NVPerfHUD が GPU の性能を最大限に引き出す

NVPerfHUD 3.0
性能解析に役立つ警告表示

DU-01231-001_v01
April 2005

DEVELOPMENT

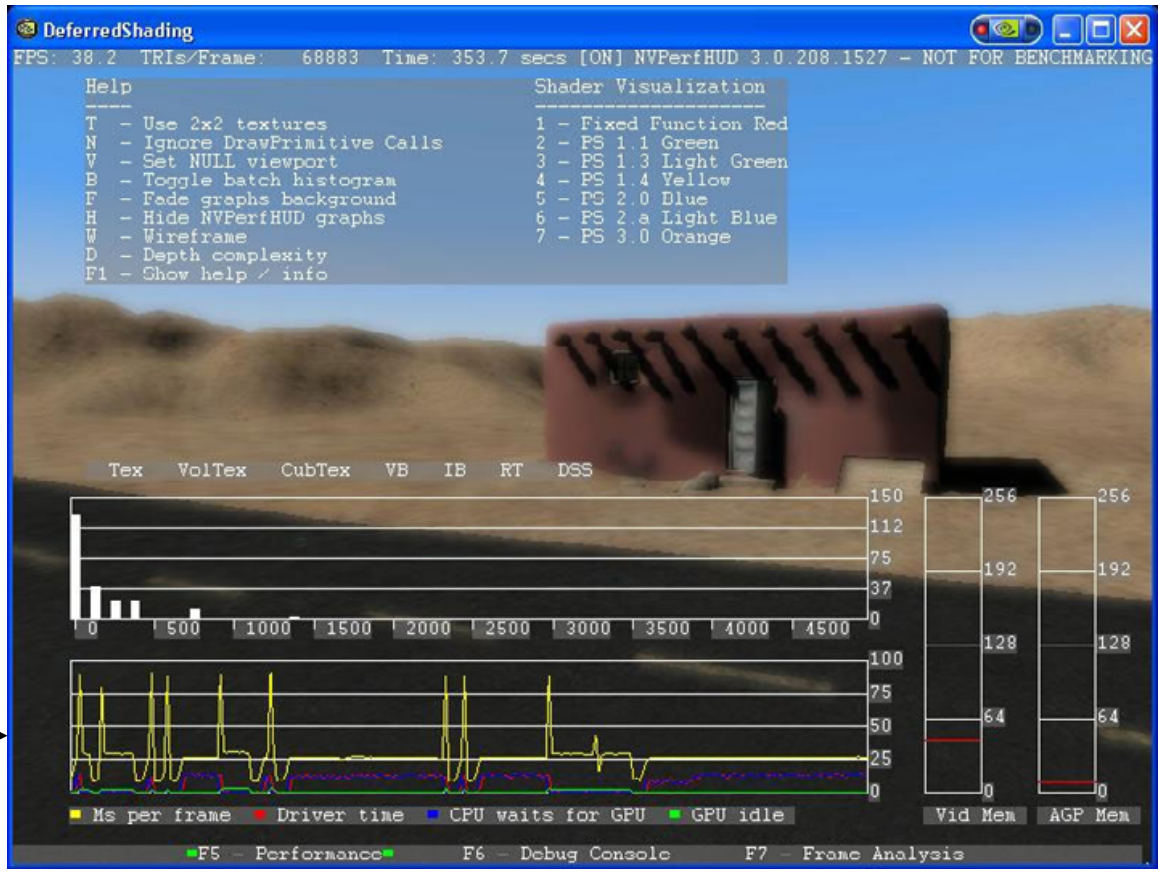
記載内容

第 1 章
NVPerfHUD について

現在のグラフィックス・プロセッシング・ユニット（GPU）では、一連のパイプライン化された処理によって、画像を生成しています。パイプラインの実行速度は最も遅い工程を超えられないので、グラフィクスアプリケーションの性能が最大限に引き出されるようにアプリケーションを調整するには、パイプラインに立脚した性能解析アプローチを採らねばなりません。NVPerfHUD は、グラフィックスパイプライン中の最も遅い工程を特定することで、お手持ちのアプリケーションの総合的なパフォーマンスを向上させます。

NVPerfHUD は、１つの工程のアプリケーション性能を一度に解析することが可能で、Direc3D 9 アプリケーションのあらゆる工程における性能上のボトルネックと機能的異常の診断に使用できる統計値をリアルタイムに表示します。

NVPerfHUD を起動すると、図 1 に示されているようにお手持ちの Direct3D アプリケーションの上部に、グラフィック・オーバーレイが表示されます。



Graph overlay showing performance
性能を示すグラフオーバーレイ

図 1　NVPerfHUD を起動した Direct3D アプリケーション

第 1 節　　必要なシステム

- 何れかの NVIDIA GPU (GeForce 3 以降)
  GeForce 6 シリーズ以降を推奨
  機能が低減されたこれらより旧式 GPU
- NVIDIA ディスプレイ ドライバー71.80 以降
- Microsoft DirectX 9.0c
- Windows 2000 または Windows XP

第 2 節　　推奨される読み物

- NVIDIA Developer Web Site
  http://developer.nvidia.com
  - *Balancing the Graphics Pipeline for Optimal Performance* – ホワイトペーパー [リンク]

- *NVIDIA GPU Programming Guide* – あらゆる最新のtips and tricks [リンク]
- NVIDIA FX Composer – シェーダーの開発環境 [リンク]
- NVShaderPerf – シェーダー性能の解析ユーティリティー [リンク]
- NVIDIA SDK – 何百ものコードサンプルと効果 [リンク]
- NVTriStrip – creates strips that are vertex cache aware [リンク]

- GPU Gems: *Programming Techniques, Tips, and Tricks for Real-Time Graphics* [リンク]
  性能に関する章が特に有用です。
- GPU Gems 2: *Programming Techniques for High-パフォーマンスグラフics and General-Purpose Computation* [リンク]
  Microsoft DirectX ホームページ [リンク]
- Microsoft Developer Network (MSDN) ホームページ[リンク]
  “performance” と“最適化” を検索してください。
- Microsoft DirectX SDK documentation – インストール後のスタートメニューにあります

第２章
準備を整える

NVPerHUD では、GPU およびドライバー自体の中から直接メトリクスを収集する特別な性能モニター用ルーチンをディスプレイドライバー中で使用しています。NVperfHUD は、様々なメトリクスを収集し、お手持ちのアプリケーションと交信させるために API インターセプトも使用しております。これらの計測技術は、NVPerfHUD を適切に機能させ、小さなオーバーヘッド（7%未満）を追加導入するために必要です。

NVPerHUd を用いてお手持ちの Direc3D アプリケーションを始動させると、DirectX グラフィックスの上部に NVPerfHUD ユーザインターフェースが表示されるのをご覧いただけます。起動ホットキーにより、お手持ちのアプリケーションおよび NVPerfHUD とのやり取りを切り換えることができます。

お手持ちのアプリケーションを NVPerfHUD 解析ができる状態にして、下記のクイックスタートの手順に従えば、NVPerfHUD が簡単にご利用いただけます。

第１節　お手持ちのアプリケーションの準備をする

NVPerfHUD は、お手持ちのアプリケーションの内部機能を把握するのに役立つ強力な性能解析ツールです。第三者がお客様のアプリケーションを許可なく不正に解析することのないように、NVPerfHUD 解析を有効化するために若干の変更を行う必要があります。お手持ちのアプリケーションで NVPerfHUD を適切にお使いいただくためのさらに詳しい情報が、本文書の末尾にございますトラブルシューティングの節に記載されています。

Note: 発送する前に、お手持ちのアプリケーション中の NVPerfHUD を必ず無効にしてください。無効にしないと、誰でもアプリケーション上の NVPerfHUD も使用することができます！

グラフィックスパイプラインをセットアップするときに最初にすることの１つは、Direct3D CreateDevice() 関数を呼び出して、ディスプレイデバ

イスを生成することです。お手持ちのアプリケーションでは、以下のような画面が表示されているはずです。

```
HRESULT Res;
Res = g_pD3D->CreateDevice(   D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL,
                          hWnd, D3DCREATE_HARDWARE_VERTEXPROCESSING,
                          &d3dpp, &g_pd3dDevice );
```

お手持ちのアプリケーションが NVPerfHUD によって始動されると、特別な **NVIDIA NVPerfHUD** アダプターが生成されます。お手持ちのアプリケーションは、このアダプターを選択することによって、NVPerfHUD にアプリケーションを解析する許可を与えることができます。さらに、アプリケーションのなかには、**NVIDIA NVPerfHUD** アダプターID を意図せずに選択して、不正な解析に曝されるものがあるので、デバイスタイプとして、必ず“D3DDEVTYPE_REF” を選択してください。NVPerfHUD アダプターを選択されたのであれば、お手持ちのアプリケーションは、実際にはリファレンスラステライゼーを使用していないと思われます。

お手持ちのアプリケーション中での NVPerfHUD 解析を可能とする最小限のコード変更は、次のようなものです。

```
HRESULT Res;
Res = g_pD3D->CreateDevice( g_pD3D->GetAdapterCount()-1,
                            D3DDEVTYPE_REF, hWnd,
                            D3DCREATE_HARDWARE_VERTEXPROCESSING,
                            &d3dpp, &g_pd3dDevice );
```

（上記のように **GetAdapterCount()-1** を呼び出すことによって）最後のアダプターを使用する場合には、NVPerfHUD によって生成される **NVIDIA NVPerfHUD** アダプター識別子がリストの最後にあることが想定されています。

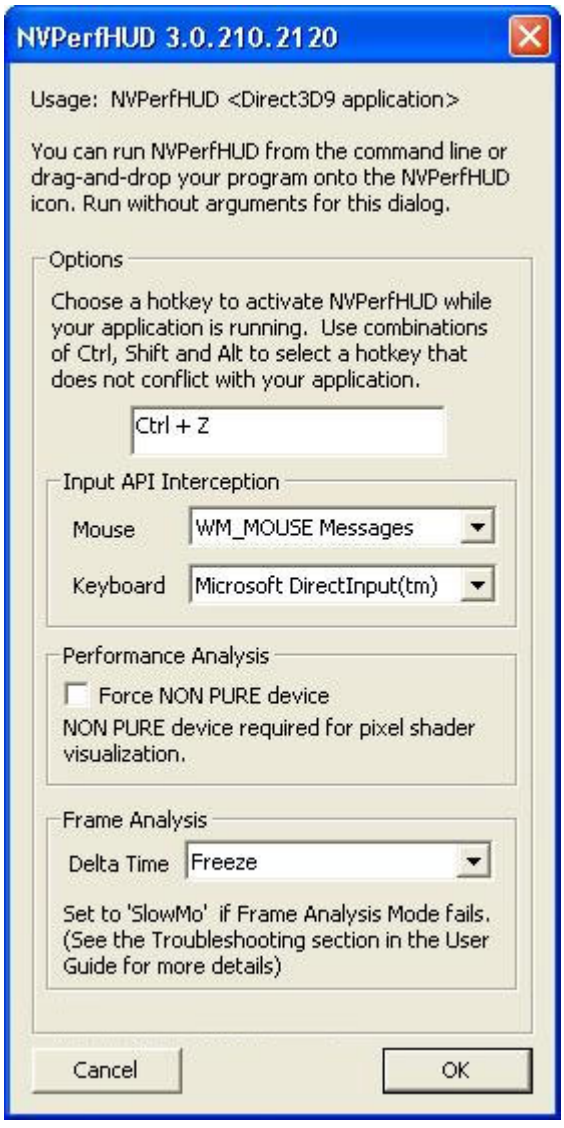
Note: お手持ちのアプリケーションで NVPerfHUD 解析を起動するために必要な操作は、NVIDIA NVPerfHUD アダプターを選択し、**DeviceType** フラグを設定することだけです。これらのパラメータのうち 1 つだけ変更した場合には、NVPerfHUD 解析は実行できないことがあります。

お気づきのように、**NVIDIA NVPerfHUD** アダプターが利用できないときには、この簡略な実装法がお手持ちのアプリケーションにソフトウェアのリファレンスラステライザーを使用させてしまうことがあります。この問題を回避するために、お手持ちのアプリケーション中における **CreateDevice()** への呼び出しを以下のコードに交換することをお勧めします。

```
// Set default settings
UINT AdapterToUse=D3DADAPTER_DEFAULT;
D3DDEVTYPE DeviceType=D3DDEVTYPE_HAL;

#if SHIPPING_VERSION
// When building a shipping version, disable NVPerfHUD (opt-out)
#else
// Look for 'NVIDIA NVPerfHUD' adapter
// If it is present, override default settings
for (UINT Adapter=0;Adapter<g_pD3D->GetAdapterCount();Adapter++)
{
      D3DADAPTER_IDENTIFIER9  Identifier;
```

```
        HRESULT                 Res;

        Res = g_pD3D->GetAdapterIdentifier(Adapter,0,&Identifier);
        if (strcmp(Identifier.Description,"NVIDIA NVPerfHUD") == 0)
        {
                AdapterToUse=Adapter;
                DeviceType=D3DDEVTYPE_REF;
                break;
        }
}
#endif

if (FAILED(g_pD3D->CreateDevice( AdapterToUse, DeviceType, hWnd,
                D3DCREATE_HARDWARE_VERTEXPROCESSING,
                &d3dpp, &g_pd3dDevice) ) )
{
        return E_FAIL;
}
```

これによって、お客様がご使用になりたいときに NVPerfHUD 解析を実行することが可能となり、正常に実行されていれば、お手持ちのアプリケーションはソフトウェアリファレンスラステライザーを使用いたします。下記のクイックスタートの手順に従って、お手持ちのアプリケーションを立ち上げ、NVPerfHUD を実行してください。



第２節　クイックスタート

NVPerfHUD を用いてお手持ちのアプリケーションを実行すると、初期設定のグラフと情報がお手持ちのアプリケーションの上部に表示されます。NVPerfHUD の高度な機能は、下記のセットアッププロセスで用意する起動ホットキーによってご利用いただけます。

