用 NVPerfHUD 优化图形芯片性能

显示性能分析所需的信息

2005 年 4 月

DEVELOPMENT

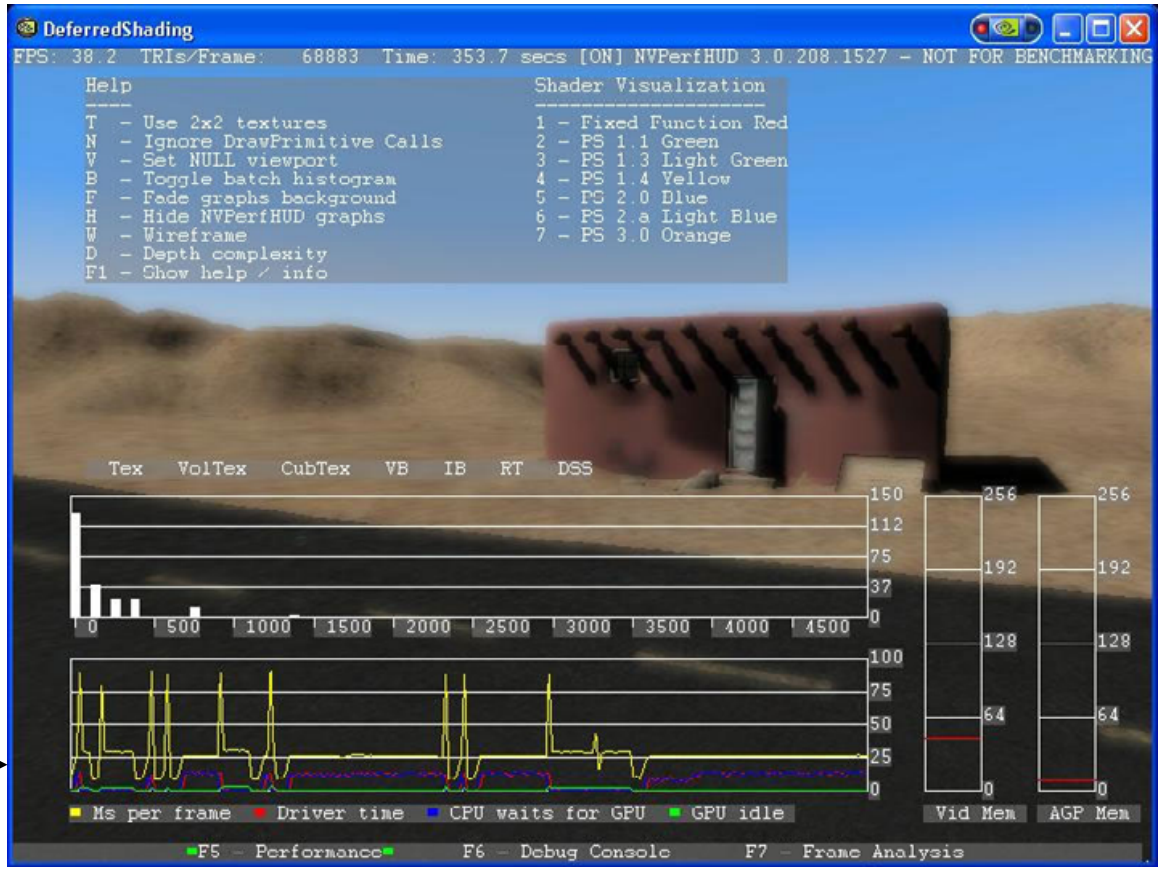
目录

Chapter 1.
第 1 章
关于 NVPerfHUD

现代图形芯片（GPU）通过按管线顺序进行的操作产生影像。管线的最高速度取决于最慢一级的速度，因此要想将图形应用程序的性能调整到最佳状态，就需要采用基于管线的方法进行性能分析。NVPerfHUD 能够找出图形管线中最慢的一级，从而帮助您提高应用程序的整体性能。

NVPerfHUD 允许您以每次一级的方式分析应用程序的性能，并显示实时统计数据，这些数据可以用来诊断 Direct3D 9 应用程序中任何一级的性能瓶颈及功能问题。

启用 NVPerfHUD 以后，Direct3D 应用程序的顶部将出现一张图形覆盖图，如图 1 所示。



显示性能的图形覆盖图

图 1. 启用了 NVPerfHUD 的 Direct3D 应用程序

1.1. 系统要求

- 任意一款 NVIDIA 图形芯片（GeForce 3 以上）
  建议使用 GeForce6 系列或更高的图形芯片
  GeForce 3 以前的图形芯片只支持部分功能
- NVIDIA 显示驱动程序 71.80 或以后的版本
- Microsoft DirectX 9.0c
- Windows 2000 或 Windows XP

1.2. 推荐阅读

- NVIDIA 开发者网站
  http://developer.nvidia.com
- 《平衡图形管线以获得最佳性能》 — 白皮书 [链接]
- 《NVIDIA 图形芯片编程指南》 — 所有最新技巧及诀窍 [链接]
- NVIDIA FX Composer — 着色器开发环境 [链接]
- NVShaderPerf — 着色器性能分析工具 [链接]
- NVIDIA 软件开发工具包 — 数百个示例代码及效果 [链接]
- NVTriStrip — 创建能识别顶点高速缓存的条带 [链接]
- 《GPU Gems：实时图形的编程技巧及诀窍》 [链接]
  其中关于性能的几个章节特别有用
- 《GPU Gems 2：高性能图形及通用目的计算的编程技巧》[链接]
  Microsoft DirectX 网站 [链接]
- 微软开发者网络（MSDN）网站 [链接]
  搜索“性能”和“优化”
- Microsoft DirectX 软件开发工具包文档 — 安装后在开始菜单中

Chapter 2.
第 2 章
开始

NVPerfHUD 使用显示驱动程序中特殊的性能监测程序从图形芯片及驱动程序内部直接收集数据。NVPerfHUD 还通过 API 侦听收集各种数据并与应用程序相互作用。NVPerfHUD 需要这些监测技术才能正常工作，同时这些技术也会占用额外的资源（小于 7%）。

如果用 NVPerfHUD 启动 Direct3D 应用程序，您将在 DirectX 图形的顶部看见 NVPerfHUD 用户界面。通过激活热键您可以在应用程序和 NVPerfHUD 之间切换。

为您的应用程序启用 NVPerfHUD 分析并按照下面的“快速开始”流程对应用程序进行设置并与 NVPerfHUD 一起运行。

2.1 准备应用程序

NVPerfHUD 是一个功能强大的性能分析工具，能够帮您了解应用程序的内部功能。为了防止未经授权的第三方不经您同意就对您的应用程序进行分析，您必须稍做修改才能启用 NVPerfHUD 分析。关于如何让应用程序与 NVPerfHUD 更加协调的工作的详情，请阅读本文档末尾的故障排除一章。

注意：在您的应用程序发售以前请务必禁用 NVPerfHUD 分析，否则任何人都可以使用 NVPerfHUD 对您的应用程序进行分析。

建立图形管线的第一步是调用 Direct3D CreateDevice（）函数创建显示设备。应用程序中的代码可能如下所示：

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

当应用程序被 NVPerfHUD 启动以后，将会创建一个特殊的 **NVIDIA NVPerfHUD** 适配器。应用程序可以选择该适配器以允许 NVPerfHUD 对自己进行分析。此外，有些应用程序可能会无意中选中 **NVIDIA NVPerfHUD** 适配器 ID，从而导致未经授权的分析。为了防止这种情况，您必须将设备类型选为“D3DDEVTYPE_REF”。只要您选择了 NVPerfHUD 适配器，应用程序就不会实际使用参考光栅器。

在应用程序中启动 NVPerfHUD 分析的最短代码可能如下所示：

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

使用最后一个适配器（通过调用 **GetAdapterCount（）-1**，如上所示）即是假设 NVPerfHUD 创建的 **NVIDIA NVPerfHUD** 适配器标识符将位于列表的最后。

注意： 为应用程序启用 NVPerfHUD 分析只需要选择 NVIDIA NVPerfHUD 适配器并将 **DeviceType** 标记设为 **D3DDEVTYPE_REF**。如果您只更改了其中一个参数，NVPerfHUD 分析将不被启用。

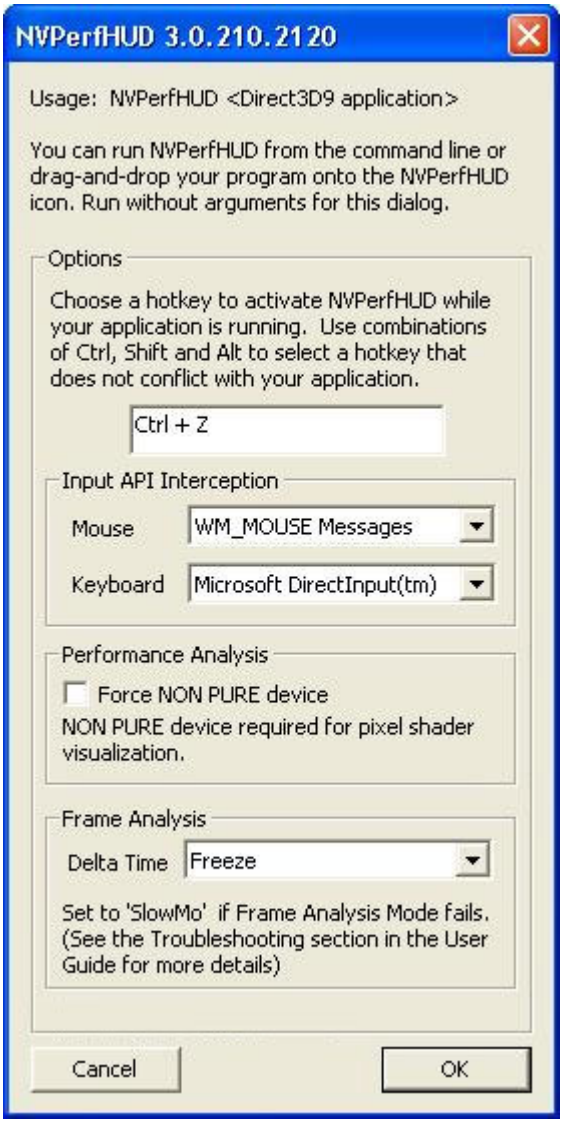
您将看到，如果 **NVIDIA NVPerfHUD** 适配器不可用，这种快速实现方法将会导致应用程序使用软件参考光栅器。为了避免这种情况，我们建议您用以下代码替换应用程序中对 **CreateDevice**（）函数的调用。

```
// 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;
}
```

这将在您需要时启用 NVPerfHUD 分析，同时确保应用程序不会在正常运行时使用软件参考光栅器。请按照以下快速开始流程设置应用程序与 NVPerfHUD 一起运行。

2.2. 快速开始



当您的应用程序和 NVPerfHUD 一起运行时，应用程序顶部会出现默认图表及信息。在安装时您将如下所述设置激活热键，NVPerfHUD 的高级功能将通过该热键使用。

1. **安装 NVPerfHUD**
   安装程序将在桌面上放置一个新的图标。
2. **运行 NVPerfHUD**
   第一次运行 NVPerfHUD Launcher 时将自动出现一个配置对话框。每当您运行 Launcher 而不指定要分析的应用程序时都会看到这个配置对话框。
3. **选择激活热键**
   请确定该热键不会与应用程序使用的键发生冲突。
4. **配置 API 侦听**
   告诉 NVPerfHUD 如何捕获鼠标及键盘事件。如果您的应用程序使用的方法不被支持，您将不能使用键盘和/或鼠标。

   注意：在使用激活热键激活 NVPerfHUD 以后，所有键盘事件都会被 NVPerfHUD 捕获。

5. **可选项：启用 NVPerfHUD 的 Force NON PURE device 功能。**
   如果您的应用程序正在使用 PURE 设备，则必须选中 **Force NON PURE device** 功能复选框。如果您的应用程序正在使用 PURE 设备而该复选框又未被选中，您将不能使用帧分析模式或性能模式下的象素着色器视图。

