、64 ビットと 128 ビットの 4 要素浮動小数点レンダーターゲット、および 1 要素と 2 要素の 32 ビット浮動小数点レンダーターゲットとミップマップされたテクスチャをサポートします。

OpenGL では、浮動小数点テクスチャは NV_float_buffer 拡張を通じて照射され、複数の低精度要素を、NV_fragment_program にある pack/unpack 命令を使用して単一の大きい精度の要素にパックできます。アプリケーションは NEAREST_MIPMAP_NEAREST 浮動小数点キューブマップとボリュームテクスチャを pack/unpack 命令を使用して普通の RGBA8 テクスチャでエミュレートできます (サポートは単一の fp32 または fp16 に限定されます) 。

GeForce FX ハードウェアでは、浮動小数点レンダーターゲットはブレンディングをサポートせず、テクセルあたり 32 ビットより大きい浮動小数点テクスチャはミップマッピングやフィルタリングをサポートしません。

5.9. 法線マップ

GeForce FX、GeForce4、および GeForce3 GPU は、2 要素法線マップから法線マップ半球を再マッピングする専用のハードウェアを持っています。このような GPU の場合は CxV8U8 フォーマットを使って

ください。このフォーマットを使う方がシェーダで Z を導き出すよりも高速だからです。

法線マップの詳細については、次の当社のバンプマップ (Bump Map) 要約ホワイトペーパーを参照してください。 http://developer.nvidia.com/object/bump_map_compression.html.

ローポリモデルをハイポリモデルのように見せる高品質な法線マップを作成するには、NVIDIA Melodyを使用してください。ローポリワーキングモデルをロードしてから、ハイポリ参照モデルをロードし、「Generate Normal Map (法線マップ生成) 」ボタンをクリックして、あとはMelodyがうまくやるのを見るだけです。Melodyは次のURLから入手できます。 http://developer.nvidia.com/object/melody_home.html.

5.10. より新しいチップとアーキテクチャ

GeForce FX は様々なアーキテクチャのモデルが様々な価格で入手可能です。同じファミリーの製品は同じバーテックスシェーディングおよびピクセルシェーディング機能を持っています。異なるのは内部性能特性だけですが、これは開発者には見えません。これらの機能により、開発者は、GeForce FX 以降をターゲットにして簡単にゲームを最適化することが可能で、たとえば、ジオメトリ LOD や画面の解像度だけでスケーラビティに対処できます。

GeForce 6 シリーズ GPU は float がかなり高速ですが、高性能であるため half も引き続きサポートしています。このシリーズはまた、float および half 型の浮動小数点レンダーターゲットおよび浮動小数点テクスチャに関して、固定小数点のものに比べてより適合します。

5.11. 要約

GeForce FX および GeForce 6 シリーズは、業界で最も柔軟性のあるシェーダ機能を持ち、これは長いシェーダプログラムから真の導関数計算にまで及びます。しかし、GeForce FX ハードウェアでは、純粋の浮動小数点シェーダは固定小数点および浮動小数点シェーダを一体化したものほど高速ではありません。

多くのシェーダについて、GeForce FX アーキテクチャで最高の性能を発揮する最良の方法は、ps_1_*および ps_2_*シェーダを混合して使用することかもしれません。たとえば、ピクセル単位のライティングの場合、ps_1_1 シェーダで拡散照明をし、ps_1_4 あるいは ps_2_0 シェーダで別のパスでスペキュラーを行う方が高速な場合があります。

第6章
一般的なアドバイス

この章では複数の GPU ファミリーにわたって利用可能な GPU のプログラミングに関する一般的なアドバイスを示します。

6.1. GPU の識別

過去には開発者は GPU のデバイス ID をよく問い合わせて（Windows を通じて）、実行している GPU が何かを確認したものです。デバイス ID はこれまで単純に増加していました。しかし、GeForce 6 シリーズの GPU では、これはもう当てはまりません。キャプスビット（DirectX の場合）あるいは拡張文字列（OpenGL の場合）を信頼して、実行中の GPU の機能を確認することを勧めます。OpenGL のレンダラ文字列を使用している場合は、NV-40 ベースのチップはすべてその名前に「FX」の名前を持っているわけではないことをお忘れなく（「GeForce 6xxx」とか「Quadro FX 3400」という名前が付いています）。

デバイス ID は開発者がサポートの電話を減らそうとして使うことがよくあります。デバイス ID を誤用すると、逆にサポートの電話が増えることになります。よくあることですが、新しい GPU を作るとき、それを認識せず、動かないアプリケーションが多くあります。

あまり強調できませんが、デバイス ID を認識しなければ何も情報が得られない、という考え方があります。デバイス ID を認識しないだけで、あまり思い切った操作を行なわないようにしてください。

あまり強調できませんが、デバイス ID を認識しなければ何も情報が得られない、という考え方があります。デバイス ID を認識しないだけで、あまり思い切った操作を行なわないようにしてください。

