
NVIDIA
GeFORCE
6 Series
NVIDIA GPU
Programming Guide
Version 2.2.0
NVIDIA 图形处理器编程指南
中文
版本 2.2.0
NVIDIA

注意

所有 NVIDIA 的设计规范、参考公告板、文件、制图、诊断手册、目录及其他文档（共同或分别简称为“资料”）至发布之日均为准确资料。NVIDIA 对这些资料作任何明示的、默示的、法定的或其他方式的保证，并明确否认一切默示的不侵权保证、适销性保证和特殊用途适合性保证。

NVIDIA 公司确信所提供的信息具有准确性和可靠性。但 NVIDIA 公司不对因使用这类信息所造成的后果负责，也不对因使用这类信息对第三方专利或其他权益造成的侵害负责。对于 NVIDIA 公司的任何专利或专利权的使用，NVIDIA 公司概不承认任何默示的、或其他方式的许可。本指南中提到的规范如有变更，恕不通知。NVIDIA 公司的产品未经 NVIDA 公司书面授权不得用作生命支持设备或系统中的关键部件。

商标

NVIDIA、NVIDIA 标志、GeForce 和 NVIDIA Quadro 是 NVIDIA 公司的注册商标。其他公司或产品名称可能分别是有关公司各自的商标。

版权

主要修订记录

版本	日期	更改
2.2.0	2004 年 11 月 16 日	增加了法线贴图格式参考说明 增加了 ps_3_0 性能表现的参考说明 增加了“常规参考说明”一章
2.1.0	2004 年 7 月 20 日	增加了“立体开发”一章
2.0.4	2004 年 7 月 15 日	更新了“多对象渲染（MRT）”部分
2.0.3	2004 年 6 月 25 日	增加了“多图形芯片支持”一章
2.0.0	2004 年 6 月 1 日	增加了“NV40（GeForce 6 系列）”的章节 更名为《NVIDIA 图形芯片编程指南》
1.0.0	2003 年 7 月 14 日	《GeForce FX 编程指南》

目录

第 1 章 关于本文档 8
1.1. 序言 8
1.2. 发送反馈意见 9
第 2 章 如何优化应用程序 10
2.1. 进行准确的测试 10
2.2. 发现瓶颈 11
2.2.1. 了解瓶颈 11
2.2.2. 基本测试 12
2.2.3. 使用NVPerfHUD软件 13
2.3. 瓶颈：CPU 13
2.4. 瓶颈：GPU 15
第 3 章 提升图形芯片性能的相关技巧 17
3.1. 技巧目录 17
3.2. 批处理 19
3.2.1. 减少批处理的使用 19
3.3. 顶点着色器 20
3.3.1. 使用索引原语调用 20
3.4. 着色器 20
3.4.1. 选择可以正常工作的最低版本的像素着色器 20
3.4.2. 使用ps_2_a配置文件（Profile）编译像素着色器 21
3.4.3. 选择可以正常工作的精度最低的数据类型 21
3.4.4. 使用代数计算来减少运算量 22
3.4.5. 不要把矢量值放入含有多个内插值的标量部件中 23
3.4.6. 不要编写过于通用的库函数 23
3.4.7. 不要计算标准化矢量的长度 24
3.4.8. 合并恒定常量（Uniform Constant）表达式 24
3.4.9. 不要将恒定参数（Uniform Parameter）用于在像素着色器生命周期中不发生改变的常量 25

3.4.10. 平衡顶点着色器和像素着色器 25
3.4.11. 如果受到像素着色器的限制，就把可线性化的计算交给顶点着色器完成 26
3.4.12. 使用标准库函数mul() 26
3.4.13. 用D3DTADDRESS_CLAMP (或 GL_CLAMP_TO_EDGE)代替saturate()以取得附属纹理坐标（Dependent Texture Coordinates）26
3.4.14. 首先使用低位内插值 26
3.5. 纹理贴图 27
3.5.1. 使用Mipmapping纹理映射 27
3.5.2. 慎用三线过滤和各向异性过滤 27
3.5.3. 用纹理查找代替复杂的函数 27
3.6. 性能表现 30
3.6.1. 倍速Z-Only和模板渲染 30
3.6.2. Early-Z优化 30
3.6.3. 先规定深度 31
3.6.4. 内存分配 31
3.7. 反锯齿技术 31
第 4 章 GeForce 6 系列编程技巧 33
4.1. 支持 3.0 着色器模型 33
4.1.1. 3.0 像素着色器 33
4.1.2. 3.0 顶点着色器 34
4.1.3. 动态分支功能 35
4.1.4. 代码维护更简捷 35
4.1.5. 实例功能 36
4.1.6. 小结 36
4.2. sRGB解码功能 36
4.3. 单阿尔法（Alpha）合成 37
4.4. 支持的纹理格式 37
4.5. 浮点纹理 38
4.5.1. 局限性 39