6. **将您的应用程序拖放到 NVPerfHUD 桌面图标上。**
   点击 OK 确认您的配置选项，然后将您的.EXE、.BAT 或.LNK（快捷方式）文件拖放到 NVPerfHUD Launcher 图标上。您也可以从命令行运行 NVPerHUD.exe 文件并将要分析的应用程序指定为命令行参数。有些开发者选择创建批处理文件或修改 IDE 设置，以便让其自动执行。
7. **可选项：如果您的应用程序崩溃或在帧分析模式下出现问题，请将“Delta Time”设置更改为 SlowMo。**这将有助于确定是 NVPerfHUD 的问题还是应用程序的问题。更多信息请参看故障排除一章。

注意：运行 NVPerfHUD 时如果不指定应用程序，将会出现配置对话框。

现在您应该看到应用程序的顶部出现默认的 NVPerfHUD 图表及信息。使用您选择的激活热键对 NVPerfHUD 进行操作，按 **F1** 键将显示帮助信息。再次使用热键将控制返回到应用程序。

注意：为了确保您能准确找出应用程序的瓶颈，NVPerfHUD 将 PresentationInterval 设为 D3DPRESENT_INTERVAL_IMMEDIATE，以强制关闭垂直刷新同步。

2.3. 基本工作流程

NVPerfHUD 激活以后，您能够进行图形管线测试、显示性能数据图表，并用数种性能视图模式查看潜在的问题。您还可以从默认的性能分析模式切换到调试控制台模式或帧分析模式。调试控制台显示了来自于 DirectX 调试运行时的消息、NVPerfHUD 的警告以及来自应用程序的客户消息。在帧分析模式下您可以使用高级状态检测器查看图形管线中每一级的详情。

当 NVPerfHUD 激活以后，您可以用以下键在这些模式之间切换：

F5 **性能分析**
使用时间图表及定向测试来找出瓶颈

F6 **调试控制台**
查看来自于 DirectX 调试运行时的消息、NVPerfHUD 警告以及来自应用程序的客户消息。

F7 **帧分析**
定格当前帧并以每次一个绘图调用的方式逐步查看该帧，使用如下所述的状态检测器查看图形管线中每一级的情况。

Chapter 3.
第 3 章
性能分析模式

本章讲述了如何理解 NVPerfHUD 在性能分析模式下显示的信息，以及如何使用定向性能测试来找出并优化应用程序的性能瓶颈。

图 2 显示了性能分析模式下的图表及覆盖图

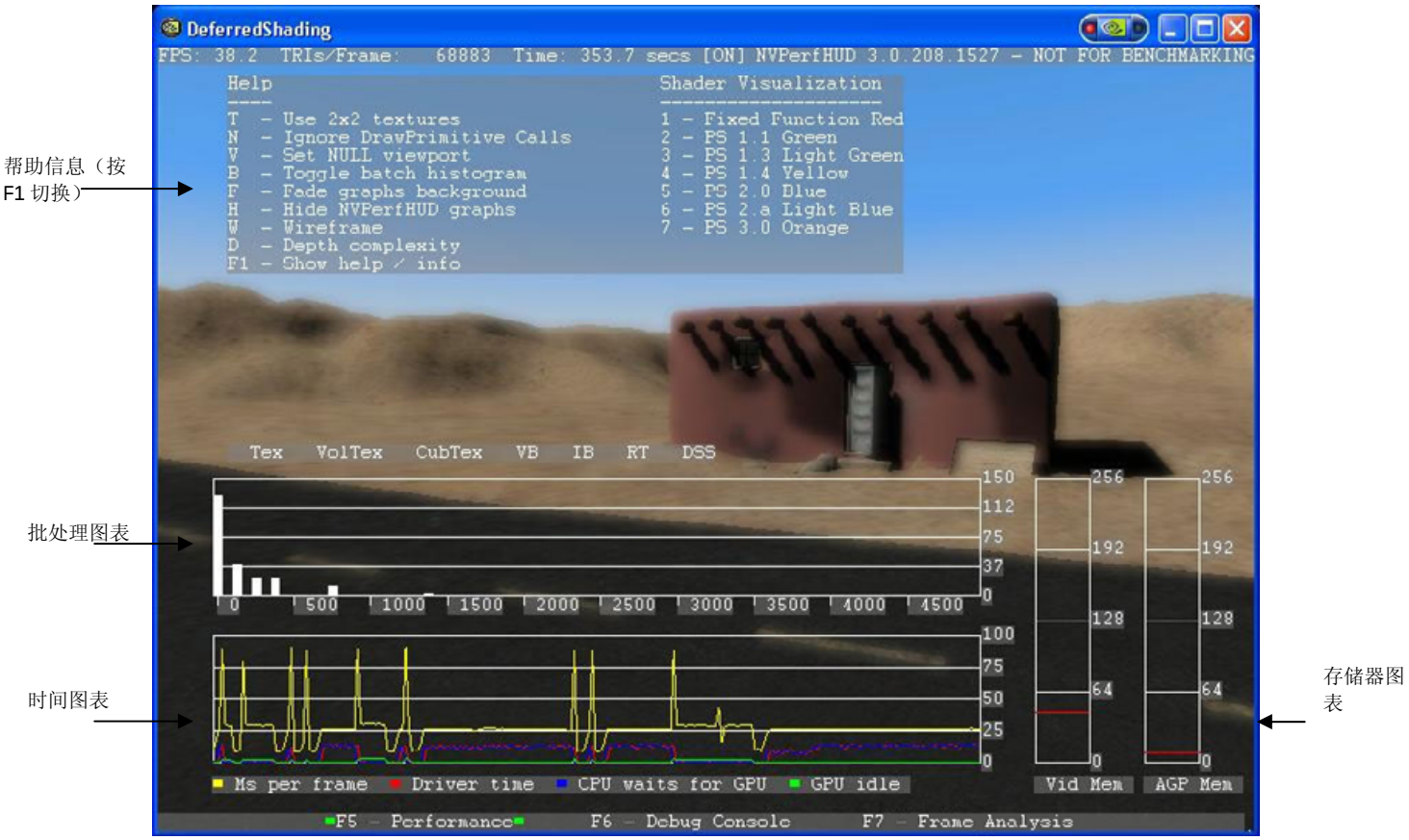


图 2. NVPerfHUD 性能分析模式

3.1. 性能图表

当您首次启动应用程序时，NVPerfHUD 在默认情况下会显示几个图表及基本性能数据。

通过激活热键及下列选项，还可以根据需要显示额外的图表及信息：

- **F1** **F1** 循环显示帮助信息
- **B** **B** 切换批处理大小柱状图
- **F** **F** 将背景变淡以提高图表的可读性
- **H** **H** 隐藏图表
- W W 线框
- **D** **D** 深度复杂性

注意：如果启用了 DirectX 调试运行时，则时间图表及定向测试将被禁用。由于调试运行时环境需要占用额外的资源，同时考虑到其性能特征，因此它不适用于性能分析模式。这时会显示一条警告信息，但绘图调用图表、批处理柱状图及存储器图表仍然可用。

3.1.1. 阅读图表

该模式下显示的数据被分为以下消息区：

每秒帧数（FPS）及三角形/帧

基本性能数据显示在屏幕的左上角（见图 2）。这两个数字可以用来衡量应用程序完成任务的速度。

图 3. NVPerfHUD 信息条（顶部）

滚动曲线图

这些曲线图就像心率监护仪一样，将每一帧从右向左滚动，使您能够看见波形随着时间的变化。

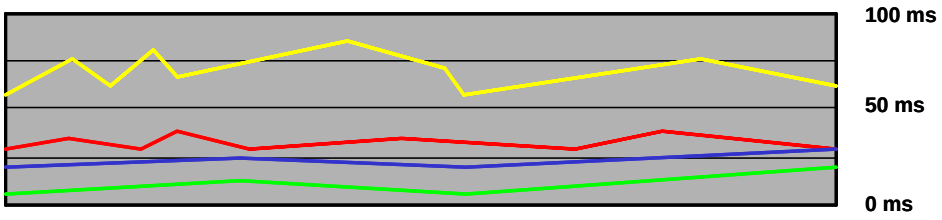


图 4. 性能图表

- 绿色 = GPU_IDLE（图形芯片空闲）
  每一帧中图形芯片空闲的总时间
- 蓝色 = DRIVER_WAITS_FOR_GPU（驱动程序等待图形芯片）
  驱动程序等待图形芯片的累计时间
  （要想了解为什么会发生这种情况，请参看附录 A）
- 红色 = TIME_IN_DRIVER（驱动程序的时间）
  每一帧中 CPU 执行驱动程序代码的总时间，包括 DRIVER_WAITS_FOR_GPU （蓝色）
- 黄色 = FRAME_TIME（帧时间）
  从一帧结束到下一帧的时间 — 您应该让 FRAME_TIME（黄色）线尽可能的低。

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

注意：FRAME_TIME（黄色）线与 TIME_IN_DRIVER（红色）线之间的时差是被应用程序和操作系统占用的时间。

您可能会看到一些偶发高峰，它们是由后台运行的操作系统进程引起的。下图显示了一个由硬盘访问、纹理上载、操作系统纹理切换等操作引起的帧速率突然升高。

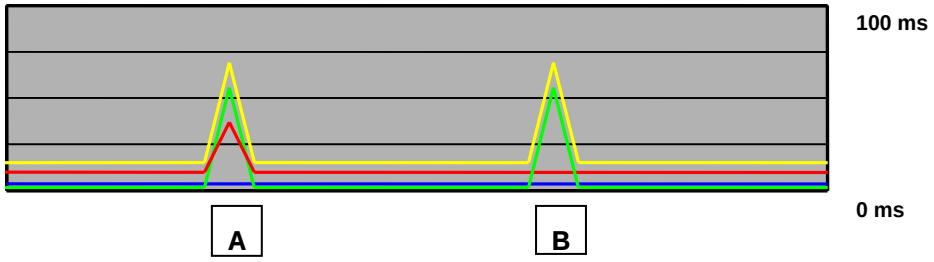


图 5. 偶发峰值

零星的偶发峰值是正常现象，但您应该知道峰值产生的原因。如果经常产生高峰，那可能说明您的应用程序在执行需要占用大量 CPU 资源的操作时效率太低。

- 类型 A：
  如果 TIME_IN_DRIVER（红色）线和 FRAME_TIME（黄色）线同时出现高峰，那可能是驱动程序正在将纹理从 CPU 上载到图形芯片。
- 类型 B：
  如果 FRAME_TIME（黄色）线出现高峰而 TIME_IN_DRIVER（红色）却没有，那可能是您的应用程序正在执行一些占用 CPU 资源较多的操作（例如音频解码）或正在访问硬盘。当操作系统处理其他进程时也会产生这种现象。

请注意，绿线在两种情况下都可能形成高峰，原因是您没有向图形芯片发送数据。

Draw Primitives 图表

该图表显示了每一帧调用 DrawPrimitive、DrawPrimitiveUP 及 DrawIndexedPrimitives（DP）的次数。在下一页及第 6 章中我们将讲述如何利用该信息来找出性能瓶颈。

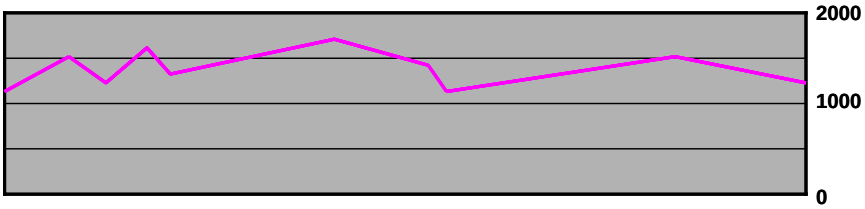


图 6.　DP 调用图表

批处理大小柱状图

只有当您将批处理大小图表打开时它才会出现。打开它的方法是激活 NVPerfHUD（用激活热键）并按 B 键。图中的柱表示批处理的数量。左边第一个柱表示拥有 0 到 100 个三角形的批处理的数量，第二个柱表示拥有 100 到 200 个三角形的批处理的数量，依此类推。如果您用了很多小的批处理，那么左边的柱将变高。