GPU を TNT クラス GPU と誤認したため、あるいはデバイス ID を認識しないために、GeForce 6 シリーズ GPU で動作しないゲームもあります。こうなるとサポートは悪夢のような状態になります、NV4X 世代のチップは最も機能が豊富なチップですが、一部のゲームはコーディングがまずいために動かないからです。

デバイス ID はキャップス文字列や拡張文字列の代替にはなりません。キャップスは、種々の理由から、やがて変わります。おそらく、キャップスはやがて動作しますが、仕様が厳しくなり明確になるため、あるいは単に、あるドライバやハードウェア能力を維持するのが困難またはコストがかかるという理由から、機能しなくなっていくこともあるでしょう。

レーダーターゲットおよびテクスチャフォーマットも次から次に消えていくでしょうから、必ずサポート状況をチェックしてください。

デバイスIDに関して問題がある場合は、当社のDeveloper Relationsグループに連絡してください。アドレスは、devrelfeedback@nvidia.comです。

すべてのNVIDIA GPUのデバイスIDの現在のリストは、こちらにあります。 http://developer.nvidia.com/object/device_ids.html。

6.2. ハードウェアシャドウマップ

GeForce 3 以降の GPU の NVIDIA ハードウェアは、OpenGL および DirectX でハードウェアシャドウマッピングをサポートしています。「ハードウェアシャドウマッピング」とは、特別なトランジスタを、特にシャドウマップ深度比較とパーセンテージクローザフィルタリング演算専用にしたということです。高品質なフィルタリングを施されたシャドウマップエッジを非常に効率的に生成しますから、この機能を利用されることを推奨します。ハードウェアシャドウマッピング専用のトランジスタを使用しているため、ps_2_0 以降で当社のシャドウマッピングアルゴリズムをエミュレートする場合には、性能と品質が落ちるでしょう。

ゲームエンジンでシャドウマップを使っている場合は、次の URL をご覧になってください。

- http://developer.nvidia.com/object/hwshadowmap_paper.html
- http://developer.nvidia.com/object/cedec_shadowmap.html
- http://developer.nvidia.com/object/d3dshadowmap.html
- http://developer.nvidia.com/object/Shadow_Map.html
- SIGGRAPH 2002 の *Perspective Shadow Maps*

- *Perspective Shadow Maps: Care and Feeding* in *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics* (http://developer.nvidia.com/GPUGems)

Simon Kozlov氏によるGPU Gamesの「Perspective Shadow Maps: Care and Feeding」では、パースペクティブシャドウマップを実際面で生かす改善点が説明されています。当社はKozlov氏のこの章にある概念を取り入れ、概念実証としてそれを当社のエンジンに実装し、それが現実の状況で非常に効果があることを確認しました。このサンプルはバージョン 7.0 以降のSDKから入手できます。次のURLから入手できます。
http://download.nvidia.com/developer/SDK/Individual_Samples/samples.html

DirectX では、次の方法でハードウェアシャドウマップを作成できます。

1) D3DUSAGE_DEPTHSTENCIL を使用してテクスチャを作成
2) フォーマットは、D3DFMT_D16、D3DFMT_D24X8 (または D3DFMT_D24S8 とするが、お使いのシェーダではステンシルビットにアクセスできません)
3) GetSurfaceLevel(0)を使って IDirectDrawSurface9 インターフェースを取得
4) SetDepthStencilSurface()で Z バッファとして Surface ポインタをセット
5) DirectX ではカラーレンダリングターゲットの設定も必要になるが、D3DRS_COLORWRITEENABLE をゼロにセットしてカラー書き込みを無効にできる。
6) シャドウキャスティングジオメトリをシャドウマップの z バッファにレンダー

7) このステップで使用するビュー-プロジェクションマトリクスの保存をオフにする。

8) レンダ-ターゲットと z バッファをメインシーンバッファに切り替えて戻す

9) シャドウマップテクスチャをサンプラにバインドし、テクスチャ座標を次のように設定。

$$V' = Bias(0.5/TexWidth, 0.5/TexHeight, 0) * Bias(0.5, 0.5, 0) * Scale(0.5,0.5,1) * ViewProj_{saved} * World * Object * V$$

マトリクスは CPU で連結でき、連結されたトランスフォーメーションはバーテックスシェーダで、あるいは固定関数パイプラインのテクスチャマトリクスを使用して、適用できます。

10) 固定関数パイプラインまたは ps_1.0-1.3 を使用する場合は、投影フラッグを D3DTTFF_COUNT4 | D3DTTFF_PROJECTED に設定。

11) ピクセルシェーダ 1.4 以降を使用する場合は、シャドウマップサンプラから投影テクスチャフェッチを行なう。

12) ハードウェアはシャドウマップテクスチャ座標の投影 x および y を使ってテクスチャを参照する。

13) シャドウマップの深度値をテクスチャ座標の投影 z 値と比較する。テクスチャ座標深度がシャドウマップ深度より大きければ、フェッチに返す結果は 0 (シャドウ) 、それ以外は、結果は 1 となる。