1. **NVPerfHUD をインストールする**
   インストーラーが新しいアイコンをデスクトップ上に配置します。

2. **NVPerfHUD を実行する**
   NVPerfHUD Launcher を初めて実行すると、設定ダイアログが自動的に表示されます。解析するアプリケーションを指定しなくても、Launcher を実行すると、常に設定ダイアログをご覧になれます。
3. **起動ホットキーを選択する**
   お手持ちのアプリケーションが使用しているキーとかち合っていないことを確認してください。
4. **API インターセプトを設定する**
   お手持ちのマウスおよびキーボードイベントをキャプチャする方法を NVPerHUD に指示してください。お手持ちのアプリケーションがサポートされていない方法を使用すれば、キーボードおよび／またはマウスを使用することはできません。

> **Note:** 起動ホットキーを使用して NVPerfHUD を起動すると、その後のキーボードイベントはすべて、NVPerfHUD によってインターセプトされます。

5. **オプション**：NVPerfHUD の Force NON PURE デバイス機能を ON にします。
   お手持ちのアプリケーションが PURE デバイスを使用している場合、Force NON PURE デバイスチェックボックスをチェックします。お手持ちのアプリケーションが PURE デバイスを使用しており、ボックスがチェックされていなければ、フレーム解析モードのご使用または Performance Mode でのピクセルシェーダーの可視化のご使用はできません。
6. **お手持ちのアプリケーションを NVPerfHUD デスクトップアイコン上にドラッグ＆ドロップします。**
   OK をクリックして、設定オプションを確認してから、お手持ちの .EXE, .BAT or .LNK (ショートカット)ファイルを NVPerfHUD Launcher Icon 上にドラッグ＆ドロップします。コマンドラインから NVPerHUD.exe を実行し、コマンドライン引数として解析するようにアプリケーションに指示することも可能です。これが自動的に起こるように、バッチファイルを生成するか、IDE 設定を変更することを選択する開発者もいます。
7. **オプション: お手持ちのアプリケーションがフレーム解析モードで強制停止するか、不具合がある場合には、“デルタタイム Delta Time” の設定を SlowMo に変更してください。**これは、問題が NVPerfHUD にあるのか、それともお手持ちのアプリケーションにあるのかを決定するのに役立ちます。さらに詳しいことは、トラブルシューティングの節をご覧ください。

> **Note:** 常に、アプリケーションを指定せずに NVPerfHUD を実行して、設定ダイアログにアクセスするようにしてください。

初期設定の NVPerfHUD グラフと情報がお手持ちのアプリケーションの上部に表示された状態で、アプリケーションが実行されている様子がご覧いただけるはずです。NVPerfHUD とやり取りするために選択した起動ホットキーを使用してから、**F1** を押すと、オンスクリーン・ヘルプが表示されます。再度ホットキーを使用すると、お手持ちのアプリケーションに指示できる状態に復帰します。

Note: お手持ちのアプリケーション中のボトルネックを正確に特定できるようにするため、NVPerfHUD は、PresentationInterval を D3DPRESENT_INTERVAL_IMMEDIATE に設定することで、垂直リフレッシュ同期を強制的にオフにします。

第３節　　基本的な作業の流れ

NVPerfHUD が起動された時点で、グラフィックスパイプライン実験の実行、パフォーマンスメトリックスのグラフの表示、複数の性能可視化モードを用いて潜在する問題の探索を行えます。初期設定の性能解析モード性能解析モードからデバッグ・コンソール・モードデバッグコンソールモード又はフレーム解析モードフレーム解析モードに切り替えることもできます。デバッグ・コンソールでは、DirectX Debug ランタイムからのメッセージ、NVPerfHUD からの警告、お手持ちのアプリケーションからのカスタムメッセージを表示が表示されます。フレーム解析モードでは、グラフィックスパイプライン中の各工程についての詳細を表示してくれる高度な状態検査機能がご利用いただけます。

以下のキーを押すことで、NVPerfHUD を作動させた状態で、これらのモードを切り替えることができます。

F5　**性能解析 Performance Analysis**
タイミンググラフと指定された実験を使用して、ボトルネックを特定します

F6　**デバッグ・コンソール Debug Console**
DirectX Debug ランタイムからのメッセージ、 NVPerfHUD の警告およびお手持ちのアプリケーションからのカスタムメッセージを一覧表示します。

F7　**フレーム解析 Frame Analysis**
Freeze the current frame and step through it one draw call at a time, drilling down to investigate the setup for each stage of the graphics pipeline using the State Inspectors described below.カレントフレームをフリーズして、１回のドローコールで一度にフレームを進め、以下に記載されている State Inspectors を用いて、グラフィックスパイプラインの各工程についてセットップを掘り下げて調査します。

第 3 章
性能解析モード

本章では、NVPerfHUD が性能解析モードで表示した情報の解釈の仕方や、お手持ちのアプリケーションにおける性能上のボトルネックを特定して最適化するために、的を絞った性能実験を使用する方法について解説します。

図 2 は、性能解析モードで使用可能なグラフとオーバーレイを示しています。

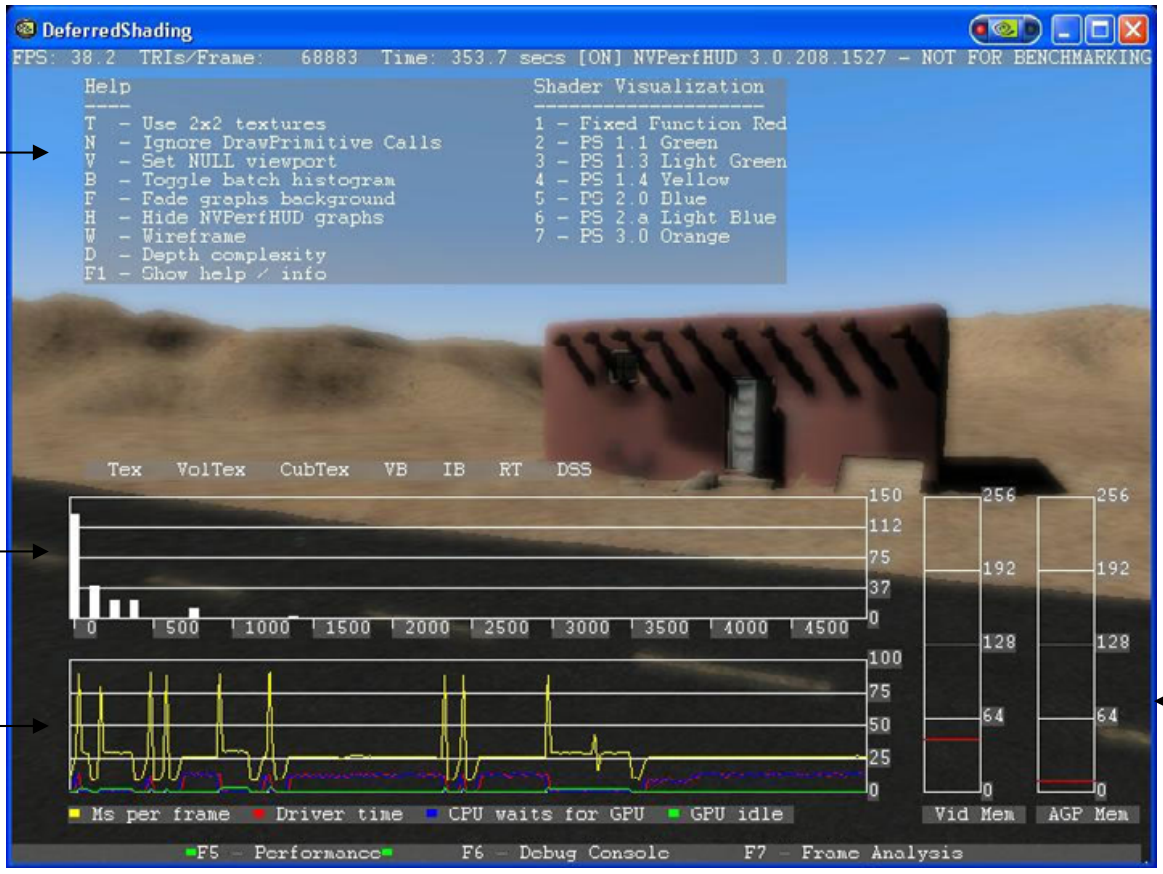


図 2　　NVPerfHUD 性能解析モード

第１節　　パフォーマンスグラフ

NVPerfHUD は、お手持ちのアプリケーションを最初に立ち上げたときに、初期設定で幾つかのグラフと基本的パフォーマンスメトリックスを表示します。

起動ホットキーと以下のオプションを使用することにより、必要に応じて、他のグラフや情報をさらに表示させることができます。

- **F1** お役立ち情報のサイクル表示
- **B** バッチサイズヒストグラムをトグル表示します
- **F** 背景をフェイドさせてグラフを見やすくします
- **H** グラフを隠します
- **W** ワイヤフレーム
- **D** Depth Complexity デプスの複雑性レベル

Note: DirectX Debug Runtime がオンになっているときには、タイミンググラフと directed tests はオフになります。Debug Runtime 環境で付加されるオーバーヘッドと性能特性のため、性能解析を行うための設定には不適切です。警告メッセージが表示されますが、ドローコールグラフ、バッチヒストグラムおよびメモリグラフはご利用いただけます。

第１項　グラフの読み方

本モードで表示されるデータは、以下の情報エリアに分割されます。

FPS および Triangles/Frame

基本パフォーマンスメトリクスが、スクリーンの左隅に表示されます（図２をご覧ください）。これらの数値全体で、お手持ちのアプリケーションが作業を遂行する速度の指標となります。

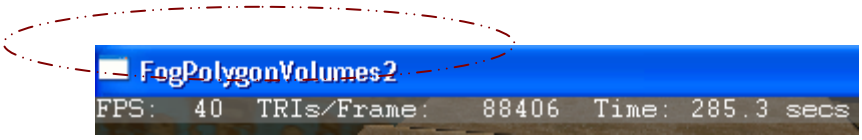


図３　　NVPerfHUD 情報バー（最上部）

グラフをスクロールする

これらのグラフは心拍数のモニターのようなもので、すべての枠を右から左にスクロールすると、経時的な変化をご覧になれます。

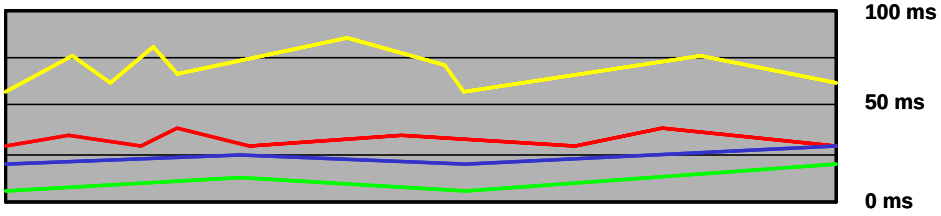


図 4　　パフォーマンスグラフ

- 緑= GPU_IDLE
  GPU がアイドル状態であった総時間量／フレーム
- 青 = DRIVER_WAITS_FOR_GPU
  ドライバーが GPU を待たなければならなかった経過時間の累計
  (このような事態が生じる理由についてさらに詳しくお知りになりたい方は、 付属書類 A をご覧ください)
- 赤= TIME_IN_DRIVER
  CPU がドライバーコードを実行している総時間量／フレーム
  （DRIVER_WAITS_FOR_GPU (青)を含む）
- 黄 = FRAME_TIME
  あるフレームから次のフレームへの経過時間の合計－FRAME_TIME (黄) の線を出来るだけ低くしたい

FRAME_TIME	17ms	34ms	50ms	75ms	100ms
FPS	60	30	20	13	10

Note: FRAME_TIME (黄) 線と TIME_IN_DRIVER (赤) 線の時間差が、アプリケーションロジックと OS によって費やされた時間に相当します。

バックグラウンドで実行されている OS のプロセスによって、スパイクの発生が時折観察されることがあります。下のグラフは、ハードディスクのアクセス、テクスチャのアップロード、OS のコンテクストスイッチなどによって生じるフレームレートの突発的な増加を示しています。

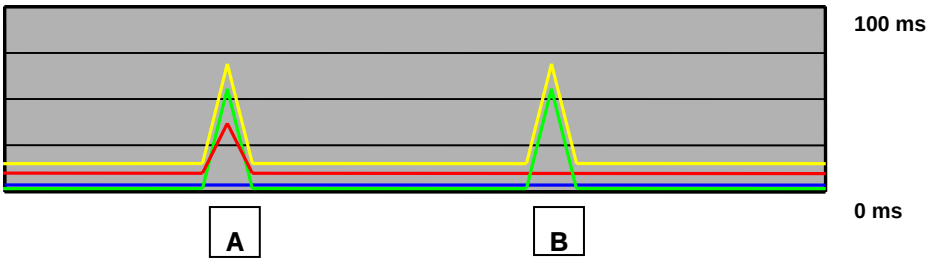


図 5　　時折発生するスパイク

散発的である限り、このような状態は正常ですが、スパイクが発生する理由を把握する必要があります。このような状態が定期的に生じるのであれば、お手持ちのアプリケーションは CPU に負荷がかかる演算が非効率的に行われている可能性があります。

- タイプ **A**:
  TIME_IN_DRIVER (赤) と FRAME_TIME (黄) の線が同時にスパイクしていれば、ドライバーがテクスチャを CPU から GPU にアップローディングしているためである可能性があります。
- タイプ **B**:
  FRAME_TIME (黄) の線がスパイクしているのに、TIME_IN_DRIVER (赤) の線がスパイクしていないのであれば、お手持ちのアプリケーションは、CPU に負荷がかかる何らかの演算（オーディオのデコーディングなど）を行っているか、ハードディスクにアクセスしている可能性があります。かかる状態は、他のプロセスを処理している OS によって生じることもあります。

GPU にデータを送信していないので、緑の線は何れのケースでもスパイクする場合があることにご留意ください。

Primitives Graph 基本グラフを描く

このグラフは、フレーム当たりの `DrawPrimitive`、`DrawPrimitiveUP` 及び `DrawIndexedPrimitives` (DP)コールの数を表しています。性能上のボトルネックを特定するためにこの情報を使用する方法は、次ページのパイプライン実験パイプライン実験および第 6 章で論述されています。