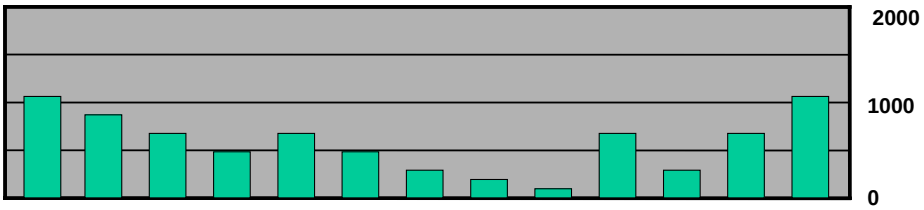


图 7.　DP 调用柱状图

存储器图表

该图表以兆为单位显示了由驱动程序分配的 AGP 及视频存储器。

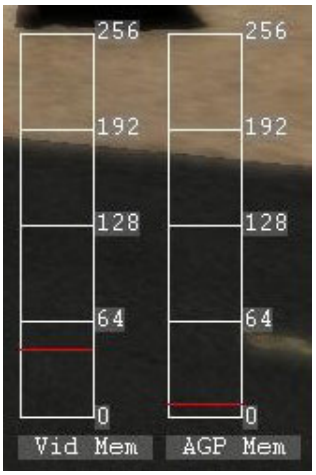


图 8. 存储器图表

3.1.2. 资源创建监视器

每当有资源被创建，资源创建监视器的指示灯就会闪烁。在 Direct3D 中动态创建资源通常会对性能带来负面影响，因此应该尽可能避免。

图 9. 资源创建监视器

被监视的资源创建事件的类型有：

TEX 使用 CreateTexture（）创建 2D 纹理

VolTex 使用 CreateVolumeTexture（）创建 3D 体积纹理

CubTex 使用 CreateCubeTexture（）创建立体贴图

VB 使用 CreateVertexBuffer（）创建顶点缓冲

IB 使用 CreateIndexBuffer（）创建索引缓冲

RT 使用 CreateRenderTarget（）创建着色器目标

DSS 使用 CreateDepthStencilSurface（）创建深度模板表面

注意： 资源创建事件还会被记录到调试控制台（F6）中，因此您可以看到引起了指示灯闪烁的具体事件。

3.2. 管线测试

要找出性能瓶颈就需要按每次一级的方式查看图形管线中的某些级。NVPerfHUD 允许您进行下列测试：

- **T — 隔离纹理单元**
  强制图形芯片使用 2x2 纹理，如果帧速率明显提高，则说明您的应用程序性能受到了纹理带宽的限制。
- **V— 隔离顶点单元**
  使用 1x1scissor rectangle 剪裁管线级中位于顶点单元之后的所有光栅及着色工作。这种方法近似于截去顶点单元之后的图形管线，可以用来判断应用程序受到的限制来自于顶点转换、CPU 负载和/或总线处理。
- **N — 消除图形芯片**
  这项测试将忽略所有 DrawPrimitive（）及 DrawIndexedPrimitives（）调用，使图形芯片的速度接近无穷大。如果整个图形管线没有性能资源占用，则帧速率将接近应用程序所能达到的最高速度。请注意，状态改变所引起的 CPU 资源占用也将被忽略。

您还可以有选择性的禁用某些版本的象素着色器并观察它们是如何在应用程序中使用的。当某个着色器版本被禁用时，那一组中所有着色器都将用同一种颜色表示。下面列出了 NVPerfHUD 提供的着色器视图选项：

- 1 — 固定功能（红色）
- 2 — 1.1 象素着色器（绿色）
- 3 — 1.3 象素着色器（浅绿色）
- 4 — 1.4 象素着色器（黄色）
- 5 — 2.0 象素着色器（蓝色）
- 6 — 2.a 象素着色器（浅蓝色）
- 7 — 3.0 象素着色器（橙色）

注意： 只有当 Direct3D 设备被创建为 NON PURE 设备时，着色器视图才能工作。您可以在 NVPerfHUD 配置设定中强制在 NON PURE 设备模式下创建设备。

Chapter 4.
第 4 章
调试控制台模式

本章讲述了调试控制台模式下为您提供的信息。调试控制台画面如图 10 所示。

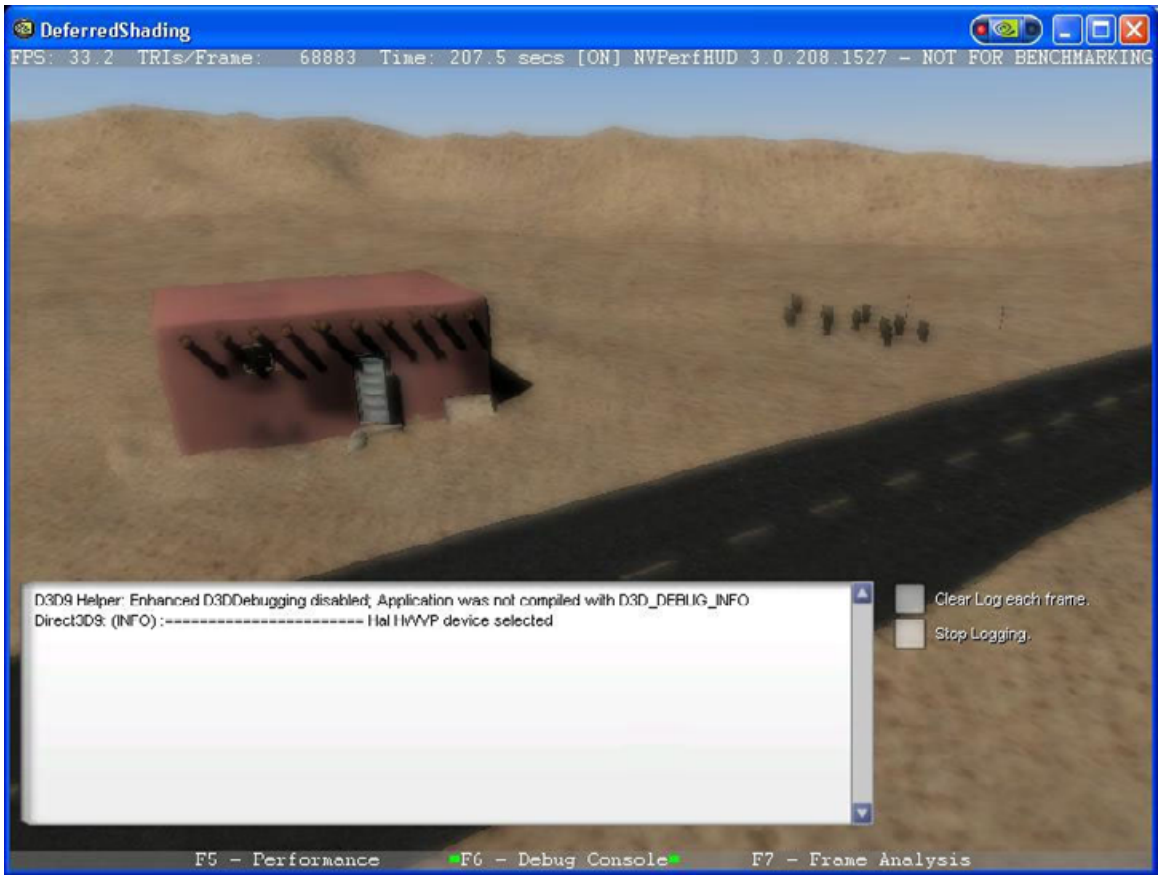


图 10. 调试控制台模式

调试控制台显示了所有通过 DirectX 调试运行时报告的消息、应用程序通过 OutputDebugString（）函数报告的消息、以及 NVPerfHUD 检测到的其他警告或错误。

资源创建事件及 NVPerfHUD 检测到的警告也会被记入控制台窗口。

您可以使用下列选项自定义调试控制台的工作方式：

- **C** 每帧清除记录
- **S** 停止记录
- **F** 将控制台淡化

启用“每帧清除记录”复选框以后，每一帧开始时控制台窗口的内容将被清除，因此您只能看到当前帧生成的警告。如果您的应用程序生成的警告太多，控制台窗口无法全部显示，您可以使用该功能。

启用“停止记录”复选框以后，控制台会停止显示新的消息。

Chapter 5.
第 5 章
帧分析模式

本章解释了如何更有效的利用帧分析模式及其高级图形管线状态检测器。图 11 显示了帧分析模式的画面。

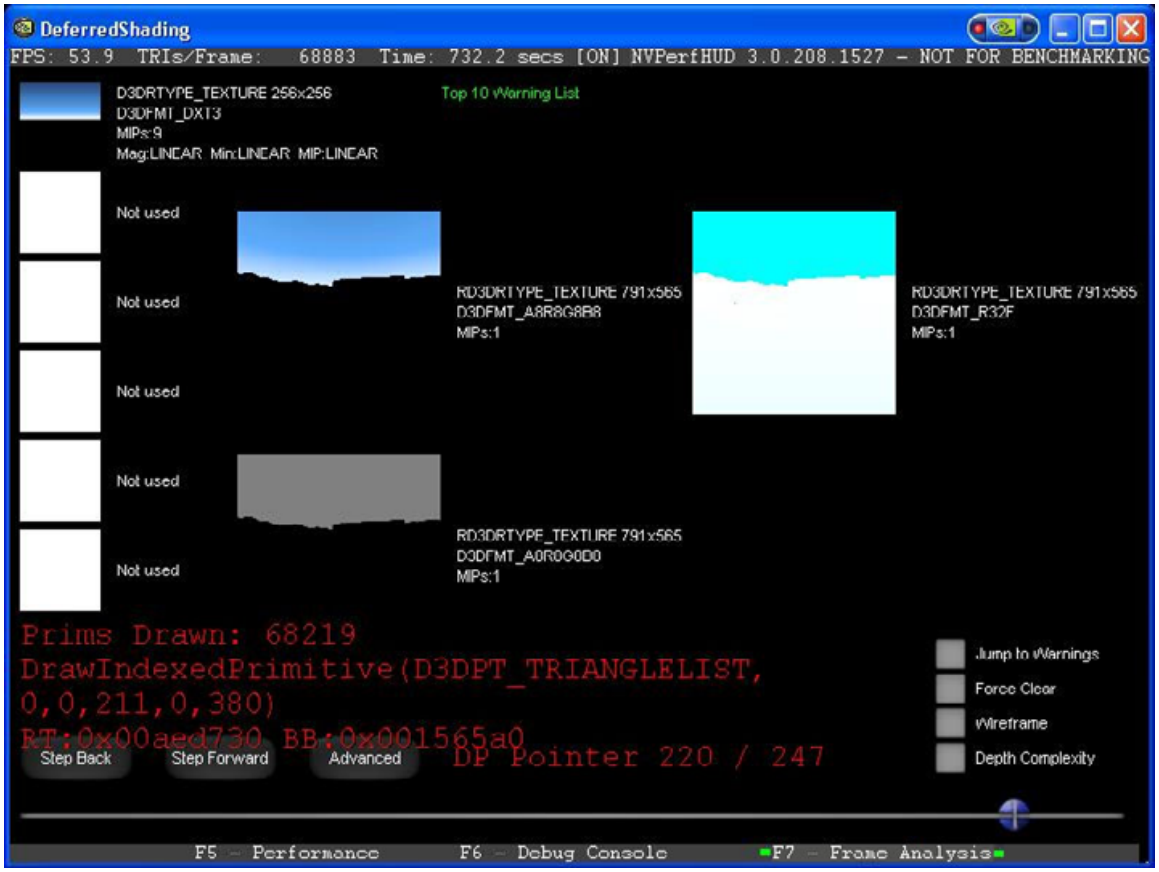


图 11. 帧分析模式

您可以使用左/右方向键显示上一次/下一次绘图调用，还能使用下列选项配置帧分析模式：

- **A** 切换高级/简单视图
- **S** 显示警告
- **W** 线框

- **D** 深度复杂性

5.1. 着色分解

当您找到含有着色伪迹的帧以后，需要检验场景绘制的顺序或进一步了解引起警告的原因时，您可以使用帧分析模式跳到帧内的任一次绘图调用。当您切换到帧分析模式以后，NVPerfHUD 将停止为应用程序计时，您也可以利用当前帧被定格的时间进行深入分析。如果您的应用程序使用了基于帧的动画，冻结时间对动画对象无效。