14) シャドウマップサンプラの D3DFILTER_LINEAR をオンにすると、ハードウェアは 4 つの深度比較を行い、ひとつのサンプルと同じコストの場合、結果にバイリニアフィルタをかけるが、これは見た目をよくするだけです。

15) この値を使ってライティングで変調する

初期の NVIDIA デバイス (バージョン 45.23 以前) は、シャドウマップは投影されるものと暗黙に想定しました。この動作は NVIDIA ドライバ 52.16 以降で変わり、プログラマは現在は、適切なテクスチャステージフラッグを明示的に設定する必要があります。特に、ps.2.0 シャドウモデルでシャドウマップを使うには、投影テクスチャ索引を明示的に発行する必要があります (たとえば、tex2Dproj(ShadowMapSampler, IN.TexCoord0).rgb) 。手動で w-divide を行い、非投影テクスチャルックアップをそれに続けて同じコマンドをエミュレートしても、うまくいきません。たとえば、tex2D(ShadowMapSampler, IN.TexCoord0/IN.TexCoord0.w)は機能しません。

同様に、ps1.1-1.3 シェーダモデルを使うときは、ドライババージョン 52.16 以降では、テクスチャステージがシャドウマップをサンプリングするために、投影フラッグを明示的に設定する必要があります (たとえば、SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_PROJECTED) 。

注 : ForceWare 61.71 以降では、テクスチャ命令 (tex2D、tex2Dlod など) はシャドウマップと正常に機能します。

当社のSDKに、DirectXおよびOpenGLの両方でハードウェアシャドウマッピングを設定する簡単な例が含まれています。これは次のURLから入手できます。
http://download.nvidia.com/developer/SDK/Individual_Samples/samples.html

第7章
NVIDIA SLI および Multi-GPU の性能に関するヒント

この章では、NVIDIAのSLI技術など、複数GPU構成でアプリケーションの性能を最大限に引き出すのに役立つ有用なヒントをいくつか紹介します。詳細については、次の資料もご覧ください。
ftp://download.nvidia.com/developer/presentations/2004/GPU_Jackpot/SLI_and_Stereo.pdf.

7.1. SLI とは？

SLI (「スケーラブルリンクインターフェース」（Scalable Link Interface）の意味）は複数の GPU が共同で高性能を実現することを可能にします。SLI 認定マザーボードは、x16 物理レーン（これらのスロットのそれぞれが PCI Express グラフィックスボードを挿入可能）を 2 つ備えた PCI Express マザーボードです。2 つのグラフィックスボードは外部 SLI ブリッジコネクタを介してリンクされ、ドライバはこの構成を認識し、SLI Multi-GPU モードに入ることができます。SLI Multi-GPU モードでは、ドライバが両方のボードを単一のデバイスとして構成し、2 つの GPU は、グラフィックスアプリケーションからは、単一の論理デバイスと見えます。

単一の論理デバイスは単一の GPU より 1.9 倍高速です、ドライバがレンダリングの負荷を 2 つの物理的 GPU に分割するからです。SLI モードでの実行には利用可能ビデオメモリが倍にはならないことに留意してください。たとえば、2 つの 256MB グラフィックボードを差し込んでも、せいぜい 256MB ビデオメモリのひとつのデバイスなのです。ドライバは通常、両方の GPU にわたってビデオメモリを複製するのがその理由です。すなわち、いつでも、GPU 0 のビデオメモリは GPU 1 のビデオメモリと同一のデータを含むということです。

SLI システムでアプリケーションを実行するときは、ドライバがどのモードで実行するかを決定します。互換モード、AFR（alternate frame rendering）、あるいは SFR（Split Frame Rendering）。

互換モードは GPU をひとつだけ使ってすべてのレンダリングを行ないます（すなわち、2 番目の GPU はずっとアイドル状態）。このモードでは性能の増加は見られませんが、アプリケーションの SLI との互換性は確保されます。

AFR の場合、ドライバは GPU 0 でフレーム n のすべてを、GPU 1 でフレーム n+1 のすべてをレンダリングします。フレーム n+2 は GPU 0 でというようになります。各フレームが自給型である限り（すなわち、フレームがほとんどデータを共有しない）、AFR は最も効率的です。なぜなら、バーバーテックス、ラスタライズ、およびパーピクセルワークなどのすべてのレンダリングワークが複数の GPU にわたって分割されるからです。もし一部のデータがフレーム間で共有されていると（たとえば、前にレンダーされたテクスチャを再使用する）データは GPU 間で転送する必要があります。このデータ転送が通信オーバーヘッドを生み、フルに 2 倍のスピードアップ実現の妨げになります。

SFR の場合、ドライバはフレームの上部を GPU 0 に、下部を GPU 1 に割り当てます。フレームの上部のサイズと下部のサイズがロードバランスされます。上部の方が下部よりもレンダリングワークが少ないため、GPU 0 の利用度が下げられると、ドライバは、両方の GPU に均等な負荷を与えようとして上部を大きくします。シーンを上部と下部に、それぞれ GPU 0 と GPU 1 に対して、クリッピングすると、フレームのすべての頂点を両方の GPU で処理するのを避けようとします。