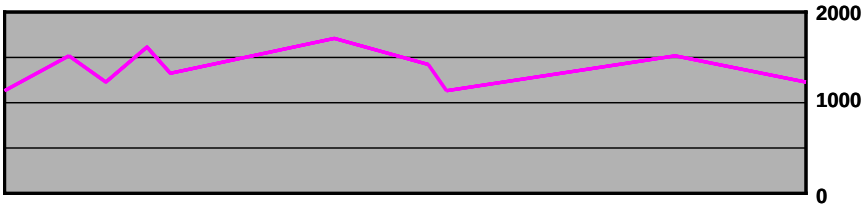


図 6　　DP コールグラフ

Batch Size Histogram バッチサイズヒストグラム

バッチサイズグラフは、（お客様の起動ホットキーを使用して）NVPerfHUD を起動し、**B** キーを押すことにより作動させないと、バッチ

サイズグラフは表示されません。単位は、バッチの数です。第一番目の棒は 0-100 個のトライアングルを有するバッチの数を表しており、以下、第二番目の棒は 100-200 個のトライアングルを有するバッチの数を表しています。小さなバッチを数多く使用している場合には、左側の棒が高くなります。

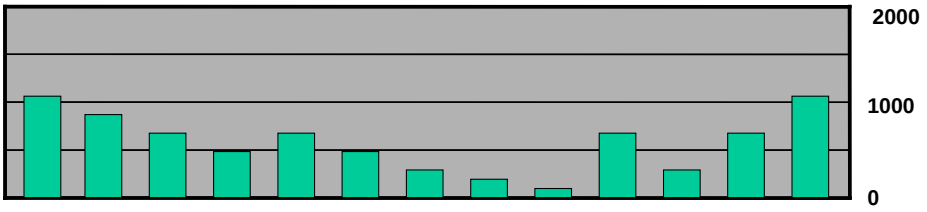


図 7　　DP コールヒストグラム

メモリグラフ

本グラフは、ドライバーによって割り当てられた AGP とビデオメモリのメガバイトを表示しています

図 8　　メモリグラフ

第 2 項　リソース生成モニターリソース生成モニター

リソース生成モニターリソース生成モニターインジケータは、リソースが生成されるごとに点滅します。Direct3D でのリソースの動的な生成は、性能を低下させるのが通常なので、可能であれば常に避けるべきです。

図 9　　リソース生成モニターリソース生成モニター

モニターされるリソース生成イベントの種類は、以下のとおりです。

Tex CreateTexture()を使用して生成された 2D テクスチャ

VolTex CreateVolumeTexture()を用いて生成された Volume テクスチャ

CubTex CreateCubeTexture()を用いて生成された Cubemap テクスチャ

VB CreateVertexBuffer()を用いて生成されたバーテックスバッファ

IB CreateIndexBuffer()を用いて生成されたインデックスバッファ

RT CreateRenderTarget()を用いて生成されたレンダーターゲット

DSS CreateDepthStencilSurface()を用いて生成された Depth stencil surfaces デプスステンシル表面

Note: リソース生成イベントは Debug Console (F6)でも記録されますので、インジケータが点滅している原因を知ることができます。

第 2 節 Pipeline Experiments パイプライン実験

性能上のボトルネックを特定するには、グラフィックスパイプラインの特定の段階に一度に集中する必要があります。NVPerfHUD をご利用いただくことにより、以下の実験を行うことができます。

- **T—テクスチャユニットを単離する**
  テクスチャ帯域幅によってお手持ちのアプリケーション性能が制約されており、フレームレートが劇的に増加する場合に、GPU に対して 2×2 テクスチャを使用するように強制します。
- **V—バーテックスユニットを単離する**
  1x1 のシザー矩形を用いて、バーテックスユニット後のパイプライン段階におけるラスタライゼーションとシェーディングの作業をすべて切り出します。このアプローチは、バーテックスユニット後のグラフィックスパイプラインを切り捨てて近似し、お手持ちのアプリケーション性能がバーテックス変換、CPU 作業および／またはバストランザクションによって制約を受けているかどうかを測定するために使用できます。
- **N—Eliminate the GPU を削除する**
  この機能は、DrawPrimitive() および DrawIndexedPrimitives() コールをすべて無視することで、無限に速い GPU を有している状態を近似します。グラフィックスパイプライン全体に性能上の負担がなければ、これによって、お手持ちのアプリケーションが達成するフレーム速度が近似されます。状態の変化によって生じた CPU のオーバーヘッドも省略されることに注意してください。

バージョンごとにピクセルシェーダーを選択的に無効にして、お手持ちのアプリケーションでピクセルシェーダーがどのように使用されているかを可視化することもできます。特定のシェーダーバージョンが無効化されると、そのグループに属する全てのシェーダーが、同じ色で表示されます。NVPerfHUD に用意されているシェーダーの可視化オプションを以下に列記します。

1 — 固定機能 (赤)
2 — 1.1 ピクセルシェーダー (緑)
3 — 1.3 ピクセルシェーダー (薄い緑)
4 — 1.4 ピクセルシェーダー (黄)
5 — 2.0 ピクセルシェーダー (青)
6 — 2.a ピクセルシェーダー (薄い青)
7 — 3.0 ピクセルシェーダー (オレンジ)

Note: シェーダーの可視化は、NON PURE デバイスとして Direct3D デバイスが生成された場合にのみ作動いたします。NVPerfHUD では、NON PURE デバイスモードでデバイスが生成されるように強制することができます。

第４章
Debug Console Mode デバッグコンソールモード

本章では、デバッグコンソールモードでご利用いただける情報について説明します。Debug Console screen デバッグコンソールスクリーンが 図 10 に示されています。

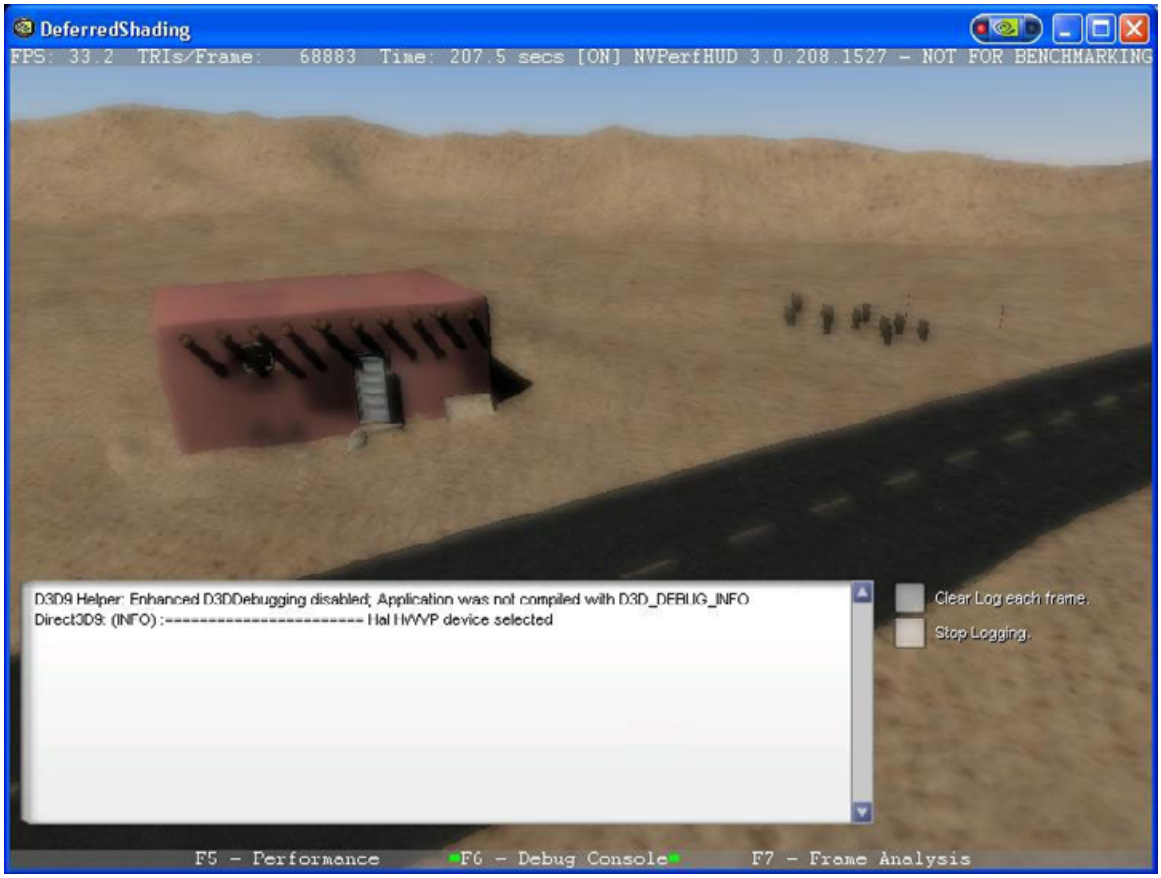


図１０　　デバッグコンソールモード

Debug Console デバッグコンソールには、 DirectX Debug ランタイムを介して報告されるメッセージ、OutputDebugString()関数を介してお手持ちの

アプリケーションに報告されるメッセージ、および NVPerfHUD によって検知されるその他の警告やエラーがすべて表示されます。

NVPerfHUD によって検知されたリソース生成イベントおよび警告は、コンソールウィンドウにも記録されます。

以下のオプションを使用して、Debug Console の作動様式をカスタマイズすることができます。

- **C** Clear Log Each Frame 各フレームのログをクリアする
- **S** Stop Logging ロギングを停止する
- **F** Fade Console コンソールをフェイドする

「Clear Log Each Frame 各フレームのログをクリアする」のチェックボックスをチェックすると、フレームの初めから、コンソールウィンドウのコンテンツがクリアされるので、カレントフレームによって生成された警告のみが表示されます。お手持ちのアプリケーションが、コンソールウィンドウに納まらない警告をフレームごとに生成する場合に、有用です。

「Stop Loggin ロギングを停止する」のチェックボックスをチェックすると、コンソールは新しいメッセージを表示しなくなります。

第５章
フレーム解析モード

本章では、フレーム解析モードを最大限に活用する方法と本モードの高度なグラフィックスパイプライン State Inspectors 状態検査機能について説明します。

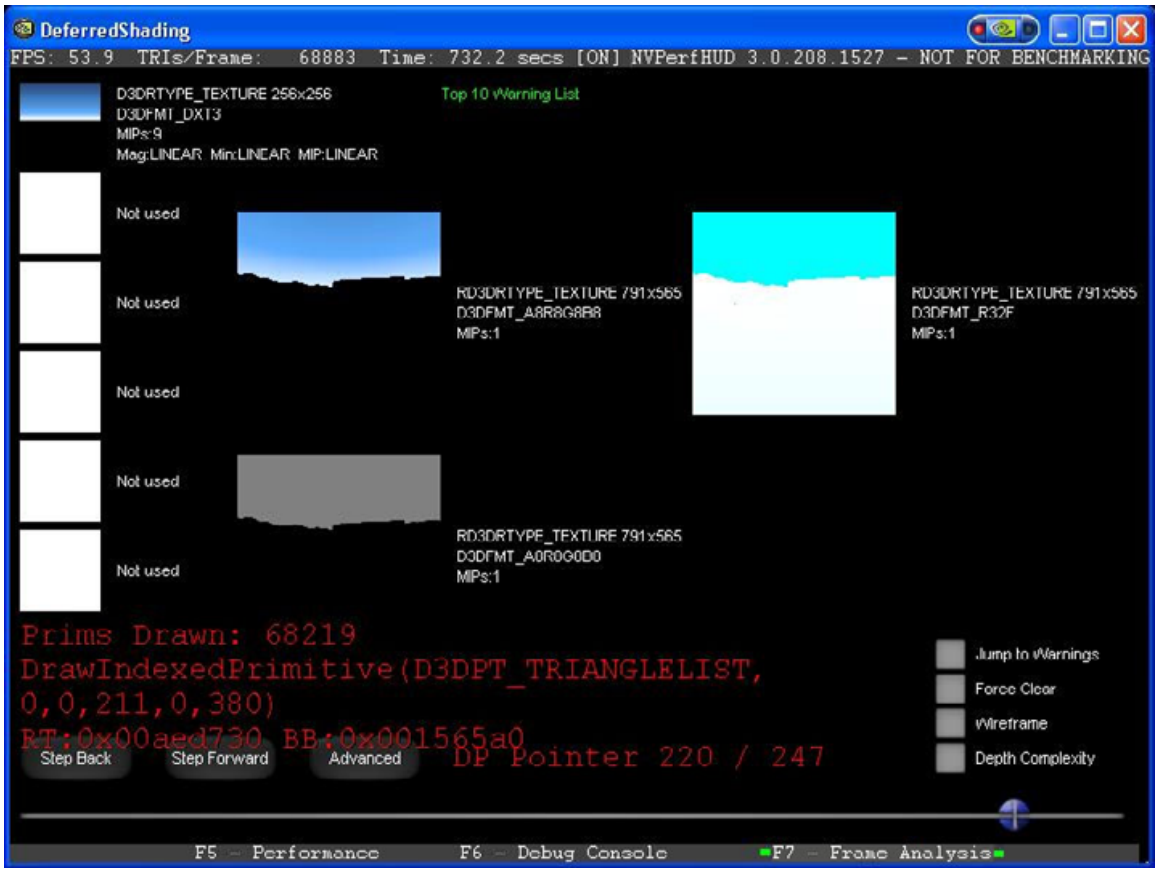


図１１　　フレーム解析モード

左/右矢印キーを使用して、前／次のドローコールおよびフレーム解析モードを設定するための下記のオプションを表示させることができます。

- **A**　拡張ディスプレイ／シンプルディスプレイを交互に切り替え表示します
- **S**　警告を表示します

- **W** Wireframe ワイヤフレーム
- **D** Depth Complexity デプスの複雑性レベル

第 1 節　　レンダリングの分解

レンダリングアーティファクトを有するフレームを特定した場合、シーンが描かれる順序を確認するか、警告を生じた原因をさらに詳しく調べてください。フレーム解析モードを使用することにより、フレーム内のドローコールをすべてナビゲートすることができます。フレーム解析モードに切り替えると、NVPerfHUD がお手持ちのアプリケーション用のクロックを停止するので、カレントフレームをフリーズさせながら、カレントフレームを詳細に解析することができます。お手持ちのアプリケーションがフレームを基礎としたアニメーションを使用していれば、フリーズしている時間はアニメーションされたオブジェクトに対して何らの影響も与えません。