注意： 为了有效的使用帧分析模式，您的应用程序必须以 NVPerfHUD 能够控制的方式运行。下面列出了几点要求。更多问题请参看本文档末尾的故障排除一章。

帧分析模式要求您的应用程序使用并依赖 QueryPerformanceCounter（）或 timeGetTime（）win32 函数。您的应用程序必须正确无误的处理已用时间（dt）的计算，特别是当 dt 为零时。换句话说，您的程序不应该除以 dt。

当应用程序被冻结以后，请使用键盘的**下一个（右方向键）**及**前一个（左方向键）**来逐个查看帧内的所有绘图调用。您也可以拖动屏幕底部的滑块或使用 **PgUP/PgDn** 键跳到某个绘图调用。如果鼠标或键盘事件侦听不正常，请退出应用程序并在 NVPerfHUD 配置对话框中换一个 API 侦听选项。

每一个绘图调用所显示的信息包括：

- 刚刚绘制了哪一个绘图调用，以及共有多少次调用。
- 上一次绘图调用的函数名及参数。如果绘图调用产生了警告，则警告消息也会被显示。

5.1.1. 显示警告

启用 **Show Warnings（S）**以后，屏幕顶部的列表框中将显示一个警告列表，包括被 DP 调用锁定的占用资源最多的顶点缓存（VB）。点击某个警告将跳至相关的 DP 调用。您还可以用**上/下方向键**滚动列表。当显示警告时点击**下一个（右箭头）**或**前一个（左箭头）**将移动屏幕底部的滑块，跳至引起警告的绘图调用。

您应该检查每一个 VB 锁定，确保自己了解已锁定时间并使之尽可能的小。这有助于您了解应用程序在建立顶点缓存时 CPU 时间都用在了什么地方。

5.1.2. 纹理单元及 RTT 信息

关于每一个纹理单元及画外着色器到纹理（RTT）目标的信息包括：

- 每个纹理单元中存储的纹理及其属性：
  - 尺寸
  - 过滤参数：缩小倍数、放大倍数以及 MIP 级别
  - 纹理格式：RGBA8、DXT1、 DXT3、 DXT5 等等
  - 纹理目标：1D、2D、3D 体积纹理、立体纹理、 NP2 等等
- 如果当前绘图调用正在为画外纹理（RTT）着色，则该纹理的内容及属性将显示在屏幕中央。

当使用的纹理太多，屏幕无法全部显示时，请使用滚动条滚动列表。

5.1.3. 视图选项

帧分析模式支持几种视图选项：

- **D 深度复杂性**：该选项将打开附加的微红色混合。屏幕越明亮，使用的帧缓冲就越大。帧缓存读-修改-写（RMW）每增加 1，色值就会增加 8，RMW 最大可增加 32。当屏幕色彩饱和以后（255），您的应用程序就会过于频繁的向帧缓冲中写入过多数据，这会导致由于写满限制引起的性能瓶颈。
- **W 线框**：该选项强制使用线框着色，以便您能检查场景的几何复杂度。

5.1.4. 高级状态检测器

点击 **Advanced…（A）**按钮将会激活高级状态检测器。这时屏幕底部用于导航的滑块仍然可用，但屏幕顶部将出现几个与图形管线中每一级相对应的按钮。您可以点击按钮或用快捷键在检测器之间切换。

- **1** 索引单元 — 获取顶点数据
- **2** 顶点着色器 — 执行顶点着色器
- **3** 象素着色器— 执行象素着色器
- **4** 光栅操作 — 帧缓冲的着色后操作

> 注意： 关于显示在每一个 NVPerfHUD 状态检测器中的状态信息的详情，请参阅随最新版本 DirectX SDK 一起安装的文档。

点击每一级将显示在当前绘图调用中该级的详细活动情况。以下几节介绍了每一种状态检测器显示的信息。

5.2. 索引单元状态检测器

该检测器被选中以后将显示当前绘图调用中索引单元的信息。

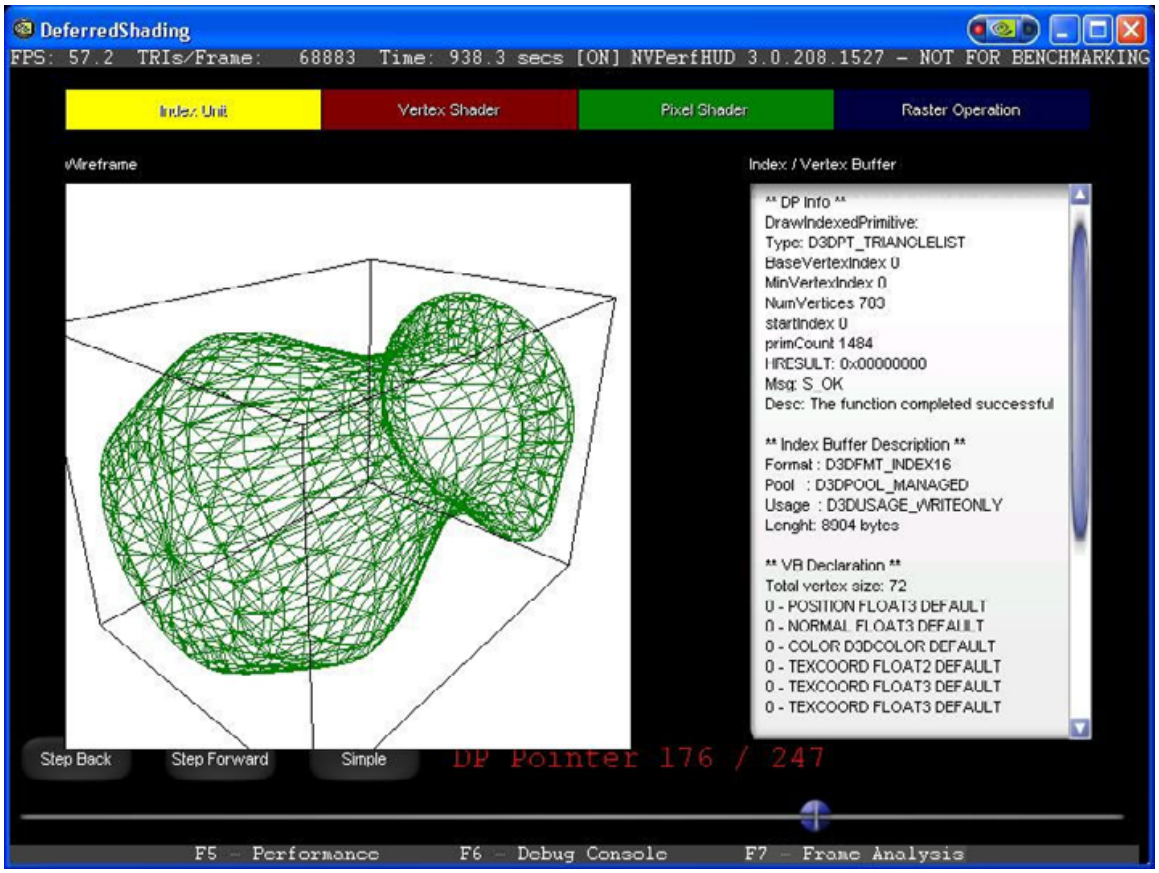


图 12. 索引单元状态检测器

在屏幕中央的边界框中显示了一个与本次绘图调用有关的旋转的几何线框着色图。

> 注意： 该版本不支持点和线。

在它旁边有一个列表框，显示了为本次绘图调用获取顶点数据所需的所有信息，包括：

- 绘图调用参数及返回标志
- 索引及象素缓冲格式、大小等等
- 可变顶点格式（FVF）

要想使用索引单元状态检测器，您应该看看线框着色以确定应用程序发送的批处理是否正确。例如，如果在进行矩阵调色盘变形处理（Matrix Palette

Skinning）时着色被中断，您应该检查顶点缓冲/索引缓冲中的参考相对位置是否正确。如果参考相对位置正确，那么导致场景中这种几何图形着色失败的原因可能是顶点着色器或顶点权重的问题。

您还应该确认索引的格式是否正确，在可能的情况下尽量使用 16 位索引。

5.3. 顶点着色器状态检测器

该状态检测器被选中以后，将显示当前绘图调用期间顶点着色器的信息。

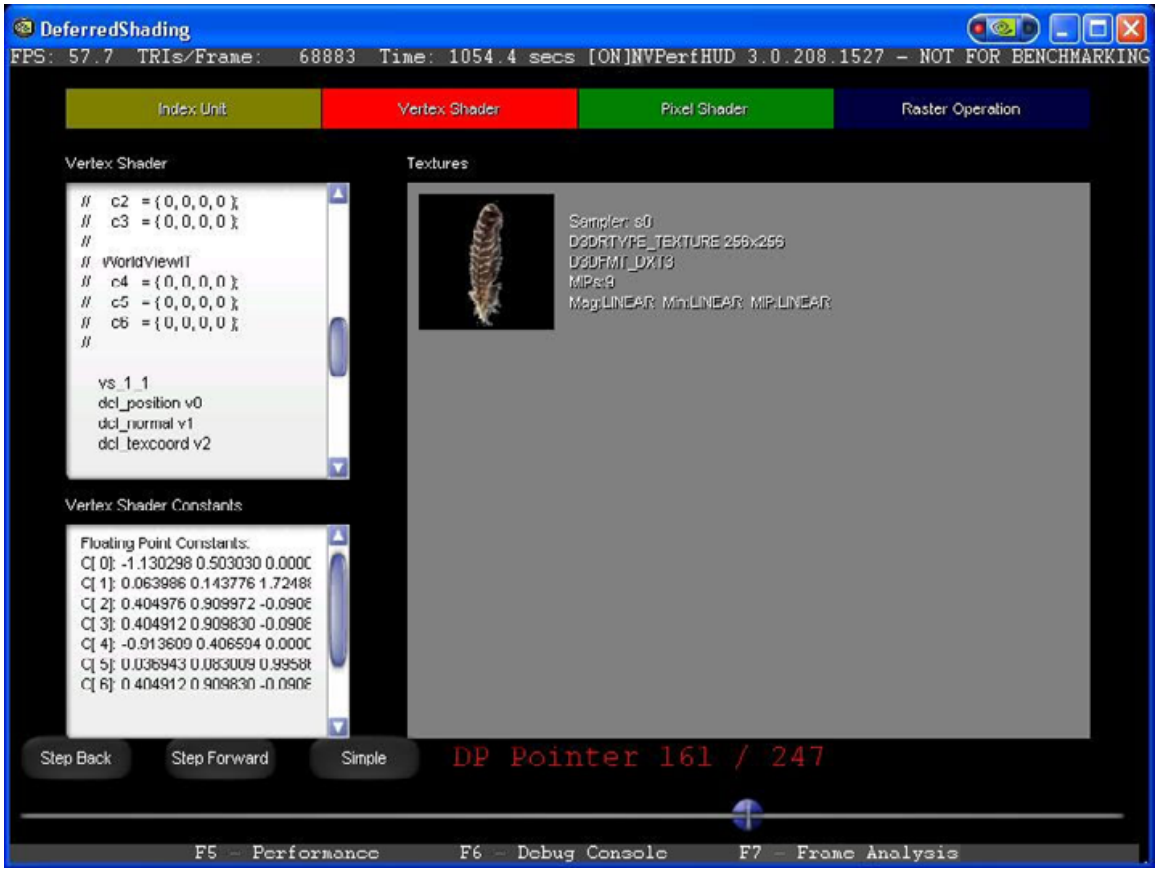


图 13. 顶点单元状态检测器

顶点着色程序及其使用的所有常量和纹理都会被显示出来以供检查。当顶点着色器使用地址寄存器时（例如矩阵调色盘变形处理），所有常量都会显示出来。每个纹理取样器也会被显示出来以供参考。使用+/-键可以放大和缩小显示的纹理。