SFR モードはそれでもデータ共有が必要です、たとえば、レンダーツーテクスチャ（render-to-texture）演算のため。AFR は一般に通信オーバーヘッドが少なく、SFR よりも頂点ロードバランスがうまくできますから、AFR は好ましいモードです。ただし、たとえば、ア

プリケーションがバッファリングするフレーム数を 1 つに制限する場合など、AFR が適用できないこともあります。

開発中のアプリケーションは、現行のドライバ (66.93 以降) では互換モードがデフォルトです。アプリケーションの開発者は、ドライバにアプリケーション固有のプロファイルを作成して、SFR にすることができます。ディスプレイドライバコントロールパネルで、GeForce タブをクリック、[Performance & Quality Settings]を選択、[Add Profile]をクリックして、アプリケーションの実行ファイルの名前を入力します。それから、[Show advanced settings]を有効にして、[Multi-GPU]をクリックします。

将来のドライバでは、種々の SLI モードをオンにするために、簡単に使えるコントロールパネルと API オプションが追加されます。

SLI システムで最大の性能を実現するために、やるべきこととやってはならないことのリストを次に示します。

7.2. CPU ボトルネックを回避する

アプリケーションが実行中の CPU の拘束を受けている場合は、より強力なグラフィックスソリューションを使ってもほとんど影響はありません。複数 GPU 構成を生かすためには、CPU がボトルネックになるのを避けなければなりません。このプログラミングガイドの第 2 章に、CPU ボトルネックを検出し回避する方法について説明があります。

同様に、アプリケーションがそのフレームレートを人為的に、任意の固定値に抑えている場合は、フレームレートはその値を超えることができません。そのようなことのないようにしてください。

7.3. デフォルトで VSync を無効にする

VSync を有効にすると、フレームレートがモニタのフレッシュレートに抑えられてしまいます。複数 GPU 構成では、グラフィックス性能が高いため、モニタのフレッシュレートよりフレームレートが（かなり）高速なのが普通です。このような高速なフレームレートは、VSync をオフにしなければ達成されません。

トリプルバッファリングはフレームレートを高速にするソリューションとはなりません。トリプルバッファリングは単に、グラフィックスアダプタがレンダリングできる追加バックバッファを割り当てるだけです。

トリプルバッファリングを使うと、グラフィックスアダプタは、ラウンドロビン式に、最大 3 つのバッファにレンダーできます。しかし、グラフィックスアダプタが常に、モニタのフレッシュレートより高速に各バッファのレンダリングを終える場合は、バックバッファの数は関係ありません。モニタのリフレッシュがフレームレート全体のゲートになってしまいます。

一層悪いことに、トリプルバッファリングは 2 つの重要な欠点を持っています。

1. より多くのビデオメモリを消費する（画面解像度が高く、アンチエイリアシングが有効になっている場合、消費される追加ビデオメモリの量は非常に大きい）。
2. 処理中のフレームが増えると、レンダリングコマンドを発行してそれを画面で見られるまでの遅延が大きくなります。

Therefore, turning VSync off by default is the only possible solution. To do so in DirectX, the `PresentationInterval` member of the `D3DPRESENT_PARAMETERS` structure has to be set to `D3DPRESENT_INTERVAL_IMMEDIATE` when calling `IDirect3D9::CreateDevice()`.

7.4. 遅延を少なくとも 2 フレームに抑える

DirectX ではドライバで最大 3 フレームまでバッファリングが可能です。フレームのバッファリングは、CPU と GPU が互いに独立して動作し、最大の性能を発揮できるようにするためには望ましいことです。一方、バッファリングされるフレームが増えると、それだけコマンドの発行からその結果を画面で見るまでの時間が長くかかります。人間は 30ms という短い遅延時間まで感知できることから（テストシナリオによるが）、この遅延は一般に望ましくありません。

そのため、一部のゲームでは、バッファリングするフレーム数を人為的に抑えています。たとえば、バックバッファを口ックして、強制的に CPU と GPU のハードウェア同期をさせています。バックバッファを口ックするとまず CPU をストールさせ、すべてのバッファを空にし、それから GPU をストールさせます。ロック終了時に、すべて

のシステムがアイドル状態になり、バッファリングされたフレームの数がゼロになります。

このようなやり方でシステムをストールさせると、一方で性能に重大な影響を及ぼすことになるので、特に、複数 GPU 構成では、避けるべきです。

バッファリングするフレームの数を抑える、より抵抗の少ないソリューションは、各フレームの終わりで、コマンドストリームにトークンを挿入することです（たとえば、DirectX のイベントクエリー）。これらのイベントが追加レンダリングコマンド発行前に消費されたことをチェックして、バッファリングされるフレームの数を抑え、結果的に遅延を 1 から 3 フレームの間に抑えるのです。

Multi-GPU システムはバッファリングされるフレームの数に特に敏感です。一般に、n 個の GPU を搭載したシステムは、最大の効率を得るために、最低限 n フレームバッファリングする必要があります。