4.6. 多渲染目标（MRTs） 39
4.7. 顶点纹理贴图 41
4.8. 总的性能建议 41
4.9. 法线图 42
第 5 章 GeForce FX编程技巧 43
5.1. 顶点着色器 43
5.2. 像素着色器的长度 43
5.3. DirectX专用的像素着色器 44
5.4. OpenGL专用的像素着色器 44
5.5. 使用 16 位浮点数 45
5.6. 支持的纹理格式 46
5.7. 在DirectX中使用ps_2_x和ps_2_a 47
5.8. 使用浮点数渲染目标 47
5.9. 法线帖图 48
5.10. 新的芯片和架构 48
5.11. 小结 48
第 6 章 总则 51
6.1. 识别图形芯片 51
6.2. 硬件阴影贴图 52
第 7 章 NVIDIA SLI与多图形芯片的性能 57
7.1. 什么是SLI? 57
7.2. 避免CPU成为系统瓶颈 59
7.3. 默认关闭VSync 59
7.4. 延迟限制在最小 2 帧内 60
7.5. 在所有帧中完全更新所有渲染目标纹理 61
7.6. 清除渲染目标和帧缓存 61
7.7. 在D3DPOOL-MANAGED中分配顶点缓存 62
第 8 章 立体游戏开发 63

8.1. 为何关注立体？ 63
8.2. 立体工作原理 64
8.3. 影响立体效果的负面因素 64
8.3.1. 错误深度的渲染 64
8.3.2. 广告牌效果 65
8.3.3. 后期处理与屏幕空间效果 65
8.3.4. 在 3D场景中使用 2D渲染 65
8.3.5. 子视角渲染 65
8.3.6. 用复杂方块更新屏幕 66
8.3.7. 用大量的分隔来解决碰撞 66
8.3.8. 改变场景中不同对象的深度段 66
8.3.9. 不提供顶点深度数据 66
8.3.10. 在窗口模式下渲染 66
8.3.11. 阴影 66
8.3.12. 软件渲染 67
8.3.13. 手动写入渲染目标 67
8.3.14. 很暗或很高的对比度场景 67
8.3.15. 顶点间有小空隙的对象 67
8.4. 改善立体效果 67
8.4.1. 在立体中测试您的游戏 67
8.4.2. 获得“飞出屏幕”的效果 67
8.4.3. 使用细致的几何图形 68
8.4.4. 提供交替视角 68
8.4.5. 查找当前游戏中的问题 68
8.5. 立体应用编程接口 68
8.6. 更多信息 69
第 9 章 性能工具概览 71
9.1. NVPerfHUD 71
9.2. NVShaderPerf 72

9.3. FX合成器 72
9.4. NVIDIA Melody 73
9.5. 开发人员工具 问题与反馈 73

Chapter 1.第 1 章
关于本文档

1.1. 序言

本指南旨在帮助您通过应用程序、图形应用编程接口（API）和图形芯片（GPU）取得最佳图形效果。本指南中的内容将有助于您编写出更好的图形应用程序，如果您需要任何帮助或建议，请随时发送电子邮件至：devsupport@nvidia.com。

本文档的组织结构如下：

- 第 1 章（本章）简要介绍了本文档的内容。
- 第 2 章介绍了如何通过查找和处理常见瓶颈来优化您的应用程序。
- 第 3 章列出一些技巧，以帮助您对确定的瓶颈进行处理。这些技巧经过分类并按其重要程度进行排列，因此，您可以先作最重要的优化。
- 第 4 章针对NVIDIA® GeForce™ 6 系列和基于NV4X的Quadro FX图形芯片讲述了一些有用的编程技巧。这些技巧主要是针对功能方面的，但某些情况下也会涉及性能表现方面的问题。
- 第 5 章针对NVIDIA® GeForce™ FX和基于NV3X的Quadro FX图形芯片讲述了一些有用的编程技巧。这些技巧主要是功能方面的问题，但某些情况下也会涉及性能表现方面的问题。
- 第 6 章针对 NVIDIA 图形芯片提供了一些常规建议，涵盖了各类主题，例如性能表现、GPU 的识别等。
- 第 7 章介绍了 NVIDIA 的可扩展连接接口（SLI）技术，该技术允许您使用多个图形芯片大大提高性能表现。

- 第 8 章描述如何利用我们对立体游戏的支持。精心编写的立体游戏不仅生动，而且在视觉上比非立体游戏具有更强的立体效果。
- 第 9 章是 NVIDIA 性能工具的概述。
- 第 10 章同时用内部代号和正式产品名称列出了我们的图形芯片，以便参考。

1.2. 发送反馈意见

如果您对本文档有任何评论或建议，请发送至：devsupport@nvidia.com。

Chapter 2.第 2 章
如何优化应用程序

本章将讲述发现并解决图形软件的性能瓶颈的典型步骤。

2.1. 进行准确的测试

许多方便的工具能够在您测试性能的同时为您提供可靠的性能测试指标。例如，NVPerfHUD的黄线（详情请见NVPerfHUD文档）测试每帧的总毫秒数，并显示当前的帧速率。