使用顶点着色器状态检测器时您应该：

- 确认在当前绘图调用中应用了预期的顶点着色器

- 确定常量没有传递**#NAN** 或**#INF**

5.4. 象素着色器状态检测器

该状态检测器被选中以后，将显示当前绘图调用期间象素着色器的信息。

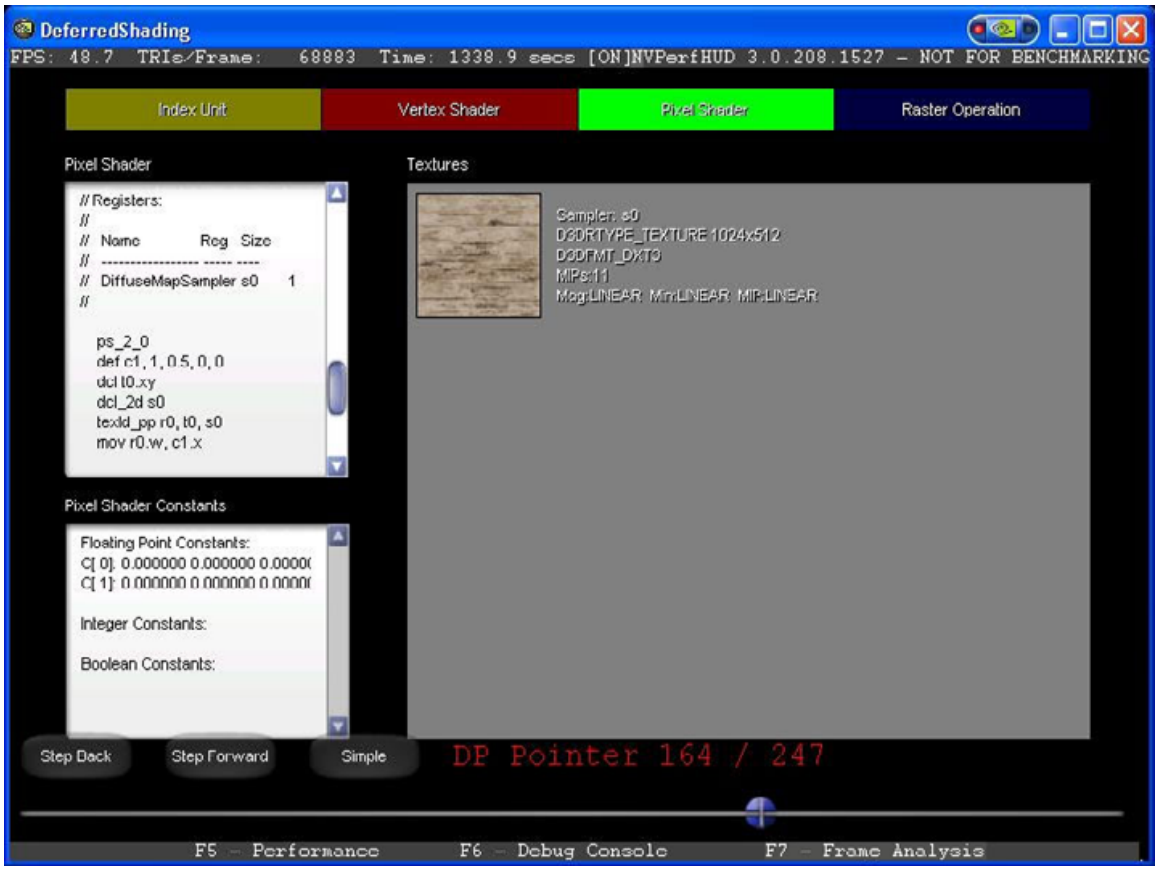


图 14. 象素着色器状态检测器

象素着色程序及其使用的所有常量和纹理都会被显示出来以供检查。每个纹理取样器也会被显示出来以供参考。使用+/-键可以放大和缩小显示的纹理。

使用象素着色器状态检测器时您应该：

- 确认在当前绘图调用中应用了预期的象素着色器。
- 确定常量没有传递**#NAN** 或**#INF**
- 确定纹理及着色器至纹理（render-to-texture）纹理被正确使用。

5.5. 光栅操作状态检测器

当该状态检测器被选中以后，将显示当前绘图调用期间光栅操作（ROP）的信息。

注意：光栅操作状态检测器中显示的关于状态信息的更多详情，请参看随最新版本 DirectX SDK 一起安装的文档。

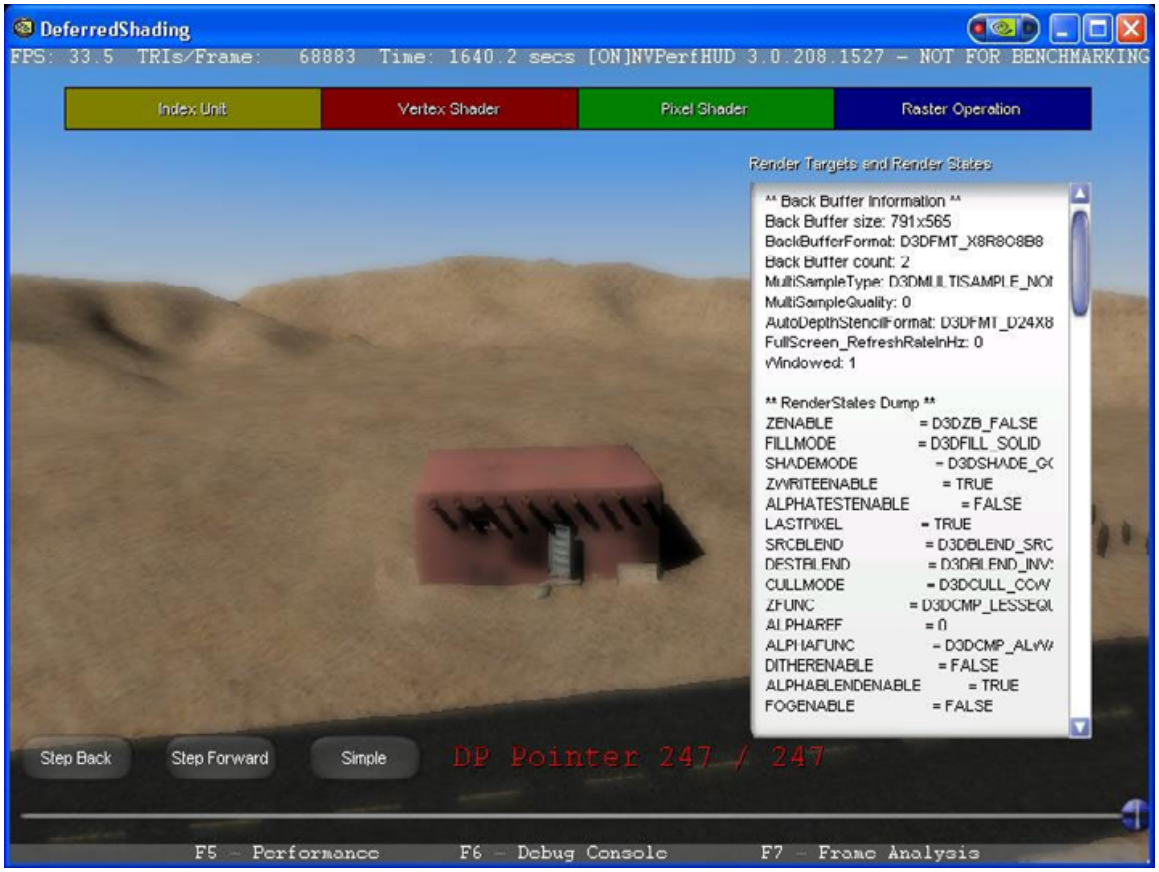


图 15：光栅操作状态检测器

本次绘图调用期间着色后光栅操作的信息将被显示出来以供检查。这些信息包括：

- 着色目标格式
- 后台缓存格式
- 导致帧缓冲处理占用大量资源的着色器状态包括：
  - **Zenable** — Z 比较操作
  - **Fillmode** — 光栅化模式
  - **ZwriteEnable** — 在深度缓冲中写入或不写 Z

- **AlphaTestEnable** — alpha 测试启用
- **SRCBLEND** 和 **DSTBLEND** — 即混合操作
- **AlphablendEnable** — 帧缓冲中的应用程序混合
- **Fogenable** — fog 启用
- **Stencil enable** — 模板缓冲写入被启用
- **StencilTes**t ……

在使用光栅操作状态检测器时，您应该：

- 当混合不正常时确认后台缓冲格式中确实包含 alpha 成分
- 确认在绘制不透明对象时未启用 **blendEnable**

Chapter 6.
第 6 章
分析性能瓶颈

6.1. 图形管线性能

在过去几年中，采用硬件加速的着色管线越来越复杂，导致性能特征也越来越复杂并可能引起混淆。在过去，降低着色器中内环的 CPU 周期以提高性能是比较简单的事情，但现在却需要反复查找瓶颈并进行系统的优化。这种反复查找和优化是调整异类多处理器系统的基础，其根据是，按照管线的定义，它的最高速度取决于最慢一级的速度。合乎逻辑的结论就是，虽然对单处理器系统进行早期无针对性的优化只能获得最小限度的性能提升，但是对多处理器系统进行同样的优化却往往没有任何效果。

努力进行图形优化，结果却看不到任何性能改善，这不是件让人愉快的事情。本章的目的是解释如何用 NVPerfHUD 找出性能瓶颈并避免浪费时间。

6.1.1. 管线概述

管线在最高一级分为两部分：CPU 和图形芯片。图 16 显示了在图形芯片中有许多并行操作的功能单元，它们从本质上可以被看作独立的用于特殊目的的处理器；还有许多点，瓶颈可能在这些点上产生。这包括顶点及索引获取、顶点着色（转换及光照）、象素着色、纹理载入、光栅操作（ROP）。

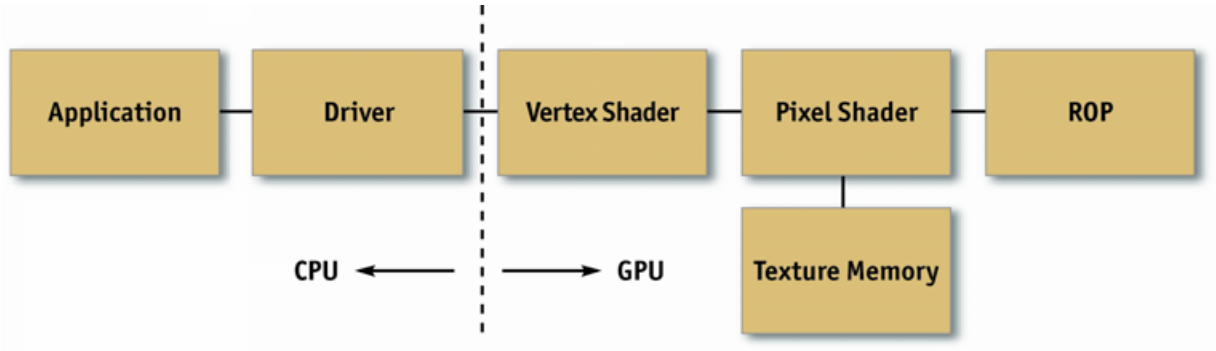


图 16. 管线概述

6.1.2. 方法

在未准确的找出瓶颈以前就进行优化，这种做法浪费了许多开发工作。因此我们将优化过程正式化，形成了以下基本的查找及优化环。

- **查找**
  通过 NVPerfHUD 测试分别隔离管线中的每一级。如果性能发生变化，则说明您已经找到了一个瓶颈。您还可以自己试着进行类似的测试，方法是修改应用程序以改变它的工作量。
- **优化**
  确定了瓶颈级以后，减少它的工作量，直到性能不再改善或者达到所需的性能水平。
- **重复**
  重复步骤 1 和 2，直到性能水平达到要求。