驚いたことに、そうしても遅延が大きくならないのです。なぜなら、デュアル CPU システムは一般にシングル GPU システムより 2 倍高速だからです。たとえば、デュアル GPU システムで 2 フレームをバッファリングすると 30ms の遅延ですが、これはシングル GPU システムで 1 フレームバッファリング（レンダリングに 30ms かかる）するのと同じ遅延です。

したがって、何個の GPU が利用可能かをアプリケーションでチェックし、必要であれば、バッファリングされるフレームの数を少なくとも GPU の数に抑えることを推奨します。nvCPL.dll API を使えば、システムが SLI モードにあるのか、および現在使用されている GPU の数を問い合わせることが可能です。特に、NvCplGetDataInt(NVCPL_API_NUMBER_OF_SLI_GPUS, &number)

関数は、システムで SLI イネーブル GPU の数を返します。NVControlPanel_API.pdf ドキュメントおよび NVSDK のサンプル NvCpl に詳細な説明があります。

7.5. 使用しているすべてのフレームにあるレーダーターゲットテクスチャをすべて更新

マルチ GPU システムの効率は GPU が共有するデータの量に反比例します。GPU は何もデータを共有せず、同期化オーバーヘッドが無く、したがって効率が最大というのが、最高の条件です。

共有データを最小限に抑えるためには、レンダリングされたフレームがそれぞれ、すべての前のフレームに依存しない必要があります。特に、レンダーツーテクスチャ手法を使用するときは、ひとつのフレームで使用されるすべてのレンダーターゲットテクスチャもその同じフレームの間に生成されることも望ましいということです。逆に言えば、フレーム一つおきにだけレンダ-ターゲットを更新し、しかもすべてのフレームでテクスチャとしてその同じレーダーターゲットの使用を避けるということです。

アプリケーションが明示的にレンダ-ターゲットの更新をスキップしてシングル GPU システムでレンダリングスピードを上げている場合は、そのアルゴリズムを複数 GPU 構成向けに修正すると有利な場合があります。たとえば、アプリケーションが複数の GPU で実行されているのかを調べて、もしそうなら、フレームごとにレンダ-ターゲ

ットを更新するか（すなわち、視覚忠実度を増すため）、あるいは 1 行に 2 フレーム、レンダ-ターゲットを更新してそれから 1 行に 2 フレーム、更新をスキップします。

もう一つの方法として、早期にレンダ-ターゲットをレンダリングし、フレーム後半でその結果をつかうだけというのも、SLI システムには有利です。こうすれば、ひとつの GPU がもう一方の GPU のレンダーツーテクスチャ演算の結果待ちに入ってストールするのを避けられます。

7.6. レンダ-ターゲットとフレームバッファをクリアする

ハードウェアは、レンダ-ターゲットが使用前にクリアされていないと、アプリケーションがレーダーターゲットのピクセルごとに更新するのか知るすべがないので、レンダリングの結果をマルチ GPU システムの GPU 間で共有する必要があります（セクション 0 同様に)。使用前にレンダ-ターゲットをクリアすれば、ドライバとハードウェアはこの同期化が必要ないことを知ることができます。

7.7. D3DPOOL_MANAGED で頂点バッファを割り当てる

セクション 0 同様に、マルチ GPU システムは複数の GPU 間で頂点バッファを共有します。DirectX の下の頂点バッファを D3DPOOL_DEFAULT で割り当てると、D3DPOOL_DEFAULT で割

り当てるのと比較して、関連するオーバーヘッドが少なくなります。このオーバーヘッドの減少は、ダイナミック頂点バッファの場合に最も効率的で、特に部分的に更新されたダイナミック頂点バッファではそうです。

第8章
立体ゲームの開発

このセクションでは NVIDIA のステレオレンダリング実装の動作について、およびそれをアプリケーションでフルに活用する方法について説明します。

8.1. なぜステレオなのか？

ゲームの開発者は、ゲームのリアリズムに対する際限の無い欲求の中で、ひとつの要素を見落としがちです。現実世界では人は 2 つの眼で見ていることがそれです。人工的な立体視（画面対実生活で）は巨大マーケットではないが、多くのゲイマーが、NVIDIA のステレオオーバーライドドライバについてくる安いシャッターめがねを付けてゲームをし、存在感を楽しんでいます。

さらに、開発中にステレオでゲームを見るのも役に立ちます。偽物のように見えるものがすぐにわかります。モーションの視差はステレオに同様の視覚的きっかけを与えますが、ステレオビューアはユーザが

回りを動く場合何を見るかを即座に感知し、モーションの視差を介して深度情報を取得します。

ゲームの開発中に立体視を使うのは非常に有効なことです。視覚的な欠陥を見つけて修正してからゲームが出るからです。これはもちろんステレオでゲームをする人たちにとっても体験感を高めることになります。