Note: フレーム解析モードを有効に使用するために、お手持ちのアプリケーションは、NVPerfHUD がこのアプリケーションを制御できるように動作しなければなりません。要件が幾つか以下に記載されています。その他の事項については、本文書の末尾にございますトラブルシューティングの節をご覧ください。

フレーム解析モードでは、お手持ちのアプリケーションが QueryPerformanceCounter() または timeGetTime() win32 関数を使用し、これらの関数に依拠していることが必要とされます。特に、dt がゼロのケースでは、お手持ちのアプリケーションは、経過時間(dt)の計算を取り扱う上でロバストでなければなりません。換言すれば、プログラムは dt によって除されるべきではありません。

お手持ちのアプリケーションがフリーズしている際には、**次（右矢印）**および**前（左矢印）**キーボードボタンを使用して、フレーム中のドローコールを全て呼び出すことができます。スクリーンの下にあるスライダーを前後にドラッグするか、または **PgUP/PgDn** を使用して、特定の泥コールについて素早くナビゲーションを行えます。マウスまたはキーボードイベントのインターセプションが適切に機能していなければ、お手持ちのアプリケーションを終了し、NVPerfHUD 設定ダイアログにある別の API インターセプションオプションを選択してください。

各ドローコールに対して表示される情報には、以下のものがあります。

- 直近のドローコールとドローコール回数の合計
- 最後のドローコールの関数名とパラメータ。ドローコールが警告を与えた場合には、警告メッセージも表示されます。

第 1 項　警告を表示する

警告を表示する(S) が起動されると、DP コールによる最も高価なバーテックスバッファ(VB)などの、警告のリストがスクリーンの最上部にあるリストボックスに表示されます。警告をクリックすると、関連の DP コールにジャンプします。**アップ／ダウン矢印**を使用して、リスト全体をスクロールさせることもできます。警告が表示されているときに、**次(右矢印)** または **前(左矢印)** をクリックすると、スクリーン下部のスライダーが、この警告を与えた次／前のドローコールに移動します。

ロックされた状態で消費された時間を把握し、これをできるだけ最小限に抑えるために、各 VB ロックを必ず調べてください。これは、バーテックスバッファをセットアップするために、お手持ちのアプリケーションが CPU 時間を把握するのに役立つはずです。

第 2 項　テクスチャユニットと RTT 情報

各テクスチャユニットとオフスクリーンのテクスチャへのレンダリング (RTT) ターゲットに対して表示される情報は以下のとおりです。

- 各テクスチャに格納されているテクスチャと各々の属性
  - 次元
  - フィルタリングパラメータ：縮小、拡大および MIP レベル
  - テクスチャのフォーマット： RGBA8、DXT1、DXT3、DXT5 など
  - テクスチャのターゲット：1D、2D、ボリュームテクスチャ、キューブテクスチャ、NP2 など
- カレントドローコールがオフスクリーンテクスチャにレンダリング(RTT)を行っている場合、そのテクスチャの内容と属性がスクリーンの中央に表示されます。

スクリーン上に収まらないテクスチャが使用されている場合には、スクロールバーを使用して、リスト全体をナビゲートしてください。

第 3 項　可視化のオプション

フレーム解析モードでは、以下のように、可視化のオプションが複数用意されています。

- **D　デプスの複雑性レベル**：このオプションは、赤みを帯びた色合いのブレンディングを追加します。スクリーンが明るいほど、より多くのフレームバッファが作動したことを意味します。各フレームバッファの Read-Modify-Write (RMW) は、最大 32 RMW まで、8 つごとの明度で増加します。スクリーンカラーが飽和(255)すると、お手持ちのアプリケーションはフレームバッファに対してオーバーライティングを行

う頻度が極めて高くなるため、フィルが律速となる性能上のボトルネックが生じることがあります。

- **W ワイヤフレーム**：このオプションはワイアフレームレンダリングを強制するので、シーンのジオメトリの複雑性レベルを調べることができます。

第４項　高度状態検査機能高度な状態検査機能

Advanced… (A) ボタンは、高度な状態検査機能を起動させます。スクリーンの下にあるスライダーを利用してナビゲーションすることもできますが、スクリーンの上部には、グラフィックスパイプライン中の各段階に対するボタンが幾つか表示されています。監視機能を切り替えるには、このボタンを押すか、ショートカットキーを押します。

- **1** インデックスユニットーバーテックスデータをフェッチします
- **2** バーテックスシェーダーーバーテックスシェーダーを実行します
- **3** ピクセルシェーダーーピクセルシェーダーを実行します
- **4** ラスター演算ーフレームバッファ中でのポストシェーディング演算

> **Note:** NVPerfHUD 状態検査機能のそれぞれに表示される状態情報に関する詳細につきましては、Direct SDK の最新バージョンにインストールされている説明書をご覧ください。

各段階をクリックすると、カレントドローコールの間に当該段階で起こっていることについて詳細な情報を見ることができます。以下の節では、各情報監視機能によって表示されている情報について説明します。

第２節　インデックスユニット状態検査機能

この状態検査機能を選択すると、カレントドローコール中のインデックスユニットに関する情報が表示されます。

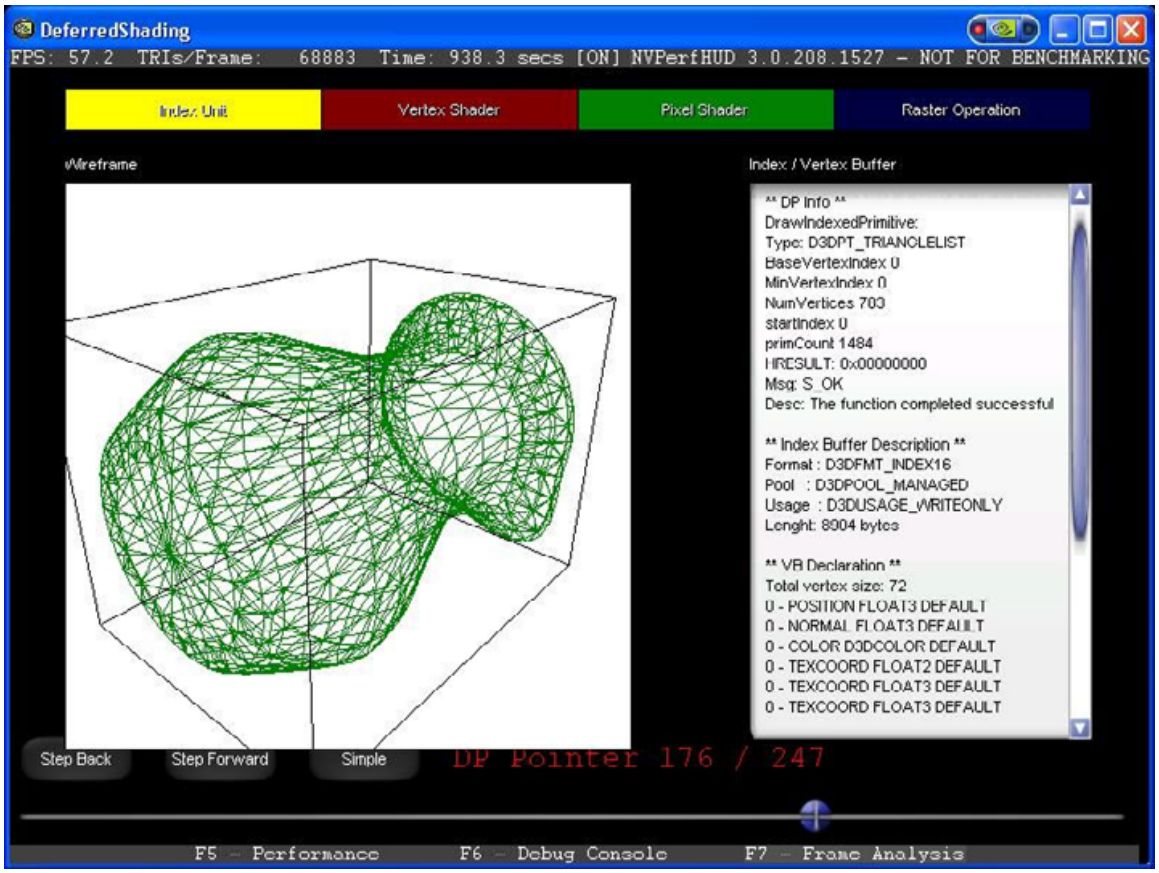


図１２　　インデックスユニット状態検査機能

スクリーンのセンターには、このドローコールに付随するジオメトリの回転式ワイアフレームレンダリングが、境界ボックス内部に表示されています。

Note: このバージョンでは、点および線はサポートされていません。

この次に、このドローコールに対してバーテックスデータをフェッチするために使用された情報がすべてリストボックスによって表示されます。

- ドローコールパラメータとリターンフラッグ
- インデックスおよびバーテックスバッファのフォーマット、サイズなど
- FVF

インデックスユニット状態検査機能を使用する場合、お手持ちのアプリケーションが送信するバッチが正確であることを確認するために、ワイアフレームレンダリングをご覧いただく必要があります。例えば、マトリックスパレットスキニングを行っており、レンダリングが破損したと

きには、バーテックスバッファ／インデックスバッファ中の基準状態が適切であることを確認してください。基準状態が適切である場合、シーン中のこのジオメトリのレンダリングの破損は、バーテックスシェーダーまたは不適切なバーテックスウエイトが原因と思われます。

インデックスのフォーマットが適切であることも確認して、可能な場合には常に、16 ビットのインデックスが使用されていることを確かめてください。

第３節　　バーテックスシェーダー状態検査機能

この状態検査機能を選択すると、カレントドローコール中のバーテックスシェーダーに関する情報が表示されます。

図１３　　バーテックスユニット状態検査機能

バーテックスシェーダープログラムが、プログラムで使用されているすべての定数およびテクスチャとともに、検査のために表示されます。バ

ーテックスシェーダーがアドレスレジスタ（例えば、マトリックスパレットスキニング）を使用している場合、すべての定数が表示されます。各テクスチャサンプラーについての情報も参考のために表示されます。+/ーキーを使用すると、表示されているテクスチャを拡大できます。

バーテックスシェーダー状態検査機能を使用する場合、以下のことを行ってください。

- 予定のバーテックスシェーダーがカレントドローコールに対して適用されていることを確認する
- 定数が**#NAN** または**#INF** をパスしていないことを確認する

第４節　　ピクセルシェーダー状態検査機能

この状態検査機能を選択すると、カレントドローコール中のピクセルシェーダーに関する情報が表示されます。

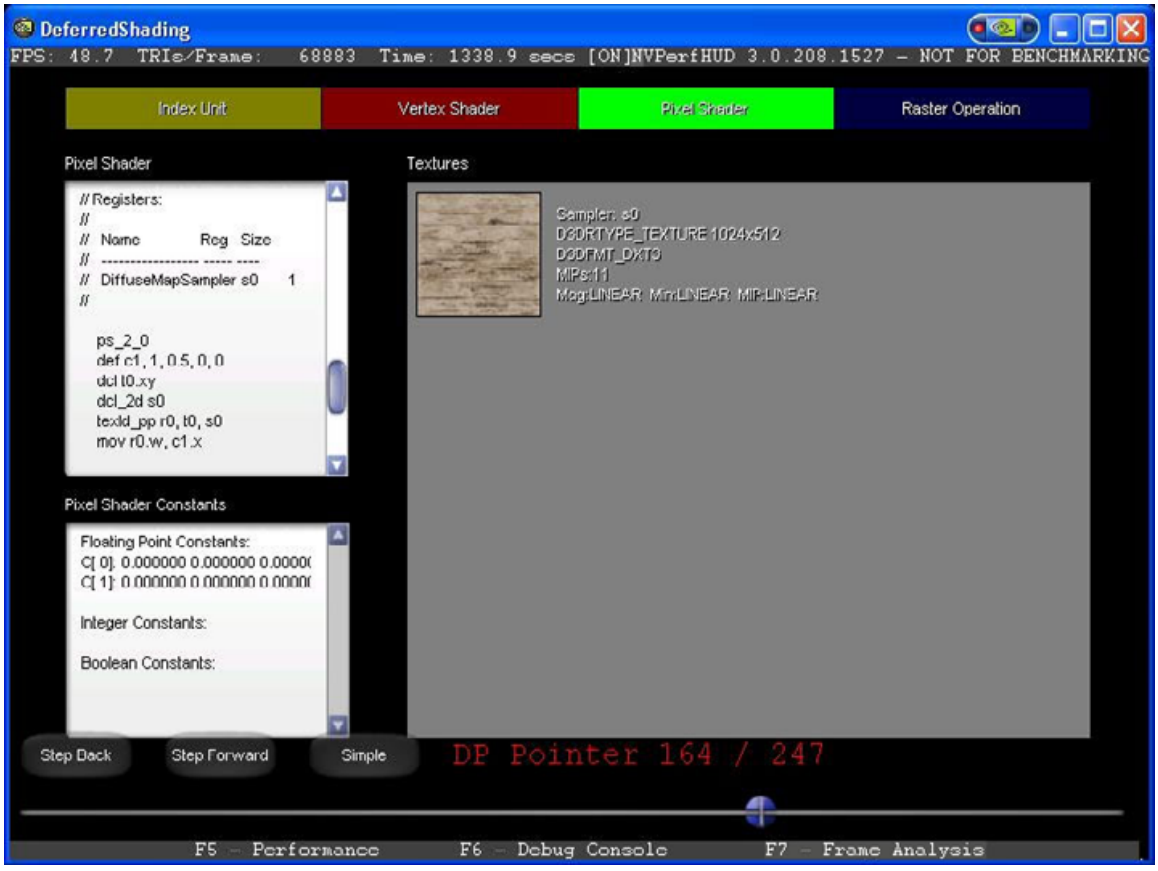


図 3　　ピクセルシェーダー状態検査機能

ピクセルシェーダープログラムが、プログラムで使用されているすべての定数およびテクスチャとともに、検査のために表示されます。各テクスチャサンプラーについての情報も参考のために表示されます。+/－キーを使用すると、表示されているテクスチャを拡大できます。