6.2. 确定瓶颈

查找瓶颈是进行优化的一部分，因为它让您在实际进行优化时能够做出明智的决定。图 17 是一个流程图，描述了查找应用程序瓶颈所需的一系列步骤。请注意，我们从管线的后端 — 伴随帧缓冲操作（也叫光栅操作），以及 CPU 的尾部开始。同时还请注意，虽然每一个单独的简单图形（通常为三角形）都各有瓶颈，但随着单个帧的前进这些瓶颈很可能发生改变，因此改变管线中多个级的工作量常常会对性能造成影响。例如，低多边形 skybox 通常会受到象素着色或帧缓冲的限制，但一个只映射到屏幕上少数几个象素的 skinned mesh 却受到 CPU 或顶点处理的限制。因此，以逐个对象或逐个材质的方式改变工作量常常会有所帮助。

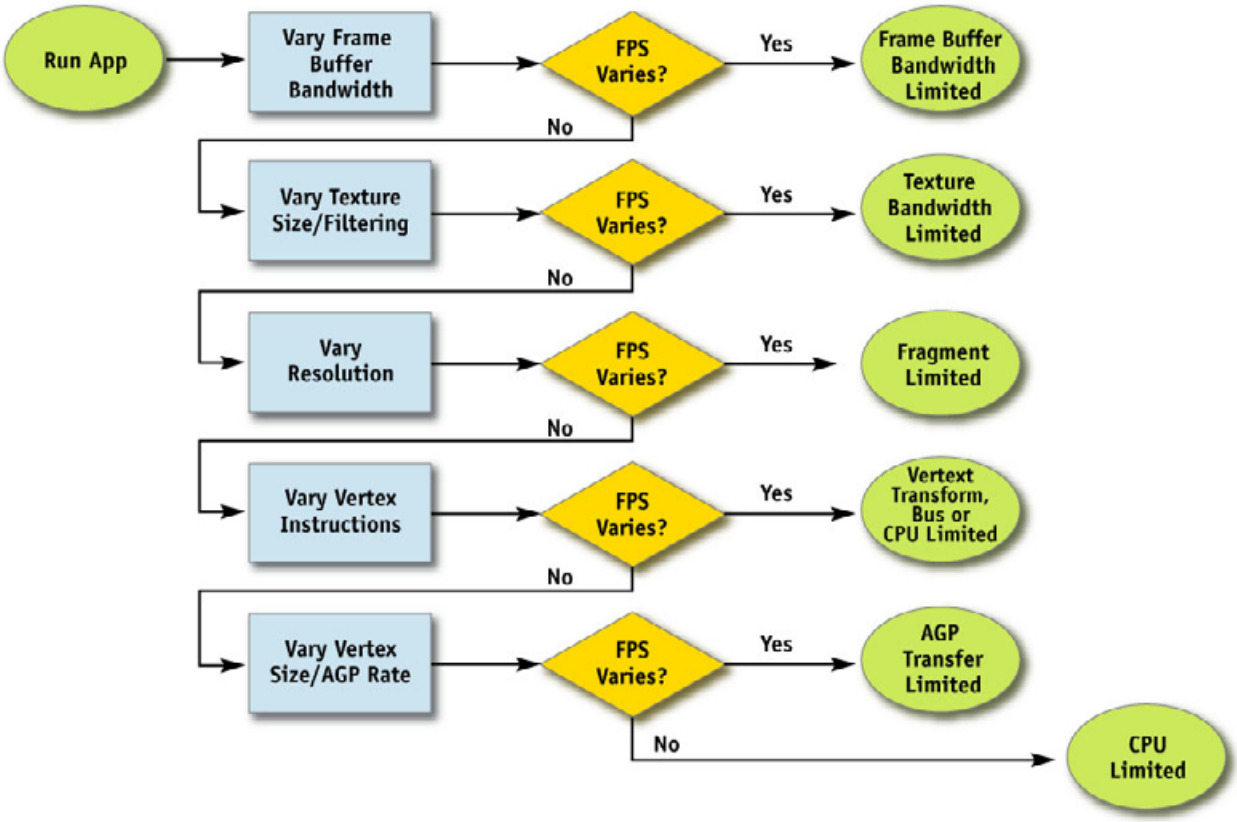


图 17. 确定瓶颈

注意： 如果您怀疑应用程序受到了 CPU 的限制，您可以随时按 N 键。如果应用程序的帧速率没有显著提高，则说明应用程序的确受到了 CPU 的限制。

6.2.1. 光栅操作瓶颈

管线的后端常常被称作 ROP，它负责读写深度及模板、进行深度及模板的比较、读写颜色、还进行 alpha 混合及测试。正如您所看到的，许多 ROP 工作量占用了大量可用的帧缓冲带宽。

要想测试应用程序是否受到了帧缓冲带宽的限制，最好的办法就是更改颜色的位深度和/或深度缓冲。如果将位深度从 32 位降至 16 位能够显著改善性能，那毫无疑问，您的应用程序受到了帧缓冲带宽的限制。

帧缓冲带宽的大小取决于图形芯片存储器的时钟速度。

6.2.2. 纹理带宽瓶颈

每当获取纹理的请求被发送到存储器时，就会占用纹理带宽。虽然现代图形芯片采用纹理高速缓存来尽量减少外部存储器请求，但这种请求仍然会产生并占用大量存储器带宽。

NVPerfHUD 激活以后按 T 键将应用程序中的所有纹理替换为 2x2 纹理。这将大大提高获取纹理的速度，并且纹理高速缓存的连贯性也更好。如果这使性能显著提高，则说明您的应用程序性能受到了纹理带宽的限制。

纹理带宽的大小同样取决于图形芯片存储器的时钟速度。

6.2.3. 象素着色瓶颈

象素着色是指生成一个象素时实际占用的资源，它与颜色及深度的值有关。这也就是运行“象素着色器”所占用的资源。请注意，由于象素着色与帧缓冲带宽都是屏幕解析度的功能，因此常常统称为“填充率”，实际上它们是管线中两个完全不同的级。要想有效的确定瓶颈并进行优化，正确区分象素着色和帧缓冲带宽是必不可少的。

要想确定象素着色是否是瓶颈，首先要用 NVPerfHUD 将所有着色器替换为非常简单的着色器。替换的办法是在性能分析模式中用 1、2、3……逐个禁用象素着色器配置文件，然后观察帧速率的改变。如果这能让性能显著提高，则说明象素着色非常可能是瓶颈所在。

接下来使用 NVShaderPerf 或者 FX Composer 中的 Shader Perf 面板找出占用资源最多的着色器。请记住，由于象素着色器所占用的资源是逐象素的，因此同样是占用资源较多的着色器，影响较多象素的那个比影响较少象素的那个更易造成性能问题。基本上，着色器所占资源 = 每个象素所占资源 * 被影响的象素数量。 主要的性能优化工作应该放在占用资源最多的着色器上。

FX Composer 含有许多着色器优化教程，它们对于降低着色器的性能资源占用将有所帮助。

注意：最新版本的FX Composer及NVShaderPerf将帮助您分析所有NVIDIA图形芯片的着色器性能。它们都可以从http://developer.nvidia.com下载。

6.2.4 顶点处理瓶颈

着色管线的顶点转换级负责取得一组顶点属性（例如模型空间位置、顶点法线、纹理坐标等等）并产生一组适合剪裁及光栅化（例如齐次剪裁空间位置、顶点光照结果、纹理坐标等等）的属性。因此，该级的性能取决于逐顶点完成的工作以及被处理的顶点数量。

确定顶点处理是否是瓶颈的办法很简单，同时运行 NVPerfHUD 和您的应用程序并按 V 键将顶点单元隔离。如果随后的帧速率大致等于原来的帧速率，那么您的应用程序就是受到了顶点/索引缓冲 AGP 传输、顶点着色器单元的限制，或者是由于锁定效率不高导致图形芯片出现延迟。

注意： 要排除锁定效率不高这一因素，您应该在 Direct3D 调试运行时中运行应用程序并确定没有错误或警告产生。

6.2.5. 顶点及索引传输瓶颈

在管线的图形芯片部分中顶点及索引是作为第一步被图形芯片获取的。获取顶点及索引的性能取决于顶点及索引的实际存放位置。存放位置一般为本地帧缓冲存储器或者系统存储器，在后一种情况下顶点及索引必须通过一种总线（例如 AGP 或 PCI-Express）才能传送到图形芯片。一般来说（特别是在个人电脑平台上），做出这种决定的是设备驱动程序而不是应用程序，虽然现代图形 API 允许应用程序向驱动程序提供使用提示，帮助驱动程序选择正确的存储器类型。请参考《NVIDIA 图形芯片编程指南》[链接]，其中介绍了在应用程序中使用索引缓冲及顶点缓冲的最佳方法。

要想确定顶点或象素获取是否是应用程序的瓶颈，方法是修改顶点格式的大小。

如果顶点及象素数据存放在系统存储器中，则获取数据的性能取决于 AGP/PCI-Express 总线的传输速度；如果数据存放在本地帧缓冲存储器中，则该性能取决于存储器时钟速度。

6.2.6. CPU 瓶颈

要想了解应用程序是否受到了 CPU 的限制，方法有两种。

较为简便的方法是观察 NVPerfHUD 时间图表中的图形芯片空闲线（绿色）。如果绿线保持平直且贴近图表的底部，则说明图形芯片从未空闲。如果绿线从底部上升，则说明 CPU 未向图形芯片提供足够的工作量。

另一种方法是按 N 键将图形芯片隔离。当您这样做时，NVPerfHUD 将强制 Direct3D 运行时忽略所有 DP 调用。随后的帧速率将略等于当图形芯片及显示驱动程序的速度无限高时应用程序的帧速率。

如果应用程序性能受到了 CPU 的限制（CPU 没有时间向图形芯片发送工作），其原因可能是：

- 太多 DP 调用 — 每一次调用都会导致驱动程序占用资源。请参看图 18 及其解释。
- 应用逻辑、physics 等的要求太高 — FRAME_TIME（黄色）线及 TIME_IN_DRIVER（红色）线中间的时差表示应用程序占用的 CPU 时间。
- 载入或分配资源 — 例如，驱动程序必须处理每个纹理，因此当驱动程序忙着载入许多（大的）纹理时，图形芯片可以完成所有等待处理的工作。请在资源创建监视器中观察这些事件，3.1.2 节对此进行了介绍。

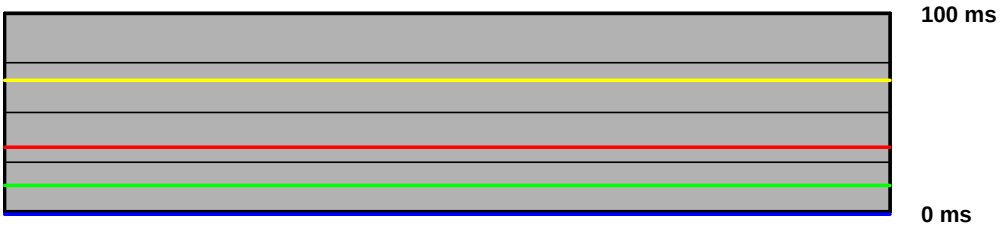


图 18. 对驱动程序的调用太多

图 18 显示了一个典型例子，其中应用程序对驱动程序的调用太多。请参看相同情形下报告批处理数量的图表。

6.3. 优化

现在我们已经找出了瓶颈，我们必须优化特定的级以提高应用程序性能。以下优化建议按照它们所优化的级进行组织。

6.3.1. CPU 优化

应用程序的性能可能因为 physics 或 AI 太复杂而受到 CPU 限制。此外还可能因为批处理大小或资源管理不当而使性能受到影响。如果您发现应用程序受到了 CPU 的限制，请按照下列建议减少着色管线中 CPU 的工作量。