8.2. ステレをの動作

NVIDIA 3D Stereo Driver は DirectX および OpenGL ベースのゲームでフルスクリーンの立体視を可能にします。3D Stereo Driver ドライバは、シャッターめがねに適したページフリップビューはもちろん、赤と青の立体レンダリングもサポートしています。互換ハードウェアを使えば、深度を感じられる画像が見られるでしょう。機能するためには、Stereo Driver バージョンが Display Driver バージョンと一致しなければならないことに留意してください。

立体ドライバを有効にして 3D ゲームをするとき、シーンは 2 つの視点からレンダリングされ、それぞれ、左右の眼からくるかのように、現実の視点の側に少し寄ります。これは固定関数レンダリングとバーテックスシェーダの両方と協力して行なわれます。

正確なステレオ効果を得るためには、開発者は次の挙げる多くの問題点を考慮する必要があります。

8.3. ステレオの障害になるもの

ステレオにマイナスの影響を及ぼす共通の問題と、それに対する対処について説明します。

8.3.1. 間違った深度でレンダリング

これは、最初に気を付けるべき問題です。3D Stereo Driver は深度を使用してステレ効果を生成しますから、正しい深度でないものがあると、ステレオで見たとき、周囲と違うためひどく目立ちます。

- バックグランド画像、スカイボックス、スカイドームをできるだけ遠くの深度に配置する。そうしないと、ステレオではそれが小さなボックスにあるような世界に見えます。
- HUD アイテムを適当な 3D 深度に置く。オブジェクトの上をホバリングするネームラベルがある場合は、それらをオブジェクトの 3D 深度に置くと、視錐台（view frustum）の二アプレーンに置くよりも、ステレオ効果が良くなります。
- また HUD をできるだけシーンの奥にレンダーするのも有益です。このトリックにより、HUD を見たときに眼の疲れを起こさないで、残りのシーンで感知深度が大きくなります。
- レーザー照準器、十字照準、カーソルは、それらが位置を合わせるオブジェクトの深度で 3D ワールドに置かないと、正しく見えません。そうしないと、それらを使うのはほぼ不可能です。なぜなら、ユーザの眼はひとつの深度で収束しますが、カーソルは別の深度にあり、また、ユーザにはどちらの点も正確な場所にない 2 つのカーソルが見えるからです。

- オブジェクトのハイライトは、オブジェクト自体の深度で行なうべきで、画面空間ではありません。

8.3.2. ビルボード効果

ビルボード効果は平面的に見えて、通常の 3D では良くありません。ステレオでは一層悪く見えます。通常の 3D では、ビルボードは動きにあわせて見えますが、3D ステレオでは、静止シーンでさえ、すぐに問題が出てきます。ほとんどのビルボード効果がステレオではきわめて平面的に見えるので、可能な場合は必ず実際のジオメトリを使いましょう、低解像度のジオメトリでもその方が良く見えます。

粒子効果ビルボード (スパーク、スモーク、ダストなど) については、OK であったり、また OK でないこともあります。最も良いのは、ステレオのアプリケーションでどう見えるかテストして、その品質が十分良いか判断し、そのビルボードが 3D で有意な深度を持つことを確認することです。

8.3.3. ポストプロセスおよび画面空間効果

2D 画面空間効果はステレオ効果のおおきな障害になることがあります。ブラーグロー、ブルームフィルタ、および画像ベースのモーションブラーがこのカテゴリに入ります。これらの効果は、通常、3D ジオメトリをテクスチャにレンダリングしてから、画面と四角に並んだ 2D 画面をレンダリングして生成されます。テクスチャのジオメトリは、その効果に対して、もはや正しい深度にはなく、そのため 3D Stereo ではうまく機能しません。

ステレオでゲームをする人のために、これらの効果を無効にするオプションを提供し、ジオメトリをバックバッファにレンダーすべきでしょう。

8.3.4. 3D シーンで 2D レンダリングを使う

2D としてレンダリングされたオブジェクトは実際にはリアルな 3D 深度を持っていません。したがって、それはモニタ深度に依存します。これは 3D Stereo では非常に平面的に見えます。HUD で 2D と 3D をミックスしている場合は、深度が一貫しないため、眼の疲労につながります。繰り返しますが、3D で、適切な深度ですべてをレンダリングし、ステレオをオンにしてテストしてください。

8.3.5. サブビューレンダリング

ピクチャインピクチャ表示、カーミラー、あるいは上方角のスモールマップなどサブビューを画面にレンダリングするときは、レンダリングする前の領域をカバーするように視点をセットしなければなりません。このトリックにより、妙なステレオ効果が、画面の意図したセクションの外側に流れ出るのを防止できます。

8.3.6. 汚い矩形で画面が更新される

変更された画面の部分だけを確認して、残りの部分を更新していない場合、3D ステレオで妙なレンダリングを起こすことがあります。こんなときは、フレームごとにすべての可視オブジェクトをレンダリングするだけで解消します。

8.3.7. 過剰な分離によるコリジョンを解消する

オブジェクトを互いに押しのけることによってコリジョンの解消をはかっている場合は、それらをあまり遠くに押しのけないようにしてください。動き回るとき通常の 3D では見栄えが悪くなりますが、ステレオではすぐに浮き出して、グランドの上をホバリングしているように見えます。