ピクセルシェーダー状態検査機能を使用する場合、以下のことを行ってください。

- 予定のピクセルシェーダーがカレントドローコールに対して適用されていることを確認する
- 定数が**#NAN** または**#INF** をパスしていないことを確認する
- テクスチャおよび RTT が適切に使用されていることを確認する

第 5 節　　ラスター演算状態検査機能

この状態検査機能を選択すると、カレントドローコール中のラスター演算(ROP)に関する情報が表示されます。

Note: ラスター演算状態検査機能に表示される状態情報に関する詳細につきましては、Direct SDK の最新バージョンにインストールされている説明書をご覧ください。

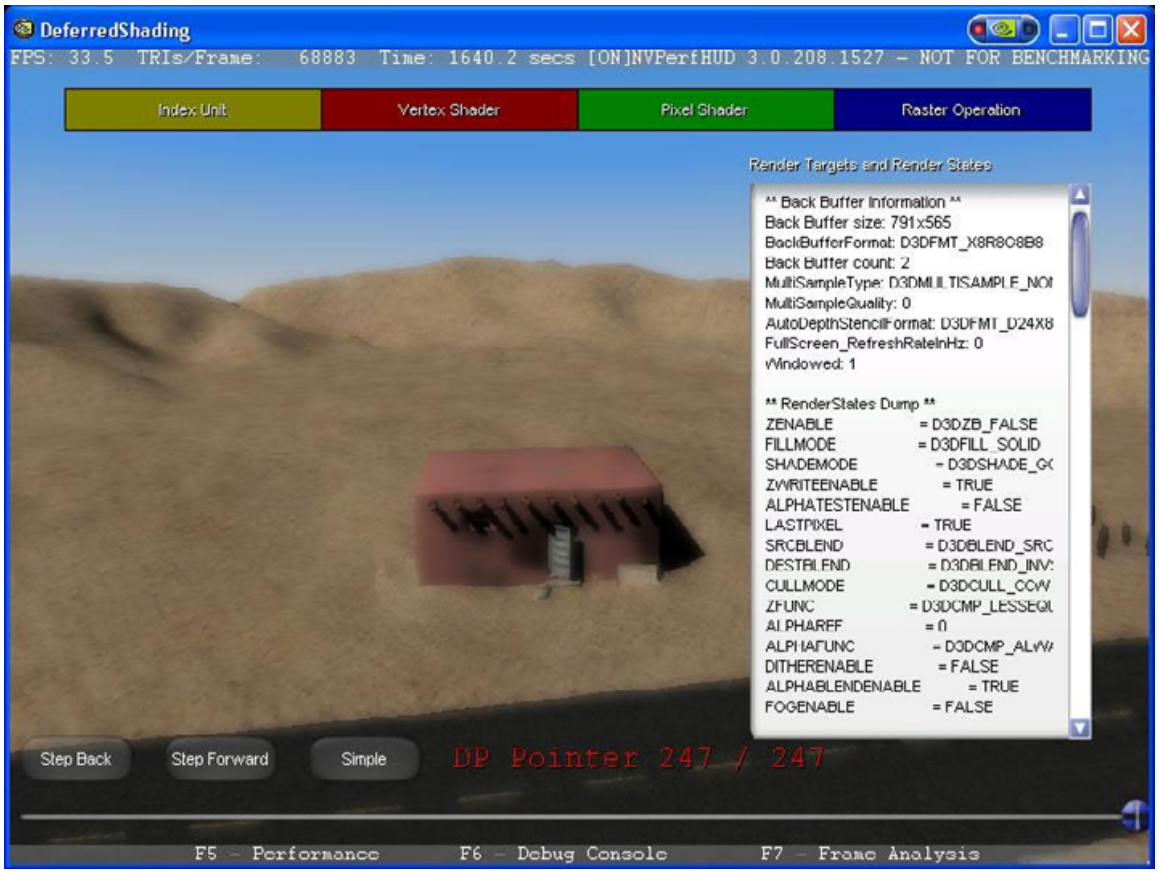


図 4　　ラスター演算状態検査機能

このドローコールに対するポストシェーディングラスター演算についての情報が検査のために表示されます。この情報には、以下のものが含まれます。

- レンダーターゲットのフォーマット
- バックバッファのフォーマット
- フレームバッファのプロセシングを高価なものにすることがある、以下のようなレンダー状態
  - **Zenable**－**Z** 比較演算
  - **Fillmode**－ラステライゼーションモード
  - **ZWriteEnable** – **Z** をデプスバッファに書き込むか否か
  - **AlphaTestEnable** –アルファテストが起動されている
  - **SRCBLEND** および **DSTBLEND** –どのブレンド演算か
  - **AlphablendEnable** – フレームバッファ中でのアプリケーションブレンディングである
  - **Fogenable**－フォグが起動されている

- **Stencil enable**ー起動されたステンシルバッファに書き込みを行っている
- **StencilTes**t....

ラスター演算状態検査機能を使用する場合、以下のことを行ってください。

- ブレンディングが上手く行われないときには、バックバッファフォーマットにアルファコンポーネントが含まれているか確認する
- 不透明なオブジェクトの描画が **blendEnable** によって行われていないことを確認する

第６章
性能上のボトルネックを解析する

第１節　グラフィックスパイプラインの性能

ここ数年の間、ハードウェアでアクセラレートされたレンダリングパイプラインの複雑さは著しく増大しており、このため、パフォーマンス特性はますます複雑で分かりにくいものとなっています。性能を向上させるために、かつては、レンダラー内での内部ループの CPU サイクルを減らすという比較的単純な作業を行えばよかったのですが、今では、ボトルネックを決定し、それらを体系的に最適化することを繰り返し実行しなければなりません。特定（Identification）と最適化（Optimization）という反復プロセスは、異種のプロセッサから構成されるマルチプロセッサシステムをチューニングするための基本的な作業であり、パイプラインは、原理的に、最も遅い工程の速度を超えることができないという考え方に基づいています。シングルプロセッサシステムの最適化が未完成で不徹底だと、性能のゲインが最初になるだけですが、マルチプロセッサシステムでは、性能のゲインがゼロになるという事実が、論理的に帰結されます。

グラフィックスの最適化にやっきになって取り組み、ゼロ性能が改良されるのを確認する作業はつまらないものです。本章の目的は、NVPerfHUD を使用して、性能上のボトルネックを特定し、時間の浪費を避ける方法について説明することです。

第 1 項　パイプラインの概要

最上級のレベルでは、パイプラインは、CPU および GPU という 2 つのパートに分けられます。図 16 には、GPU 中で並行的に演算を行っているたくさんの機能的ユニット（これらは、別個の特殊な用途のプロセッサと見ることができます）と、ボトルネックが発生する可能性がある多くのスポットが示されています。これらには、バーテックスおよびインデックスフェッチング、バーデックスシェーディング（transform and lighting）、ピクセルシェーディング、テクスチャローディング、ラスター演算（ROP）などがあります。

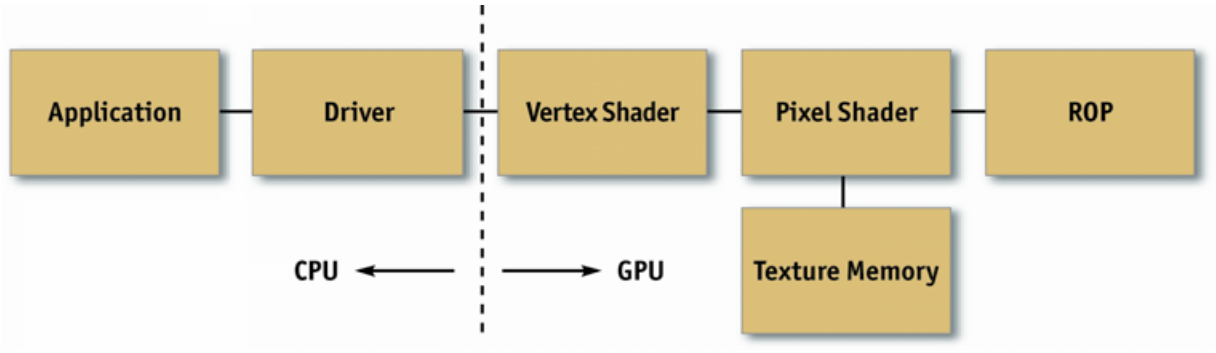


図 16　　パイプラインの概要

第 2 項　方法

ボトルネックを適切に特定せずに最適化を行うことは、開発時に無駄な努力しなければならなくなる原因です。そこで、NVIDIA は、このプロセスを以下の基本的な特定と最適化のループに公式化しました。

- **特定する**
  パイプライン中の各ステージに対して、NVPerfHUD 実験を使用して、そのステージを単離します。性能が変われば、そのステージがボトルネックであることが分かります。アプリケーションの作業量が変わるようにアプリケーションを変更することにより、同様の実験に独自に取り組むこともできます。
- **最適化する**
  ボトルネックとなっているステージが特定されたら、性能が向上しなくなるまで、あるいは、所望の性能レベルが達成されるまで、作業量を減らします。
- **繰り返す**
  所望の性能レベルに到達するまで、ステップ 1 と 2 を繰り返します。

第２節　　ボトルネックを特定する

ボトルネックを特定することで、実際に最適化に尽力する際に的確な判断が下せるので、ボトルネックの特定は最適化作業で重要な位置を占めます。図 17 は、お手持ちのアプリケーションで正確なボトルネックを特定するために必要とされる一連の工程がフローチャート形式で図示されています。パイプラインの最終工程から開始されるため、フレームバッファ演算（ラスター演算とも呼ばれます）から始まって、CPU で終了することに注意してください。単一の基本要素（通常はトライアングル）は、当然のことながら、ボトルネックが変化する可能性が最も高いフレームを通じて、単一のボトルネックを有しますが、このため、パイプライン中の複数のステージにおける作業量を変化させると、性能に影響が及ぶことが多いことにも留意しなければなりません。例えば、ポリゴンが少ないスカイボックスの場合、ピクセルシェーディングやフレームバッファアクセスによって制約されることが多いのに対して、スクリーン上の僅かなピクセルにマッピングしているだけの skinned mesh の場合、CPU またはバーテックスプロセッシングによって制約を受けることが多くなります。このため、オブジェクトごとに、あるいはマテリアルごとに作業量を変化させるのが有益であることが少なくありません。

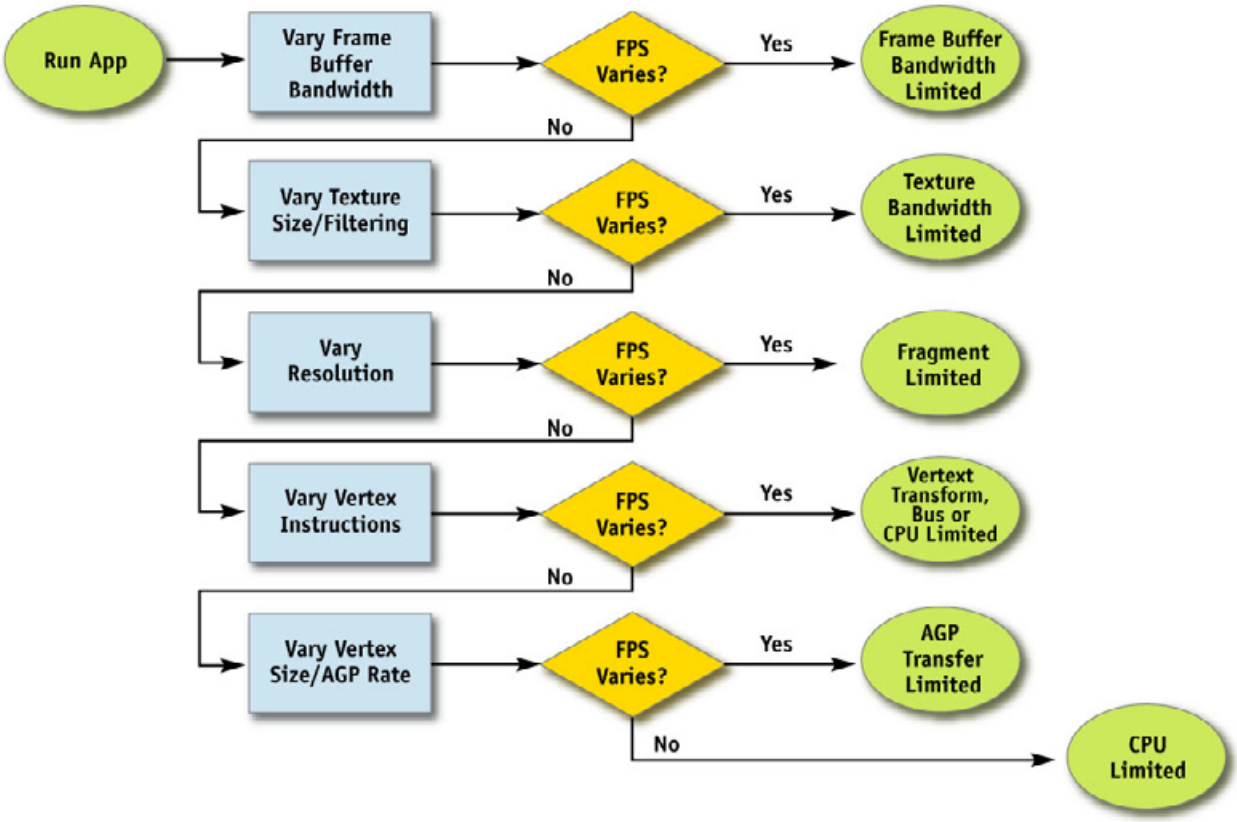


図 17　　ボトルネックを特定する

Note: CPU が律速だと思われる場合には、常に **N** を押してください。お手持ちのアプリケーションのフレーム速度が変わらなければ、CPU が律速です。