6.3.2. 减少资源锁定

资源既可以是纹理缓冲，也可以是顶点缓冲。每当您执行一个需要使用图形芯片资源的同步操作时，该操作可能使图形芯片管线出现严重延迟，这种延迟又会占用 CPU 和图形芯片的工作周期。由于 CPU 必须停止工作，等待图形芯片管线返回所请求的资源，因此 CPU 的工作周期被浪费掉了。接着管线也停止工作等待重新填充，因此图形芯片的工作周期也被浪费掉了。

当您进行下列操作时可能发生上述情况：

- 锁定以前着色的表面或读取该表面。
- 写入图形芯片正在读取的表面，例如一个纹理或顶点缓冲。

锁定正被使用的资源会导致 DRIVER_WAITS_FOR_GPU（蓝色）线上升。请参看附录 A 中关于驱动程序等待图形芯片的原因的介绍。

为了排除低效率锁定因素，您应该在Direct3D调试运行时中运行应用程序并确认没有错误或警告产生。要想进一步了解如何有效管理资源锁定，请阅读白皮书：
http://developer.nvidia.com/object/dynamic_vb_ib.html

6.3.3 尽量减少绘图调用的数量

每一次绘制几何图形的 API 函数调用都会引起相关的 CPU 资源占用，因此尽量减少 API 调用的次数，特别是尽量减少图形状态改变的数量，能够将用于特定数量的三角形着色的 CPU 工作降至最低。

我们对批处理的定义是：通过单次 API 着色调用进行着色的一组简单图形，例如 DirectX 9 中的 DrawPrimitive（）和 DrawIndexedPrimitive（）。批处理的“大小”是指它包含的简单图形的数量。

通过 NVPerfHUD 您能够了解批处理的工作情况如何。按 B 键将显示一个柱状图，图中显示了每一帧每次绘图调用的三角形数量分布。图 19 显示了一个可能表现不佳的应用程序，原因是它包含了太多只对少量简单图形进行着色的 DP 调用。

图 19. 太多小规模 DP 调用

要减少 DP 调用的数量，请试试以下方法：

- **如果使用了三角形条带，请用退化三角形将分离的条带连在一起。**这样您就能够在单次绘图调用中发送多个条带 — 假如它们共享材质。NVTristrip库可以从http://developer.nvidia.com下载，它提供了相关源代码。
- **使用纹理页面。**当不同的对象使用不同纹理时，批处理常常会被打断。将许多纹理安排在单个 2D 纹理中并恰当的设定纹理坐标，您就可以将使用多个纹理的几何图形发送到单次绘图调用中。请注意，这种技巧可能会涉及 mipmapping 及反锯齿问题。避免这类问题的方法之一是将单个 2D 纹理放在立体纹理的每一面中。最新版本的 NVIDIA SDK 包括了许多纹理地图工具，这些工具能帮助您创建并预览纹理页面。
- **将顶点着色器常量存储器用做矩阵查找表。**当许多小对象共享所有材质属性却只在矩阵状态（例如由相似的树组成的森林）上有所不同时，批处理常常会被打断。在这些情况下，您可以将多个矩阵载入顶点着色器常量存储器并按照每个对象的顶点格式将索引存入常量存储器。接着您在顶点着色器中使用该索引在常量存储器中进行查找并使用正确的转换矩阵，这样就可以一次性对 N 个对象进行着色。
- **如果场景中存在一个网格的多份拷贝，请使用几何实例。**该技巧允许您通过单次绘图调用及两个顶点流绘制同一个网格对象的多个拷贝。该网格的每一份拷贝或“实例”可以被绘制在不同的位置，并且（可选）具有不同的视觉效果。其中一个顶点流容纳将被实例化的网格的单个拷贝，另一个顶点流则容纳逐实例数据（世界转换、颜色等等）。然后您可以调用一个绘图调用并告诉它要绘制的实例数量。一般来说，当低（低于 1000）多边形目标较多时几何实例最为有用，因为它可以减少大量绘图调用对 CPU 资源的占用。最新版本的 NVIDIA SDK 中也包含了一个几何实例的例子（附带完整的源代码）。

- **使用图形芯片着色器分支提高批处理的大小。**现代图形芯片拥有灵活的顶点及象素处理管线，这种管线允许着色器内部的分支。例如，如果有两个批处理，一个需要 4 bone skinning 顶点着色器，另一个需要 2 bone skinning 顶点着色器，那么这两个批处理就是分开的。但您可以写一个顶点着色器，该着色器循环所需的 bone 数目并累计混合权重，当权重合计为 1 时停止循环。如此一来，这两个批处理就能合而为一。在不支持着色器分支的结构上也能执行类似的功能，只是要占用着色器的工作周期。其方法是全部使用 4 bone 顶点着色器并简单的 zero out 那些 bone influences 少于 4 个的顶点上的 bone weight。
- **尽量推迟管线中的选择。**与其打破批处理为光泽度设定象素着色器常量，不如将纹理的 alpha 通道用做光泽因子，这样可以提高速度。同样，将着色数据放入纹理或顶点能够提交较大的批处理。

6.3.4. 降低顶点传输所占用的资源

现在的应用程序中顶点传输很少会成为瓶颈，但并不是说这种问题就不会发生。如果顶点传输或者索引传输（更少成为瓶颈）成为应用程序的瓶颈，请试试以下方法：

- **在顶点格式中使用尽可能少的字节数。** 如果字节够用的话就不要使用浮点格式（例如在颜色中）。
- 在顶点程序内生成可能可推导的顶点属性，而不要把它们存入输入顶点格式中。例如，通常没有必要储存切线、次法线或法线，因为只要给定了任意两个，第三个就能在顶点程序内使用简单的叉积推导出来。这是一种用顶点传输率换取顶点处理速度的技巧。
- **使用 16 位索引替换 32 位索引。**16 位索引的提取及传输占用的资源及存储器都比较少。
- **以较为连续的方式访问顶点数据。**现代图形芯片在提取顶点数据时会将从存储器中获得的数据放入高速缓存中。同任何存储器结构一样，相关数据的空间位置有助于最大程度提高从高速缓存中获取数据的命中率，从而降低所需的带宽。

6.3.5. 优化顶点处理

在现代图形芯片中顶点处理很少成为瓶颈，但这种问题并非不会发生，这取决于您的使用方式及目标硬件。如果您发现顶点处理是应用程序的瓶颈，请试试以下方法：

- **将逐对象计算交给 CPU 完成。**通常情况下，为了方便，随着每个对象或每帧发生改变的运算都是在顶点着色器中完成的。例如，对于视野空间的定向光照矢量转换就是在顶点着色器中完成的，虽然其计算结果只随着每一帧而改变。
- **对转换及光照（TnL）后的顶点高速缓存进行优化。**现代图形芯片拥有一个小的先入先出（FIFO）高速缓存，用于储存最近转换过的顶点结

果；如果能从高速缓存中获得结果，就不用再次进行转换或光照，而管线中原来已经完成的工作也不必再次重复。要利用该高速缓存，您必须使用编入索引的简单图形，同时必须命令顶点在网格上实现相关数据位置的最大化。有一些免费工具能帮助您完成这项工作，包括D3DX及NVTriStrip — http://developer.nvidia.com/object/nvtristrip_library.html。

- **减少被处理的顶点数量。**这很少成为主要问题，但使用简单的细节层次（LOD）方案 — 例如一组静态 LOD，可以毫无疑问的减少顶点处理的工作量。
- **使用顶点处理 LOD。**除了为许多被处理的顶点使用 LOD 以外，试试用 LOD 处理实际的顶点运算本身。例如，您很可能不需要对远处的角色进行完整的 4 bone skinning，同时有可能用较少的资源占用获得大致相当的光照。如果您的材质需要多次处理，那么减少处理次数、降低远处的细节层次将减少顶点处理所占用的资源。
- **使用正确的坐标空间。**很多时候您对坐标空间的选择会对计算顶点程序中某个值所需的指令数目造成影响。例如，在进行顶点光照时，如果您的顶点法线存储在对象空间而光照矢量储存在视野空间，那么您必须在顶点着色器中转换其中一个矢量。如果光照矢量在 CPU 上按照逐对象的方式被转换进对象空间，那就不再需要逐顶点转换，从而节省了图形芯片的顶点指令。
- **使用顶点分支提前结束（early-out）计算。** 如果对顶点着色器中的许多光照进行循环并进行常规低动态范围[0..1]光照，您可以检查饱和度为 1，又或者您正避开光照，则可以跳出进一步的运算。Skinning 也可以进行类似的优化，当权重合计为 1 时您可以终止运算（所有后继权重都将为零）。请注意，这取决于图形芯片执行顶点分支的方式，而且并不是在所有结构上都一定能获得性能提升。

6.3.6. 提高象素着色速度

如果您正使用又长又复杂的象素着色器，那很可能象素着色限制了应用程序的性能。如果您发现正是如此，请试试下面的方法：

- **首先对深度着色。**在对简单图形进行着色处理以前先只对深度（无颜色）进行着色处理可以极大的提高性能，特别是在深度复杂度较高的场景中更是如此。其原因是减少了对象素着色及帧缓冲访问的需要。要完全实现只对深度进行处理的好处，仅仅禁止对帧缓冲进行颜色写入还不够，您还应该禁止对象素进行一切着色，包括会对深度及颜色造成影响的着色（例如 alpha 测试）。
- **帮助 early-Z 优化丢弃象素处理。**现代图形芯片有防止对看不见的象素进行着色的设计，但是该功能依赖于对当前点之前的场景的认识，而且粗略按照从前到后的顺序进行着色将大大提高该功能的效果。此外，先在独立的处理过程中对深度进行处理（参看上文）将显著提高后续处理（占用资源最多的着色即通过这些处理来完成）的速度，因为这样做有效的将已着色的深度复杂性降到了 1。

- **将复杂的函数存储在纹理中。**将纹理用做查找表非常有用，这样做还能在不占用任何资源的情况下对结果进行过滤。最典型的例子使一个正常化立体纹理，它允许您对任意矢量进行高精度的正常化，而且只需进行一次纹理查找。
- **将逐象素工作交给象素着色器完成。**正如顶点着色器中的逐对象工作应该交给 CPU 完成一样，逐顶点计算（以及能够在屏幕空间中被正确的用内插值进行线性替换的计算）应该交给顶点着色器完成。常见的例子包括在矢量计算以及在坐标系统之间进行矢量转换。
- **使用尽可能低的精度。**DirectX 9 等 API 允许您在象素着色器代码中为那些能够以更低精度工作的量或计算指定精度信息。许多图形芯片可以利用这些信息降低内部精度以提高性能。
- **避免不必要的正常化。**一个常见的错误是过多使用正常化，在计算的每一步都对每一个矢量进行正常化。请区分哪些转换保存长度（例如标准正交转换），哪些计算不依赖矢量长度（例如立体纹理查找）。
- **可能时使用半精度正常化。**在 NV4x 系列图形芯片上进行半精度正常化基本上不需占用任何资源。请在高级着色语言（HLSL）中对将被正常化的矢量使用半精度类型。如果在 DirectX 9 中使用 ps_2_0 或更高版本的 assembly 着色器，请使用 **nrm_pp**（或在同等运算的所有操作中使用‘_pp’修饰符）。在测试您的 HLSL 着色器时，最好检查一下产生的 assembly，看看正常化所对应的操作中是否使用了_pp 修饰符，以确保您正确使用了半精度数据类型。您可以运行 fxc.exe 或 FX Composer 的 Shader Perf 面板，对 HLSL 着色器生成的 assembly 进行检验。
- **考虑使用象素着色器细节层次。**虽然其效果不如顶点 LOD 那样明显（因为远处对象会因为远景的原因自然而然的对自身进行与象素处理有关的 LOD），但是通过降低远处着色器的复杂性并对表面的处理次数，仍然可以减少象素处理的工作量。
- **确定应用程序没有受到纹理带宽的限制。**详情请参看下一节“降低纹理带宽”。