8.3.8. シーンの差分オブジェクトごとに深度範囲を変える

画面を複数の深度範囲に分割すると、ステレオ効果にひずみが発生することがあり、オブジェクトが短く見えたり、長く見えたりします。最適なステレオ効果を得るには、オブジェクトはすべて均一な深度範囲でレンダリングすべきです。

8.3.9. 頂点に深度データを渡さない

ソフトウェアトランスフォームおよびライティングのために D3D レンダリング用に頂点を送るときは、ステレオが適切に機能するように、RHW 深度情報を含めます。

8.3.10. ウィンドウモードでレンダリング

NVIDIA の 3D Stereo は、アプリケーションがフルスクリーン専用モードのときにしか作動しません。フルスクリーンモードをサポートしない場合は、ゲームプレイヤーは 3D Stereo を利用できません。

8.3.11. シャドウ

フルスクリーンシャドウカラークワッドを使ってステンシルシャドウをレンダリングしてもステレオではうまく機能しません。しかし、シーンのシャドウオブジェクトを適切な深度で再レンダリングすると、ステレオで正しく機能するようになります。シャドウマップは良好に機能し、投影シャドウはシャドウに対して適切な深度に投影している限り、機能します。

8.3.12. ソフトウェアレンダリング

NVIDIA 3D Stereo ドライバは DirectX と OpenGL を自動的にサポートします。他の API を使って 3D ジオメトリをレンダリングしている場合は、ステレオではレンダリングされません。

8.3.13. レンダ-ターゲットに手動で書き込む

レンダ-ターゲットを口ックしないで、直接書き込みをします。そうすればステレオドライバがバイパスされます。

8.3.14. 非常に暗い、またはコントラストが高いシーン

非常に暗いシーンは 3D Stereo シャッターめがねを使うとさらに暗くなります。輝度調整あるいはガンマ調整を行うことでこの問題を抑えることができます。非常に輝度の高いオブジェクトおよび非常に暗いオブジェクト上で非常に輝度の高いオブジェクトを使うとゴーストが発生し、ステレオの障害になります。ステレオでゲームのテストをすれば、これが問題かどうかすぐにわかります。

8.3.15. 頂点間に小さなギャップのあるオブジェクト

メッシュにある小さなギャップは、ステレオでレンダリングすると、一層目立つようになります。メッシュが密であることを確認し、ステレオでテストしてこれが発生していないことを確認してください。

8.4. ステレオ効果の改善

ステレオを使ってもっとリアルな体験感を実現するために使用できるアイデアをいくつか紹介します。

8.4.1. ステレオでゲームのテストをする

すぐれた 3D Stereo 効果を体験できるようにするために、最もよい方法は、ステレオで実行しながらゲームのテストをすることです。たいていの問題がすぐに明らかになり、簡単に修正できます。IODisplay（(http://www.i-glasses.com）から低価格のステレオキットを入手できます。NVIDIA 3D Stereo Driver も赤と青の立体ステレオモード（Red and Blue Anaglyph Stereo Mode）をサポートしますから、一対のペーパー3D めがねを手元に置いておけば、テストは一層簡単になります。

8.4.2. 「モニタから外れる」効果

ニアプレーンに非常に近いが視錐台（view frustum）の端と交差しないものをゲームでデザインすることができます。このデザインを使うとすばらしい「モニタから飛び出す」ステレオ効果を実現できます。ホバリングしている球体、宇宙船、飛んでいるキャラクタなどはその

ほんの一部で。これらはどれもステレオでは実にファンタスティックに見えます。

8.4.3. 高精細ジオメトリ

3D でよりリアリティをだすためにもっとポリゴンを使う。この方法はもちろんいつでも正しいですが、ステレオについては一層の効果があります。建物、植物、木、キャラクタなど、何でも、可能な場合はどこでも、ポリゴンを使いましょう。

8.4.4. 代替ビューを備える

ゲームでユーザが視点を選べるようにしするか、少なくともカメラをコントロールできるようにします。一番手、三番手、トップダウンなど、人によってステレオでは、他より格好がよくなるかもしれません。

8.4.5. ゲームにある最新の問題を調べる

3D Stereo Driver をインストールすれば、そのコントロールパネルが表示され「Stereo Game Configuration」を見ることができます。ここでは、あなたのゲームがリストに載っているか、および、そのゲームに対して当社がすでにどんな問題を発見したかを知ることができます。NVIDIA Developer Relations に連絡して確認することもできます。

8.5. ステレオ API

ステレオのサポートを改善するため現在 2 つの別個の API を開発しています。

- **StereoBLT API – Display Pre-Rendered Stereo Images in 3D**
  立体表示がアクティブな間、作成済みの左右イメージの表示が可能になります。
- **IStereoAPI – Real-time Control Over Stereoscopic Rendering**
  コンバージェンス、ステレオセパレーションなど詳細設定を問い合わせ、コントロールします。この API は、ゲームの中でリアルタイムに設定をコントロールするのに使用できるヘッダとライブラリから構成されています。修正は直ちに反映され、すべてのフレームに対して変更できます。
- **OpenGL Quad-Buffered Stereo.** これは NVIDIA Quadro GPU ファミリーに収録されていますから、特別なステレオドライバは必要なく、ウィンドウモードでも機能します。