第 1 項　ラスター演算のボトルネック

ROP と呼ばれることが多いパイプラインの最終段階は、デプスとステンシルの読み書き、デプスとステンシルの比較、色の読み書き、アルファブレンディングとアルファテストの実行に必要です。ご存知のように、ROP の作業量の多くは、使用可能なフレームバッファの帯域幅に大きな負担をかけます。

お手持ちのアプリケーションがフレームバッファの帯域幅によって制約されているかどうかを調べるための最も優れた方法は、カラーおよび／またはデプスバッファのビットデプスを変更することです。ビットデプスを 32 ビットから 16 ビットに減少させることで、性能が著しく向上すれば、

フレームバッファの帯域幅が制約要因となっていることが明らかです。

第２項　テクスチャ帯域幅のボトルネック

テクスチャフェッチがメモリに伝達されるたびに、テクスチャ帯域幅が消費されます。現在の GPU には、無用なメモリリクエストを最小限に抑えるように設計されたテクスチャキャッシュが備わっていますが、なおメモリリクエストは発生しているため、メモリ帯域幅のかなりの量が消費されます。

NVPerfHUD が作動しているときに **T** を押すと、お手持ちのアプリケーション中のテクスチャがすべて、2×2 テクスチャに置き換わります。これにより、テクスチャキャッシュのコヒーレンスが向上して、テクスチャーフェッチがずっと素早く実行されます。これによって、性能が著しく向上すれば、テクスチャ帯域幅が制約要因となっています。

テクスチャ帯域幅は、GPU メモリのクロックスピードの関数でもあります。

第３項　ピクセルシェーディングのボトルネック

ピクセルシェーディングは、関連するカラーおよびデプス値によりピクセルを生成させる際に払われる負担です。これが、「ピクセルシェーダー」を実行させる際の負担となります。ピクセルシェーディングとフレームバッファ帯域幅は何れもスクリーン解像度の関数であるため、これらは「フィルレート」という呼び名で総称されることが多いですが、これらはパイプライン中では別個の 2 つのステージであり、両者の差を区別できることは、ボトルネックを効果的に特定し、最適化するのに不可欠であることに留意してください。

ピクセルシェーディングがボトルネックであるかどうかを決定する最初のステップは、NVPerfHUD を用いて、すべてのシェーダーを極めて単純なシェーダーに置き換えることです。これを行うためには、性能解析モードで 1, 2, 3, … など用いて、一度に 1 つのピクセルシェーダーのプロファイルを無効にして、フィルレートの変化を観察します。これによって性能が向上すれば、ピクセルシェーディングが犯人である可能性が極めて高いといえます。

次のステップは、NVShaderPerf または FX Composer 中の Shader Perf パネルを用いて、どのシェーダーが高価なシェーダーであるかを突き止めます。ピクセルシェーダーのコストはピクセル単位であるため、少数のピクセルにしか影響を与えない高価なシェーダーは、多くのピクセルに影響を与える高価なシェーダーほどには、大きな問題とならない可能性があることに注意する必要があります。基本的には、shader シェーダー_cost コスト＝cost コスト_per パー_pixel ピクセル* number_of_pixels_affected です。コストが最も大きなシェーダーに、性能最適化作業を集中させてください。

FX Composer には、お手持ちのシェーダーの性能コストを低減するのに役立つと思われるシェーダー最適化チュートリアルが数多く用意されています。

Note: X ComposerおよびNVShaderPerfの最新バージョンは、あらゆるNVIDIA GPUファミリーのシェーダー性能を解析できます。何れも、http://developer.nvidia.com からご入手いただけます。

第４項　バーテックスプロセッシングのボトルネック

レンダリングパイプラインのバーテックスを変換する段階は、一群のバーテックス属性（例えば、モデルー空間位置、バーテックス法線、テクスチャ座標など）を取得し、クリッピングとラステライゼーションに適した一群の属性（例えば、均質なクリップー空間位置、バーテックスライティングの結果、テクスチャ座標など）を生成するために必要です。当然のことながら、この段階の性能は、プロセッシングを受けるバーテックスの数とともに、バーテックス当たりで行われる作業の関数です。

バーテックスプロセッシングがボトルネックであるかどうかを決定するには、NVPerfHUD をお手持ちのアプリケーションで実行して、**v** キーを押し、バーテックスユニットを単離するだけです。得られたフレーム速度が元のフレーム速度と概ね同じであれば、バーテックス／インデックスバッファの AGP 転送、バーテックスシェーダーユニット、または非効率なロックおよびそれに伴う GPU の失速が、お手持ちのアプリケーションの制約原因です。

Note: 非効率なロックを除外するために、Direct3D デバッグランタイムでアプリケーションを作動させ、エラーや警告が表示されないことを確かめてください。

第５項　バーテックスとインデックス転送のボトルネック

バーテックスとインデックスは、パイプラインのGPU部分で最初の工程としてGPUによってフェッチされます。バーテックスとインデックスフェッチングのパフォーマンスは、実際にバーテックスとインデックスが配置されている場所（何れかのシステムメモリであることが通常）に応じて変わります。これは、バーテックスとインデックスが、AGPもしくはPCI-Expressのようなバスを通じて、またはローカルフレームバッファメモリを通じてGPUに転送されることを意味しています。現代のAPIでは、ドライバーが正しいタイプのメモリを選択できるようにするための使用上のヒントをアプリケーションが与えられるようになっていますが、特にPCプラットフォーム上では、この決定は、お手持ちのアプリケーションでぇなく、デバイスドライバーに一任されていることが多いです。お手持ちのアプリケーションでインデックスバッファとバーテックスバッ

ファを最適な状態で使用する方法については、NVIDIA GPU Programming Guide[リンク] をご参照ください。

お手持ちのアプリケーションでバーテックスまたはインデックスフェッチングがボトルネックとなっているかどうかを決定するには、バーテックスフォーマットサイズを変更します。

データがシステムメモリに配置されている場合には、バーテックスおよびインデックスフェッチングの性能は AGP/PCI-Express 速度の関数であり、ローカルフレームバッファメモリ中に配置されている場合には、メモリクロックの関数です。

第 6 項　CPU のボトルネック

お手持ちのアプリケーションの性能が CPU によって制約されているかどうかを知るには 2 通りの方法があります。

お手持ちのアプリケーション性能が CPU によって制約されていることを見分ける 1 つの簡便な方法は、NVPerfHUD タイミンググラフ中の GPU アイドル（緑）の線を見ることです。グラフ底部で緑の線が平坦であれば、GPU はアイドル状態にはありません。緑の線がグラフの底部から飛び上がっていれば、CPU が GPU に対して十分な作業を与えていないことを意味しています。

N を押して GPU を単離することによっても、お手持ちのアプリケーションが CPU によって制約を受けているかどうかを見分けることができます。これを実行すると、NVPerfHUD は、Direct3D ランタイムにすべての DP コールを強制的に無視させます。得られたフレームレートは、無限速度の GPU とディスプレイドライバーを用いたときにお手持ちのアプリケーションで得られるフレームレートを近似しています。

お手持ちのアプリケーションの性能が CPU によって制約されている場合（CPU の処理量が多すぎて GPU に情報を伝達できない場合）、以下の原因が考えられます。

- DP コールが多すぎる－各コールについてドライバーのオーバーヘッドが存在します。図 18 と添付の説明をご覧ください。
- アプリケーションロジック、フィジックスなどを要求している－FRAME_TIME (YELLOW)線と TIME_IN_DRIVER (RED) 線との乖離は、CPU がお手持ちのアプリケーションに占用されていた時間量を表します。
- リソースをロードまたはアロケートしている－例えば、ドライバーは各テクスチャを処理しなければならないので、ドライバーが多数の（大きな）テクスチャをロードしているためにビジーである間に、GPU がすべての係属中の作業を終えることがあります。3.1.2 節で説明したリソース生成モニターでこれらのイベントを監視してください。

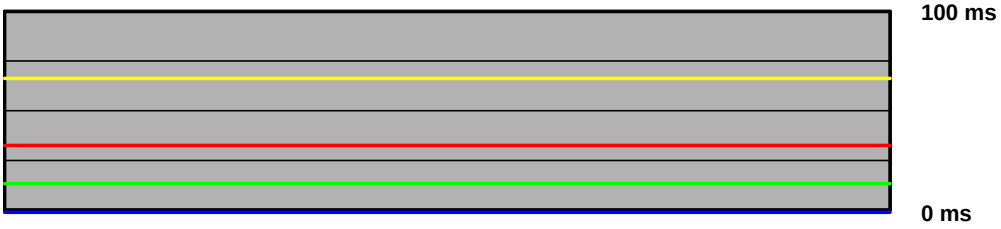


図 18　　ドライバーへのコールが多すぎる

図 18 には、お手持ちのアプリケーションが、ドライバーへのコールを過度に行っている典型的な事例が示されています。バッチ数を表すこのグラフも調べることにより、このシナリオを二重にチェックしてください。

第３節　　最適化

これで、ボトルネックが特定されたので、アプリケーション性能を向上させるために、特定の段階の最適化を行わなければなりません。以下では、アプリケーションの総合的な性能を向上させ得る最適化についてのヒントが、段階ごとにまとめられています。

第１項　CPU の最適化

アプリケーション性能は、複雑なフィジックスまたは AI のために、CPU によって制約されることがあります。バッチサイズとリソース管理が不十分であるために、性能が低下することもあります。お手持ちのアプリケーションが CPU によって制約されていることが分かった場合には、以下のヒントを試して、レンダリングパイプライン中での CPU の作業を減らしてみてください。

第２項　リソースロッキングを減らす

リソースは、テクスチャまたはバーテックスバッファの何れかであると思われます。GPU リソースへのアクセスを要求する同期動作行う場合には、GPU パイプラインが大幅に失速している可能性が常に潜んでおり、これにより、CPU と GPU サイクルの両方に負担がかかります。GPU パイプラインが要求されたリソースを排出して戻すのを待つために、CPU はループ中を回転しながら待機しなければならないので、CPU サイクルが無駄に使用されます。次いで、パイプラインがアイドル状態となり、再度補給されなければならないために、GPU サイクルが浪費されます。

れは、以下のことを実行しているときに常に生じる可能性があります。

- 既にレンダリングした表面をロックし、またはこのような表面から読み込みを行っている。
- テクスチャやバーテックスバッファなど、GPU が読み込んでいる表面に書き込みを行っている。

ビジー状態のリソースをロッキングすることは、DRIVER_WAITS_FOR_GPU (青)の線を上昇させるのに役立ちます。ドライバーが GPU を待機する原因についてさらに詳しい情報を知りたい方は、付属書類 A をご覧ください。

非効率なロックを除外するために、Direct3Dデバッグランタイムでアプリケーションを作動させ、エラーや警告が表示されないことを確かめてください。リソースを効率的にロックする方法について詳しく知りたい方は、以下のホワイトペーパーをお読みください。
http://developer.nvidia.com/object/dynamic_vb_ib.html

第 3 項　ドローコールの数を最小限にする

ジオメトリを描画するための API 関数の呼び出しには、すべて CPU の負担を伴うので、API コールの数を最小限に抑えることで、特に、グラフィックス状態変化の数を最小限に抑えることで、一定数のレンダリングされたトライアングルに対して使用される CPU の作業は最小限に抑えられます。

ここでは、*バッチ*を、DirectX9 における `DrawPrimitive()`および `DrawIndexedPrimitive()` などの単一の API レンダリングコールでレンダリングされる基本作業のグループと定義します。バッチの「サイズ」とは、その中に含まれる基本作業の数を意味します。

NVPerfHUD を使用すれば、バッチがどの程度良好に行われているかを知ることができます。**B** を押すと、フレーム当たりのドローコール当たりのトライアングル数の分布が表示されます。図 19 には、基本作業の数が少ない DP コールが多数存在するために、性能が低下していると思われるアプリケーションが示されています。

図 19　　多くの小さな DP コール

DP コールの数を減らすためには、以下の作業を試してください。

- **トライアングルストリップを使用している場合には、縮重トライアングルを使用して、バラバラのストリップを一つにまとめます。**これにより、複数のマトリックスがマテリアルを共有していれば、単一のドローコールで複数のストリップを送信することができます。`NVTristrip` libraryは、http://developer.nvidia.comから入手することが可能で、これに対するソースコードを提供します。
- **テクスチャページを使用する。** 異なるオブジェクトが異なるテクスチャを使用していると、バッチは分解されることが多くなります。多くのテクスチャを単一の 2D テクスチャに配置し、テクスチャ座標を適切に設定することで、複数のテクスチャを使用しているジオメトリを単一のドローコール中に送ることができます。この技法には、ミップマッピングやアンチアイリアシングに伴う問題が生じ得ることに注意しなければなりません。これらの多くの問題を回避する 1 つの技術は、各 2D テクスチャをキューブマップの各フェースに詰め込むことです。NVIDIA SDK の最新バージョンには、テクスチャページを作成して、プレビューするのに役立つ様々なテクスチャアトラスツールが用意されています。
- **バーテックスシェーダーコンスタントメモリをマトリックスの照合表として使用する。**多くの小さなオブジェクトがすべてのマテリアル特性を共有しているが、マトリックス状態のみが異なる場合（例えば、同じような木からなる森）に、バッチの分解が起こりやすくなります。これらのケースでは、複数のマトリックスをバーテックスシェーダーコンスタントメモリーにロードして、それぞれのオブジェクトについて、インデックスをバーテックスフォーマットでコンスタントメモリ中に格納することができます。次いで、バーテックスシェーダー中のコンスタントメモリを検索し、正確な変換マトリックスを使用するた