6.3.7. 降低纹理带宽

如果您发现应用程序受到了存储器带宽的限制，但这种限制大多发生在从纹理获取数据的时候，请考虑以下优化措施：

- **减小纹理的大小。**考虑一下目标解析度及纹理坐标。该程序的用户会看到最高的 miplevel 吗？如果不会，请考虑缩小纹理的尺寸。当帧缓冲存储器由于过载不得不将纹理存储到非本地存储器（例如通过 AGP 或 PCI-Express 总线存储到系统存储器）时，这种措施将非常有效。NVPerfHUD 的存储器图表有助于诊断这种问题，因为它显示了驱动程序在各种堆中分配的存储器大小。
- **在可以缩小的表面上坚持使用MIP mapping。**MIP mapping可以通过减少纹理锯齿来提高图像质量。有各种过滤器可以用来创建高质量、不模糊的MIP map。NVIDIA提供了一套能帮您对MIP map创建进行优化的纹理工具，包括一个Photoshop插件、一个命令行工具以及一个库。其下载地址为http://developer.nvidia.com/object/nv_texture_tools.html。如果没有MIP mapping，您就只能对纹理进行点采样，这可能导致不必要的微光效果。

> 注意：如果您发现 mipmapping 使某些表面变得模糊，请不要禁用 mipmapping 或加入大的 LOD 负偏移。您应该使用各向异性过滤。

- **压缩所有有色纹理。**所有纹理（例如印花纹理或精细纹理）都应该被压缩。压缩时可采用 DXT1、DXT3 或 DXT5，这取决于纹理的 alpha 要求。这可以降低存储器使用率以及对纹理带宽的要求，并能提高纹理高速缓存的效率。
- **若非必要，避免占用资源较多的纹理格式。**大的纹理格式（例如 64 位或 128 位浮点格式）在纹理提取时明显会占用更多带宽。因此请按照需要选用格式。
- **使用适当的各向异性纹理过滤级别。**如果使用低频率纹理和高级别的各向异性过滤，图形芯片的工作量会增加，同时图像质量也得不到提高。如果您受到了纹理带宽的限制，那么使用低级别的各向异性过滤能够将图像质量提高到满意的水平。理想的应用程序应该能够根据不同的纹理设定各向异性的级别。
- **若非必要，禁用三线过滤。**在现代图形芯片结构上，三线过滤即使不占用额外的纹理带宽，也会在象素着色器中占用额外的计算周期。对于 miplevel 转换不易识别的纹理上应该关闭三线过滤，以节约填充率。

6.3.8. 优化帧缓冲带宽

ROP 是管线的最后一级，它与帧缓冲存储器直接进行数据交换，占用的帧缓冲带宽也最多。因此，如果带宽限制了应用程序的性能，其原因往往都可追溯到 ROP。一下是优化帧缓冲带宽的方法：

- **首先对深度着色。**这不仅可以减少象素着色占用的资源（见上文），还能减少对帧缓冲带宽的占用。
- **减少 alpha 混合。**alpha 混合需要对帧缓冲进行读和写，因此可能占用双倍的带宽。根据需要尽可能减少 alpha 混合，并当心高级别 alpha 混合的深度复杂性。
- **可能的话关闭深度写入。**深度写入会占用额外的带宽，因此应该在多次着色中禁用它（在多次着色时最后的深度已在深度缓冲中）。在渲染 alpha 混合的效果（例如微粒效果）以及将对象渲染到阴影贴图（实际上是渲染到基于颜色的阴影贴图）的时候，您同样可以关闭深入写入。
- **避免无关的颜色缓冲清除。**如果您确定应用程序将覆盖写入帧缓冲中每一个象素，那就应该避免进行颜色清除，因为这会占用宝贵的带宽。另外还需注意，您应该尽可能的清除深度及模板缓冲，因为许多 early-Z 优化都依赖于清除过的深度缓冲中的确定性内容。
- **由前向后着色。**由前向后着色除了能带来我们上面提到的象素着色改善以外，还能为帧缓冲带宽带来类似的好处，这是因为 early-Z 硬件优化能够丢弃无关的帧缓冲读写。事实上，即使是不具有这些优化功能的旧硬件也能从中得到益处，这是因为更多的象素会导致深度测试失败，从而使写入帧缓冲的颜色及深度减少。
- **优化 skybox 着色。**Skybox 常常受到帧缓冲带宽的限制，但对 skybox 进行优化的方法却不止一种。您可以把它们放在最后着色、读取（但不要写入）深度，并允许 early-Z 优化和普通深度缓冲，以节省带宽。您也可以首先对 skybox 进行着色，并禁用所有深度读取或写入。至于这两种方法哪种更节约带宽，这取决于目标硬件以及最后一帧中有多少可见的 skybox。如果大部分 skybox 都不可见，那么第一种方法更好，否则第二种方法能节省更带宽。
- **只在必要时使用浮点帧缓冲。**浮点帧缓冲明显比小的整数格式占用更多的带宽。这同样适用于多着色器目标。
- **可能的话，使用 16 位深度缓冲。**深度处理占用大量的带宽，因此用 16 位代替 32 位可以显著提高性能，而且 16 位深度缓冲对于无需模板的小型室内场景也足够了。对于需要深度的着色器至纹理效果（例如动态立体纹理）来说，16 位深度缓冲也够用了。
- **可能的话，使用 16 位颜色。**这个方法特别适用于着色器至纹理效果，因为很多这类效果（例如动态立体贴图或彩色投射阴影贴图）在 16 位颜色下已经工作的很好了。

Chapter 7.
第 7 章
故障排除

欢迎您在我们的开发者论坛中发表问题或意见，或者通过电子邮件发送到 NVPerfHUD@nvidia.com，信件内容我们将予以保密。

7.1. 已知问题

- NVPerfHUD 无法处理多设备。它只支持由 **Direct3Dcreate9（）**创建的第一个设备。
- 当使用软件顶点处理时，NVPerfHUD 可能崩溃。
- 在本地使用 **rdtsc** 的应用程序无法在 NVPerfHUD 的帧分析模式下正常工作 — 可能的解决办法请参阅下面的“故障排除”。
- 帧分析模式需要您的应用程序使用并依赖 **`QueryPerformanceCounter()`**或 **`timeGetTime()win32`** 函数。您的应用程序必须正确无误的处理已用时间（dt）的计算，特别是当 dt 为零时。换句话说，您的程序不应该除以 dt。如果您怀疑存在这个问题，请在 NVPerfHUD 配置对话框中将 Delta Time 选项设为“SlowMo”。如果这能解决问题，请考虑对应用程序进行修改，使它能够处理 dt 为 0 的情况。
- 索引单元状态检测器不支持对点和线的显示。
- 当应用程序使用固定的转换和光照（T&L）时，状态检测器不显示详细信息。
- 如果您在帧分析模式下使 NVPerfHUD 失活（通过激活热键）然后退出您的应用程序，您可能会看到一个警告对话框，其内容为“Number of references when exiting was not 0”（退出时的引用不为 0）。
- 当 NVPerfHUD 处于激活状态时如果点击关闭按钮，在窗口模式下运行的应用程序可能无法正常退出。

7.2. 常见问题

NVPerfHUD 提示我的应用程序不能用 NVPerfHUD 进行分析。

为了防止未经授权的第三方不经您同意就对您的应用程序进行分析，您必须稍做修改才能启用 NVPerfHUD 分析。修改办法请参考本用户指南的“开始”一节。

启动 NVPerfHUD 以后我的应用程序没有响应。

在使用热键启用 NVPerfHUD 以后，它将接收所有键盘输入并且不会将任何键盘输入传递给应用程序。您可以用自己选定的激活热键将该模式打开/关闭。

我能在屏幕顶部看见 NVPerfHUD 的标题，但是它不响应我的激活热键。

NVPerfHUD 使用几种方法来侦听击键事件。如果您使用了尚未支持的方法，请告诉我们，我们将进行修改。

请注意，在 Win2000 中 NVPerfHUD 使用 DirectInput 来侦听激活热键并在激活后侦听键盘命令。DirectInput 提供两种类型的数据：缓冲数据及直接数据。缓冲数据是存储起来的事件记录，供应用程序提取。直接数据则是对某个设备的当前状态的记录。

这意味着如果您想使用 NVPerfHUD 的高级功能（例如瓶颈查找测试、着色器视图等），您的应用程序需要使用 IDirectInputDevice8::GetDeviceData 接口，而不是 IDirectInputDevice8::GetDeviceState 接口。

NVPerfHUD 弄乱了 alpha 及一些着色状态。

NVPerfHUD 在帧尾对 HUD 进行着色。它还会为了绘制自己而更改着色状态，却并不将着色状态还原。换句话说，处于性能原因，它不会压入和弹出着色状态 — 因此，它假设您的应用程序会在每一帧开始时重置着色状态。

GPU_IDLE（绿色）线不报告任何数据。

对于 GeForce4（NV25）及其以前的图形芯片，该消息不可用。在较新的图形芯片上，如果 NVPerfHUD 不能同显示驱动程序正常通讯，该消息也不可用。请确认您使用的最新版本的 NVPerfHUD 以及 NVIDIA 显示驱动程序。

性能图表中的彩线不显示或全部为零，但 NVPerfHUD 的标题及批处理柱状图却正常。

当使用 DirectX 9 调试运行时的时候这些功能无法使用。请通过 DirectX 控制面板将调试运行时切换回 RETAIL 运行时，以进行准确的性能分析。如果您只使用 NVPerfHUD 的调试控制台模式查找应用程序的功能问题，则可以使用调试运行时。

我在 NVPerfHUD 的图表中央看见一些额外的线条。

如果您正在运行较老的驱动程序（特别是在 Win2000 上），您可能在 NVPerfHUD 图表的上面看见部分重叠的老版本 NVPerfHUD 1.0 的图表。将驱动程序升级到 71.8x 或更新的版本应该可以解决这一问题。

场景中的一些对象会在帧分析模式下不停变化

当切换到帧分析模式以后，NVPerfHUD 将停止应用程序的时钟，以便您能在当前帧定格时对它进行深入分析。您的应用程序必须使用并依赖 QueryPerformanceCounter() or timeGetTime() win32 函数。如果您的应用程序使用了 **rdtsc** 指令，它将无法在帧分析模式下正常工作。

如果您的应用程序使用了基于帧的动画，时间冻结将动画对象无效。

我发现了一个上面未列出的问题

我们希望 NVPerfHUD 能不断完善，成为开发者用来分析应用程序的有用工具。如果您遇到了任何问题或者对于想到了有助于 NVPerfHUD 使用的其他功能，请告诉我们。

NVPerfHUD@nvidia.com

Appendix A.
附录 A
为什么驱动程序等待图形芯片

驱动程序等待图形芯片的典型情形是图形芯片被场景充分利用，它出现在这种情况下：您通过 FIFO 将两个处理器连接起来，其中一个芯片向另一个芯片传送超出其处理能力的数据。

这种情况如下图所示，CPU 向图形芯片发送的指令超出了图形芯片的处理能力。这时所有指令都开始在 FIFO 队列中阻塞，这种情况也被成为“push buffer”。为了防止 FIFO 发生溢出，驱动程序被强迫进行等待，直到 FIFO 中有空间容纳新的指令。

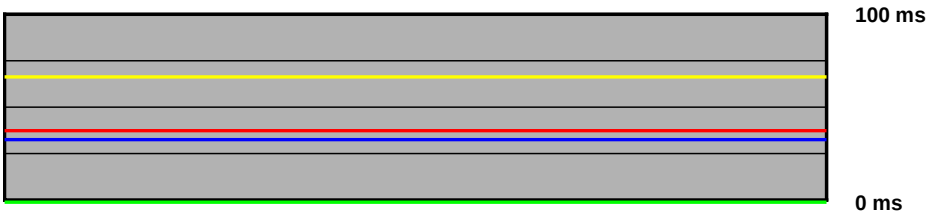


图 20. 驱动程序等待图形芯片

如果您发现帧速率：

- **高**，那么您可以在 CPU 上进行更多工作，这不会影响帧速率（对象选择、physics、游戏逻辑、人工只能等等）。
- **不足**，您应该降低场景复杂性，以减轻图形芯片的工作量。

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