8.6. 詳細情報

もっと詳しく知るには、NVIDIA Developer Relationsに連絡するか、あるいは、3DStereoDev@nvidia.comにemailして、詳細情報またはリリース版のStereo APIを請求してください。

また、次のURLからオンラインで最新のStereo Informationを入手できます。 http://developer.nvidia.com/object/3D_Stereoscopic_Dev.html

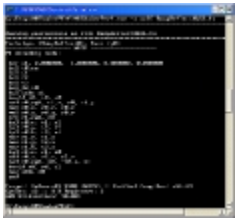
第9章
パフォーマンスツールの概要

このセクションでは、性能上のボトルネックを確定、修正するのに役立つ当社のツールについて説明します。

9.1. NVPerfHUD

NVPerfHUD は DirectX の上部に 4 つの参考になるグラフを表示します。これらのグラフはアプリケーションに関する重要な統計データであり、ボトルネックの可能性を突き止めうえで役に立つものです。グラフは心拍数モニタ形式でサンプルデータを表示します。右から左にスクロールすれば、前の 256 フレームの値を見ることができます。

NVPerfHUDは次のNVIDIA Developer Web Siteから入手できます。
http://developer.nvidia.com/object/nvperfhud_home.html.

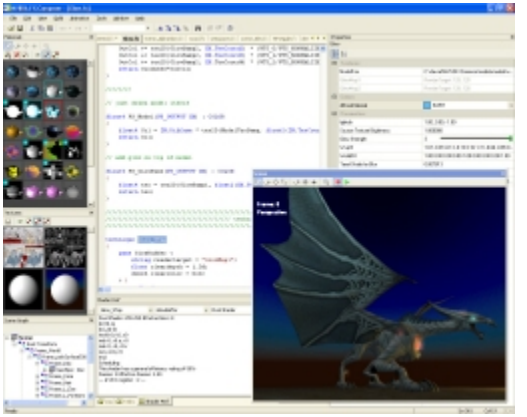


9.2. NVShaderPerf

NVShaderPerf コマンドラインユーティリティは FX Composer の Shader Perf パネルと同じ技術を使用して、シェーダ性能の数的指標をレポートします。これは、HLSL、GLSL、Cg、!!ARBfp1.0、ps_1_x、および ps_2_x で書かれた DirectX および OpenGL シェーダをサポートします。サイクルカウント、レジスタ利用および GPU 稼働率など、GeForce 6 シリーズおよび GeForce FX GPU のすべてのファミリーでシェーダのパフォーマンスレポートを得ることができます。

NVShaderPerfはつぎのURLから入手できます。
http://developer.nvidia.com/object/nvshaderperf_home.html.

9.3. FX Composer



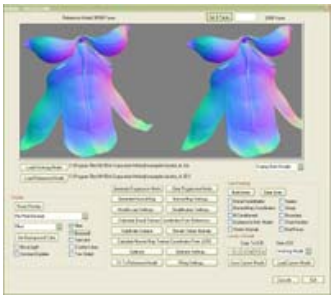
FX Composer は、ユニークなリアルタイムプレビューと最適化機能を備えた統合開発環境を開発者に提供し、高性能シェーダ開発を支援します。FX Composer は、プログラマにはシェーダ開発と最適化を容易にすること、アーチストには特定のシーンに合わせてシェーダをカスタマイズするためのわかり易い GUI を提供することを目標に設計されています。

FX Composer により、先進の解析と最適化でシェーダ性能をチューニングすることが可能です。

- バーテックスシェーダとピクセルシェーダの性能チューニングワークフローを可能にする
- GeForce 6 シリーズおよび GeForce FX GPU のすべてのファミリーについて性能をシミュレートする
- 計算済み関数をテクスチャルックアップテーブルにキャプチャ
- GPU サイクルカウント、レジスタ利用、稼働率、FPS などの経験的な性能上の数的指標を提供。
- 最適化のヒントにより性能上のボトルネックをレポート

FX Composerの最新版は次のURLからダウンロードできます。
http://developer.nvidia.com/fxcomposer.

9.4. NVIDIA Melody



ローポリモデルをハイポリモデルのように見せる高品質な法線マップを作成するには、NVIDIA Melodyを使用します。ローポリワーキングモデルをロードしてから、ハイポリ参照モデルをロードし、「Generate Normal Map（法線マップ生成）」ボタンをクリックして、あとはMelodyがうまくやるのを見るだけです。Melodyは次のURLから入手できます。 http://developer.nvidia.com/object/melody_home.html.

9.5. 開発者ツール 質問とフィードバック

当社のツールに関して皆さんのフィードバックをいただければ幸いです。皆さんのコメントや懸案事項を、sdkfeedback@nvidia.com 宛てに送信してください。