めに、このインデックスを使用することで、N オブジェクトを一度にレンダリングします。

- **シーンの中で同じメッシュが複数コピー存在する場合には、ジオメトリインスタンシングを使用します。**この技術によって、単一のドローコールと 2 つのバーテックスストリームで、同一メッシュのオブジェクトの複数コピーを描くことができます。各コピー、すなわちメッシュの「インスタンス」は、異なる場所に描くことができ、（必要に応じて）異なる可視化によって描くことができます。一方のストリームには、インスタンスを行うべきメッシュが単一コピー含まれており、他方のストリームには、per-instance データ（ワールド変換、カラーなど）が含まれています。次いで、単一のドローコールを発して、描きたいインスタンスの数を指示することができます。一般に、多くのドローコールの CPU オーバーヘッドがこれにより軽減されるので、多くの低（100 以下）ポリオブジェクトが存在する場合に、ジオメトリインスタンシングは最も有用です。NVIDIA SDK の最新バージョンには、ジオメトリインスタンシングの例（完全なソースコード付き）も用意されています。

- **シェーダー分岐を使用してバッチサイズを増加させる。**現在の GPU には、シェーダー内部での分岐を可能にする可変バーテックスおよびピクセルプロセッシングパイプラインが備わっています。例えば、一方のバッチには 4 ボーンスキニングのバーテックスシェーダーが必要で、他方には 2 ボーンスキニングのバーテックスシェーダーが必要であるために、2 つのバッチが分かれている場合には、代わりに、必要とされるボーンの数だけループして、ブレンディングの重みを蓄積させ、この重みが合算されて 1 つになった時点でループから抜け出すバーテックスシェーダーを書き込むことが可能です。このようにして、2 つのバッチを 1 つにまとめることができます。シェーダー分岐がサポートされていないアーキテクチャの場合、あらゆるものに対して 4 ボーンのバーテックスシェーダーを使用し、4 ボーンの作用より少ないバーテックス上のボーンウェイトを消去することで、シェーダーサイクルを犠牲にして、同様の機能を実行することができます。
- **できるだけパイプラインの下流まで決定を保留する。**光沢を与えるようにピクセルシェーダー定数を設定するためにバッチを分割するよりも、テクスチャのアルファチャネルを光沢因子として使用する方が速いといえます。同様に、テクスチャとバーテックスの中にシェーディングデータを入れることで、さらに大きなバッチの提供が可能となります。

第 4 項　バーテックス転送の負担を低減させる

現在のアプリケーションではバーテックス転送がボトルネックとなることは希ですが、バーテックス転送が問題となることも起こりえないことではありません。バーテックスの転送や、さらに可能性は低いですが、インデックスの転送がお手持ちのアプリケーションでボトルネックとなっている場合には、以下のことをお試しください。

- **バーテックスフォーマットで使用されるバイト数をできるだけ少なくする。**バイトが十分であれば、あらゆるものについて（例えば、色）浮動小数の使用を控えてください。
- 入力バーテックスフォーマットの内部にバーテックス属性を格納するのではなく、バーテックスプログラムの内部の引き出し可能なバーテックス属性を生成させる。例えば、接線、従法線および法線のうち 2 つが与えられれば、バーテックスプログラム中で単純な外積を用いて 3 番目の要素は導けるので、接線、従法線および法線をすべて格納する必要はないことがしばしば見受けられます。この技術により、バーテックスプロセッシング速度を犠牲にして、バーテックス転送速度を上げることができます。
- **32 ビットのインデックスではなく、16 ビットのインデックスを使用する。**16 ビットのインデックスの方が、フェッチや回転移動の負担が少なく、メモリの使用が少なくて済みます。

- **比較的順を追う様式でバーテックスデータにアクセスする。**現在のGPUキャッシュメモリはバーテックスをフェッチするときにアクセスを行います。あらゆるメモリ階層において、レファレンスの空間的場所は、キャッシュ中のヒットを最大化させることにより、必要な帯域幅を減らすのに役立ちます。

第５項　バーテックスプロセッシングを最適化する

現在のGPUでは、バーテックスプロセッシングがボトルネックとなることは希ですが、使用パターンや標的ハードウェアによっては、これが問題となる可能性はあります。お手持ちのアプリケーションで、バーテックスプロセッシングがボトルネックになっていることが明らかとなった場合には、これらのヒントをお試しください。

- **オブジェクトごとの計算をCPU上に取り出す。**オブジェクトまたはフレームごとに一度変化する計算が、バーテックスシェーダー中で行われることがしばしばあります。例えば、方向光ベクトルを視点空間に変換することが、バーテックスシェーダー中で行われることがありますが、計算の結果はフレームごとにしか変化しません。
- **ポストTnLバーテックスキャッシュを最適化する。**現在のGPUには、最も新しく変換されたバーテックスの結果を格納する小さなFIFOキャッシュが備わっています。このキャッシュの中にあるヒットは、パイプライン中で前に行われたすべての作業とともに、T&L作業を保存します。このキャッシュを活用するためには、インデックスが付された基本作業を使用し、メッシュ全体にわたってレファレンスの場所を最大化するようにバーテックスに命令を与えなければなりません。D3DXやNVTriStripなど、この作業に役立つツールが無料でご利用いただけますー http://developer.nvidia.com/object/nvtristrip_library.html。
- **プロセッシングされるバーテックスの数を減らす。**これは実に根本的な問題ですが、静的LOD群など、シンプルなlevel-of-detailスキームを用いれば、バーテックスプロセッシングの作業負荷が確実に減少しあmす。
- **バーテックスプロセッシングLODを使用する。**プロセッシングされたバーテックスの数に対してLODを使用するのと平行して、実際のバーテックス計算事態にも、LODを試してください。例えば、遠くのキャラクターに対してはフル4ボーンスキニングを行う必要がない可能性が高く、照明に関して負担の少ない近似でやり過ごせることができると思われます。お手持ちの機材がマルチパスである場合、遠方にあるLODを低くするためにパスの数を減らすことで、バーテックスプロセッシングの負担が減少するでしょう。
- **正しい座標空間を使用する。**座標空間の選択が、バーテックスプログラム中で値を計算するのに必要とされる指示の数に影響を与える場合が多々あります。例えば、バーテックスライティングを行う場合、バ

ーテックス法線がオブジェクト空間中に格納されていれば、光ベクターが視点空間中に格納されるので、バーテックスシェーダー中の 2 つのベクターのうち一方を変換しなければなりません。光ベクターが、CPU 上で、オブジェクトごとに逐一オブジェクト空間に変換されれば、バーテックスごとの変換が不必要となるため、GPU のバーテックス命令を省略できます。

- **計算を「アーリーアウト」させるためにバーテックス分岐を使用する。**バーテックスシェーダーの中で多数の光をループさせ、通常の低ダイナミックレンジの[0..1]照明を行うのであれば、一つへの集中をチェックすることができ、あるいは、照明を行わないのであれば、さらなる計算を止めることができます。スキニングを用いて同様の最適化を行うことも可能であり、重みの合計が 1 になった時点で（従って、その後の重みはすべてゼロになります）停止させることができます。これは、GPU がバーテックス分岐を実施する方法に依存しているため、すべてのアーキテクチャの性能の向上を保証するものではないことにご注意ください。

第 6 項　ピクセルシェーディングのスピードをアップさせる

長くて複雑なピクセルシェーダーを使用している場合、ピクセルシェーディングが制約となっていることが多くあります。これに該当する場合には、以下のヒントをお試しください。

- **まずデプスをレンダリングする。**一次シェーディングパスをレンダリングする前にデプス専用（カラーなし）パスをレンダリングすると、実行しなければならないピクセルシェーディングとフレームバッファメモリへのアクセスの量が減少するため、とりわけデプスの複雑性レベルが高いシーンでは、性能を劇的にアップさせることができます。デプス専用パスを最大限に活用するには、フレームバッファへのカラーの書き込みを無効にするだけでは十分でなく、ピクセル上のすべてのシェーディング（例えばアルファテストなど、カラーだけでなくデプスに影響を与えるシェーディングを含む）無効にしなければなりません。
- **早期 Z の最適化がピクセルプロセッシングを放棄できるようにする。**現在の GPU には、見ることのできないピクセルをシェーディングしないようにするためのシリコンが備わっていますが、これらの GPU は、カレントポイントまでのシーンの把握を利用しているため、概ね前から後ろへの順番でレンダリングを行うことが大いに役立ちます。まず別個のパスでデプスを規定しておくと（上記参照）、シェーディングされたデプス複雑性レベルを 1 まで効果的に減少させることによって、その後のパス（高価なシェーディングがすべて完了している）を劇的にスピードアップさせることができます。

- **複雑な関数をテクスチャ中に保存する。**テクスチャは、照合表として極めて有用なことがあり、その結果を無償でフィルタリングできるという利点もあります。この場合の標準的な例は正規化キューブマップであり、単一のテクスチャの照合を行うだけで、任意のベクターを高精度で正規化することができます。
- **ピクセル単位の作業をバーテックスシェーダーに移行させる。**バーテックスシェーダー中で行われているオブジェクト単位の作業をCPU に移行させる必要があるのと同様に、バーテックス単位の計算は（スクリーン空間に正しく線形補間可能な計算とともに）バーテックスシェーダーに移行させる必要があります。一般的な例としては、座標系の間の演算ベクターと変換ベクターが挙げられます。
- **必要最小限の精度を使用する。**DirextX 9 のような API を使用することで、減少した精度で機能できる量または計算に対するピクセルシェーダーコードにおいて、精度についての手がかりを特定することが可能です。多くの GPU は、内部の制度を減少させて性能を向上するために、これらの手がかりを活用することができます。
- **不必要な正規化を避ける。**正規化を過度に用いようとして、計算を行う際に、途中の全工程であらゆるベクターを正規化することが共通の誤りです。どの変換が長さを保存し（正規直交基底による変換など）、どの計算がベクターの長さに依存していないか（キューブマップの照合など）を把握してください。
- **可能であれば半精度の正規化を使用する。**半精度での正規化は、実質的に、NV4x クラスの GPU 上での自由演算といえます。HLSL で、正規化すべきベクターに対して「half」タイプを使用してください。DirectX 9 で、ps_2_0 以降のバージョンのアセンブリシェーダーを使用している場合には、**nrm_pp** を使用します（等価な計算を行うために、すべての演算に対して '_pp' 修飾子を使用します）。お手持ちの HLSL シェーダーをテストするときには、正規化に対応する演算に対して _pp 修飾子が使用されていることを確かめ、確実に'half'データタイプを適切に使用するために、生成されたアセンブリをチェックするのはよい考えです。fxc.exe または FX Composer Shader Perf パネルを実行して、HLSL シェーダーから生成されたアセンブリを確認することができます。NVShaderPerf のコマンドラインユーティリティーも使用できます。
- **ピクセルシェーダーの level-of-detail (LOD)の使用を検討する。**バーテックス LOD ほどの効果はありませんが（遠方にあるオブジェクトは、遠近感を得るためにピクセルプロセッシングについて当然それ自身に LOD を行っているからです）、表面全体にわたってパスの数を減らしながら、遠方にあるシェーダーの複雑性レベルを低下させると、ピクセルプロセッシングの作業量を軽減することができます。

- **テクスチャ帯域幅が制約要因となっていないことを確認する。**さらに詳しい情報は以下の「**Error! Reference source not found.**」の項をご覧ください。

第７項　テクスチャ帯域幅を減らす

お手持ちのアプリケーションがメモリ帯域幅によって制約されている場合、テクスチャからのフェッチする場合に起こることが最も多いですが、以下の最適化を検討してください。

- **テクスチャのサイズを小さくする。**ターゲット解像度とテクスチャ座標を検討します。お客様のユーザは最高のミップレベルに遭遇したことがあるでしょうか？遭遇したことがなければ、テクスチャのサイズの縮小を検討してください。これは、オーバーロードされたフレームバッファメモリが非ローカルメモリ（AGP または PCI-Express バス上のシステムメモリなど）からテクスチャリングの発生を強制している場合に特に有益です。NVPerfHUD メモリグラフには、ドライバーによって様々なヒープ中に割り振られたメモリの量が表示されるので、この問題を診断するのに役立ちます。
- **最小化可能なすべての表面に対して常にMIPマッピングを使用する。**MIPマッピングは、テクスチャエイリアシングを減らすことによって、画質を向上させます。ぼやけて見えない高品質のMIPマップを生成するために、様々なフィルターを使用することができます。NVIDIAは、Photoshopプラグイン、コマンドラインユーティリティーおよびライブラリなど（http://developer.nvidia.com/object/nv_texture_tools.htmlからすべてご入手いただけます）、最適なMIPマップの作製をお手伝いする一連のテクスチャツールをご用意しています。MIPマッピングがないと、テクスチャからのポイントサンプリングに限定されるため、望ましくないちらつきが生じる可能性があります。

> **Note:** 幾つかの表面に対するミップマッピングにより表面がぼやけて見える場合でも、ミップマッピングを無効にしたり、大きなネガティブ LOD バイアスをかけたりしないでください。これらの代わりに、アニソトロピックフィルタリングを使用してください。

- **カラーテクスチャをすべて圧縮する。**単にデカルやディテールテクスチャとして使用されているテクスチャは全て、テクスチャの具体的なアルファの必要性に応じて、DXT1、DXT3、または DXT5 のうちの 1 つを用いて圧縮すべきです。これにより、メモリの使用および必要なテクスチャ帯域幅が減少し、テクスチャキャッシュの効率が向上すると思われます。
- **不要であれば、高価なテクスチャフォーマットは避ける。**64 ビットや 128 ビットの浮動小数点フォーマットなどの巨大なテクスチャフォーマットには、そこからフェッチするのにずっと多くの帯域幅を要することは明らかです。必要な場合にのみこれらを使用します。

- **適切な異方性テクスチャフィルタリングレベルを使用する。**低周波数のテクスチャと高レベルの異方性フィルタリングを使用すると、GPU は、画質を向上させない余分な作業を行います。テクスチャの帯域幅が制約要因となっている場合には、十分な画質を与える最も低い異方性レベルを使用してください。アプリケーションはテクスチャ特異的な異方性設定を有しているのが理想的です。
- **不要な場合にはトリリニアフィルタリングを無効にする。**余分なテクスチャ帯域幅を消費していない場合でもトリリニアフィルタリングは、最も現代的な GPU アーキテクチャ上でのピクセルシェーダーにおいて演算するために余分なサイクルを負担しています。ミップレベルの遷移が容易に識別できないテクスチャ上では、トリリニアフィルタリングをオフにして、フィルレートを節約します。

第８項　フレームバッファの帯域幅を最適化する

パイプラインの最終段階である ROP は、フレームバッファメモリと直接連動しており、単一の段階としては、フレームバッファの帯域幅を最もたくさん消費します。このため、お手持ちのアプリケーションで帯域幅が問題となっていれば、ROP に問題の端緒が発していることが多いです。以下に、フレームバッファの帯域幅を最適化する方法を記載します。

- **まずデプスをレンダリングする。**これによってピクセルシェーディングの負担が軽減されるだけでなく（上記参照）、フレームバッファ帯域幅の負担も軽減されます。
- **アルファブレンディングを減らす。**アルファブレンディングは、フレームバッファへの読み書きを必要とするため、帯域幅を二倍消費する可能性があることに留意してください。アルファブレンディングを必要とする状態のみにアルファブレンディングを低減し、アルファブレンディングされた高レベルのデプス複雑性レベルには注意を払ってください。
- **可能であれば、デプスの書き込みをオフにする。**デプスに書き込みを行うと帯域幅がさらに消費されるので、アルファブレンディングされたエフェクトをレンダリングする場合やオブジェクトをシャドウマップ中にレンダリングする場合（カラーを使用したシャドウマップ中にレンダリングする場合、デプスの読取もオフにすべきです）には、マルチパスレンダリング（この場合、最後のデプスはすでにデプスバッファ中に存在する）中では無効にすべきです。
- **無用なカラーバッファのクリアを避ける。**お手持ちのアプリケーションによって、全てのピクセルがフレームバッファ中で確実にオーバーライトされるのであれば、カラーをクリアすると貴重な帯域幅を消費するので、カラーのクリアは避けるべきです。但し、可能であれば常に、デプスおよびステンシルバッファはクリアすべきです。早期 Z

の最適化の多くは、クリアされたデプスバッファの確定的な内容に依拠しているからです。

- **前から後ろにレンダリングを行う。**早期 Z ハードウェアの最適化によって、フレームバッファの無駄な読み書きを廃棄できるので、上述したように、前から後ろにレンダリングすると、ピクセルシェーディングに関して有利であるだけでなく、フレームバッファ帯域幅の面でも同じような有用性があります。実際、より多くのピクセルがデプステストをパスできないため、フレームバッファへのカラーおよびデプスの書き込みが減少するので、これらの最適化を備えていない旧式のハードウェアでも、この操作は有益なことがあります。
- **スカイボックスレンダリングを最適化する。**スカイボックスは、フレームバッファの帯域幅によって制約されることがしばしばありますが、これを最適化する方法を決定しなければなりません。スカイボックスを最後にレンダリングして、デプスを読み込み（但し、書き込みは行いません）、早期 Z の最適化が、定期的なデプスバッファとともに、帯域幅を節約することも可能ですし、まずスカイボックスをレンダリングしてから、すべてのデプスの読み書きを無効にすることもできます。これらのうち何れの技法がより多くの帯域幅を節約するかは、標的ハードウェアの関数であるとともに、ファイナルフレーム中で見ることのできるスカイボックスの数の関数です。スカイボックスの大部分が不明瞭な場合には、前者の技法が適している可能性が高く、それ以外の場合は、後者の技法でより多くの帯域幅を節約できると思われます。
- **必要な場合には、浮動小数点フレームバッファのみを使用する。**これらが、より小さな整数フォーマットに比べて、ずっと多くの帯域幅を消費することは明らかです。マルチプルレンダリングターゲットにも同じことが当てはまります。
- **可能であれば、16 ビットのデプスバッファを使用する。**デプスのトランザクションは莫大な帯域幅を使用するので、32 ビットに代えて 16 ビットを使用することが非常に有効な場合があり、ステンシルを必要としない小規模な屋内のシーンでは、これで十分なことが多くあります。デプスを必要とする render-to-texture エフェクト（ダイナミックキューブマップなど）の場合にも、十分なことが多いです。
- **可能であれば、16 ビットのカラーを使用します。**これは、特に、render-to-texture エフェクトの場合に当てはまります。ダイナミックキューブマップや投射されたカラーシャドウマップなど、これらのエフェクトの多くは、16 ビットカラーでも十分良好に機能するからです。

第 7 章
トラブルシューティング

ご質問やコメントがございましたら、弊社のデベロッパーフォーラムにてお尋ねいただくか、NVPerfHUD@nvidia.comまで内々にEメールをお送りください。

第 1 節　既知の事項

- NVPerfHUD は、複数の装置の処理は行いません。NVPerfHUD は、**`Direct3DCreate9()`**で生成された最初の装置のみをサポートします。
- ソフトウェアバーテックスプロセッシングを使用すると、NVPerfHUD はクラッシュすることがあります。
- **`rdtsc`** を元から使用しているアプリケーションは、NVPerfHUD のフレーム解析モードでは適切に機能できない可能性があります。考えられる解決策については、以下のトラブルシューティングをご覧ください。
- フレーム解析モードでは、お手持ちのアプリケーションが **`QueryPerformanceCounter()`**または **`timeGetTime()`** win32 関数を使用していなければなりません。特に **`dt`** がゼロのケースでは、お手持ちのアプリケーションは、経過時間 (dt)の計算を取り扱う上でロバストでなければなりません。換言すると、お手持ちのプログラムを **`dt`** で除してはいけません。これが問題であると疑われる場合には、NVPerfHUD Configuration ダイアログ中の Delta Time オプションを「SlowMo」に設定するように試みてください。これが奏功した場合には、**`dt`** がゼロのケースを、お手持ちのアプリケーションが取り扱えるように調節することを検討してください。
- インデックスユニット状態検査機能は点と線の表示をサポートしておりません。
- お手持ちのアプリケーションが固定 T&L を使用している場合には、状態検査機能は詳細な情報を表示しません。

- フレーム解析モードをご使用の場合、NVPerfHUD を（起動ホットキーを使用して）解除してから、お手持ちのアプリケーションを終了すると、「終了時のレファレンスナンバーが 0 ではありませんでした。」という警告ダイアログが表示されることがあります。
- NVPerfHUD が作動した状態で、クローズボタンをクリックすると、ウィンドウ表示モードで実行されているアプリケーションは適切に終了しないことがあります。

第２節　　よくあるご質問

あなたのアプリケーションは NVPerfHUD 解析を行えませんという表示が NVPerfHUD に出ます。

権限のない第三者が、あなたの許可なく、アプリケーションを解析しないようにするため、NVPerfHUD 解析を行えるようにするには、若干の変更を行わなければなりません。このユーザーガイドの「準備を整える」という章に説明が記載されておりますので、ご覧ください。

NVPerfHUD が作動していると、私のアプリケーションが応答しません。

ホットキー機能を用いて NVPerfHUD を作動させると、すべてのキーボード入力を消滅させ、キーストロークイベントをアプリケーションに一切伝えなくなります。お客様が選択された起動ホットキーを用いて、このモードの on/off を交互に表示させることができます。

スクリーン上部全体に NVPerfHUD ヘッダーが表示されていますが、起動ホットキーに対して応答しません。

NVPerfHUD では、キーストロークイベントを遮断するために複数の方法を使用しています。まだサポートされていない方法を使用されている場合には、お知らせください。NVPerfHUD をアップデートさせていただきます。

Win2K では、NVPerfHUD は DirecInput を使用して、起動ホットキーを認識しており、起動された状態の間は、キーボードのコマンドを遮断します。DirectInput は、2 種類のデータ、すなわちバッファに蓄積されたデータと即時データを与えます。バッファに蓄積されたデータとは、アプリケーションがこのデータを読み出すまで、保存されたイベントの記録です。即時データとは、デバイスの現在状態のスナップショットです。

このことは、ボトルネック特定実験やシェーダーの可視化など、NVPerfHUD の高度な機能を使用したい場合には、お手持ちのアプリケーションが、IDirectInputDevice8::GetDeviceState インターフェースの代わりに、IDirectInputDevice8::GetDeviceData インターフェースを使用する必要があることを意味しています。

NVPerfHUD は、アルファと一部のレンダリング状態とを混同しています。

NVPerfHUD は、フレームの最後で HUD をレンダリングします。また、NVPerfHUD は、それ自体を描くためにレンダリング状態を変更しますが、それらを元の状態に復活させません。換言すれば、NVPerfHUD は、性能上の理由で、レンダリング状態をプッシュ・アンド・ポップさせません。従って、お手持ちのアプリケーションによって、各フレームの冒頭でレンダリング状態がリセットされていることが想定されています。

GPU_IDLE (緑) の線が全くデータを表示していません。

GeForce4 (NV25)およびそれ以前の GPU では、この情報はご利用いただけませ

ん。NVPerfHUD がディスプレイドライバーと適切に交信できない場合には、これより新しい GPU でも同様のことが起こる場合があります。最新バージョンの NVPerfHUD を使用していること、最新の NVIDIA ディスプレイドライバーを走らせていることをご確認ください。

パフォーマンスグラフ中の色付きの線が表示されておらず、全てゼロとなっていますが、NVPerfHUD のヘッダーとバッチングヒストグラムは作動しています。

これらの機能は、DirectX 9 DEBUG ランタイムを使用しているときには、無効になっています。正確な性能解析を行うために、DirectX のコントロールパネルを使用して、DEBUG ランタイムの使用から RETAIL ランタイムに切り替えてください。お手持ちのアプリケーション中の機能的な問題を調べるために、NVPerfHUD デバッグコンソールモードを使用しているにすぎない場合には、DEBUG ランタイムをご使用になっても結構です。

NVPerfHUD グラフの真ん中に余分な線が幾つか表示されています。

（特に Win2K 上で）旧式のドライバーを走らせている場合には、NVPerfHUD の上に、以前の NVPerfHUD 1.0 グラフの一部が重ねて表示されていることがあります。ドライバーを 71.8x 以降のものにアップグレードすることで、問題は解決するはずです。

フレーム解析モードでも、シーン中のオブジェクトの一部がなおアニメーションを続けています。

フレーム解析モードに切り替えると、NVPerfHUD はお手持ちのアプリケーションへのクロックを停止させるので、カレントフレームがフリーズしている状態で、カレントフレームを詳しく解析することができます。お手持ちのアプリケーションは、QueryPerformanceCounter()または timeGetTime() win32 関数を使用している必要があります。お手持ちのアプリケーションが **rdtsc** インストラクションを使用していると、適切に機能しません。

アプリケーションがフレームを利用したアニメーションを使用している場合には、アニメーションされたオブジェクトに対してフリージングタイムは全く影響を与えないでしょう。

ここに掲載されていない問題に遭遇しました。

私どもは、アプリケーションの解析を行う開発者にとって、NVPerfHUD が常に有用なツールであり続けるようにしたいと願っております。何らかの問題に遭遇した方、あるいは NVPerfHUD を使用するときに有用な機能を思いつかれた方は、私どもにご連絡ください。

NVPerfHUD@nvidia.com

付属書類 A
ドライバーが GPU を待つ理由

GPU が最大限に活用されるシナリオというのは、2 つのプロセッサが FIFO によって接続されており、処理しきれないデータを、一方のチップが他方のチップに供給する場合に典型的に遭遇される状態です。

以下に示されているこのケースでは、CPU は、処理しきれないコマンドを GPU に供給しています。このような事態が生じると、すべてのコマンドは FIFO キュー（「プッシュバッファー」とも呼ばれます。）を形成し始めます。この FIFO がオーバーフローしないようにするためには、新しいコマンドを収容する余地が FIFO 中に発生するまで、ドライバーを強制的に待機させます。

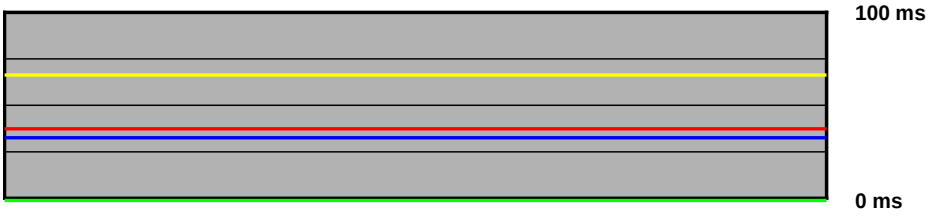


図２０　　GPU を待っているドライバー

フレーム速度が

- **高い場合**、CPU 上でさらに多くの作業を行うことができ、このようにしても、フレーム速度（オブジェクトカリング、フィジックス、ゲームロジック、AI など）に影響を及びません。
- **不十分な場合**、GPU 量への作業負荷を軽減するために、シーンの複雑性レベルを減らすべきです。

注意事項

全ての NVIDIA 設計仕様書、リファレンスボード、ファイル、図面、診断プログラム、リストおよびその他の文書 (合わせて、および、単独で、「素材」という) は、現状のままの状態で提供されるものです。NVIDIA は、本素材に関して、明示的、黙示的、法的にかかわらず一切保証を行わず、すべての不侵害、商品性、および特定目的適合性の黙示の保証責任を負わないものとします。

記載されている情報は正確であり信頼できるものです。しかし、NVIDIA Corporation はこれらの情報の使用の結果に対して、またはこれらの情報の使用により起こりうる第三者の特許もしくは他の権利の侵害に対して、いかなる保証も行うものではありません。黙示的であるかどうかを問わず、NVIDIA Corporation のあらゆる特許または特許権のもと、ライセンスを許諾するものではありません。この出版物で述べている仕様は予告なく変更されることがあります。この出版物は、以前に出された情報全てに対し優先するものであり、置き換わるものです。NVIDIA Corporation 製品は、NVIDIA Corporation からの文書による明確な承諾なく、人命に関わる装置またはシステムにおける重要な部品として使用することはできません。

商標

NVIDIA および NVIDIA 口ゴは、NVIDIA Corporation の登録商標です。
その他の会社名および製品名は、関連する各社の商標である可能性があります。

著作権

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