有效的性能对比方法：

- **确保应用程序运行正常。**例如，当软件和微软的 DirectX Debug 同时运行时，不应产生任何错误或警告。
- **确保测试环境有效。**即确保您运行的是应用程序的发布版本与动态链接库（DLL），以及正式发布的最新版本 DirectX。
- **所有应用程序都要使用发布版本（而非调试版本）。**
- **确保所有显示设置正确无误。**即所有显示设置都应保持默认值。各向异性过滤和反锯齿设置会明显影响性能。
- **禁止垂直同步。**这能确保帧速率不被显示器刷新率所限制。
- **在目标硬件上运行。**如果您想知道一个明确的硬件设置是否能正常运行，请确保系统使用了恰当的 CPU、图形芯片（GPU）、并有足够的内存。当您从低端系统转移到高端系统时，瓶颈会显著改变。

2.2. 发现瓶颈

2.2.1. 了解瓶颈

在此，假设我们已确定了一个性能较差的情况，此时我们需要做的是找出性能瓶颈。瓶颈通常随着场景内容而转移。瓶颈还经常在单帧的过程中转移，从而使问题变得更为复杂。因此，“发现瓶颈”意味着“找出这个场景中对性能限制最大的瓶颈”。*消除这种瓶颈就能获得最大的性能优化空间。*

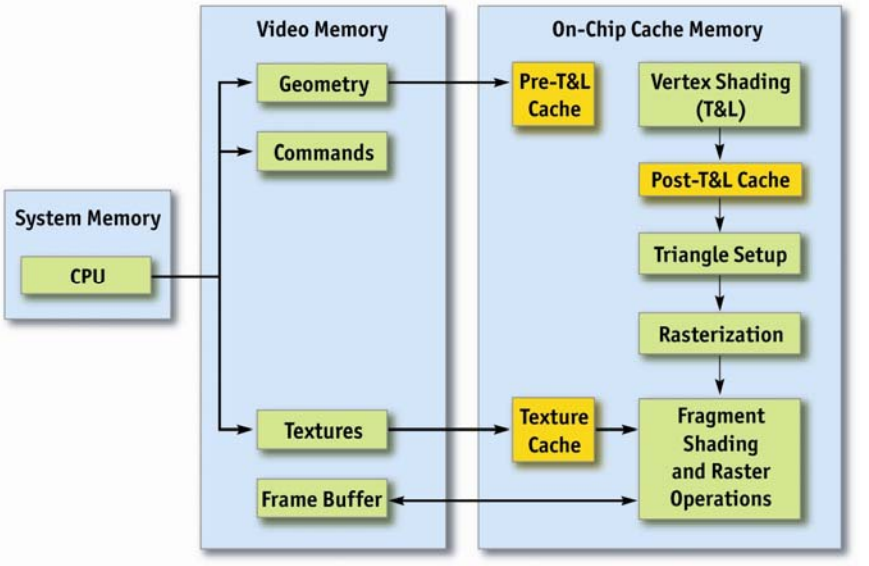


图 1. 潜在的瓶颈

在理想状态下，不会存在任何瓶颈——CPU、AGP 总线和 GPU 数据传输通道的各个阶段都顺利的加载完成（请参见图 1）。然而，在实际中，应用程序无法达到这种状态——总有什么问题会抑制最佳性能的发挥。

瓶颈可能存在于 CPU 或 GPU 中。NVPerfHUD 的绿线（请参见 9.1 节了解更多有关 NVPerfHUD 的详情）能显示在一帧中 GPU 闲置多少毫秒。如果 GPU 每帧闲置一毫秒，这表明应用程序至少部分地受 CPU 的限制。如果 GPU 在帧处理中的大部分时间都是闲置的，或在所有的帧中均闲置一毫秒，且应用程序没有与 CPU 和 GPU 同步，这说明

CPU 是此时最大的瓶颈。提高 GPU 的性能只会增加 GPU 的闲置时间。

2.2.2. 基本测试

您可以进行几个简单的测试来确定应用程序的瓶颈。您不需要使用任何特殊工具或驱动程序，因此这些测试很容易进行。

- **停止文件存取。**任何硬盘数据的存取都将影响帧速率。这种情况很容易发现——只需检查电脑的硬盘指示灯，或使用 Windows 的性能管理工具、AMD 的编码分析家（CodeAnalyst）、(http://www.amd.com/us-en/Processors/DevelopWithAMD/0,,30_2252_3604,00.html)、或 Intel 的 Vtune (http://www.intel.com/software/products/vtune/)来检查磁盘运行管理信号即可。注意，硬盘数据存取也可能是由于内存交换引起的，特别是当您的软件占用大量内存时，也会引起硬盘数据存取。
- **以不同的速度在 CPU 上运行同一 GPU。**您需要找到一个能够调节（如：down-clock）CPU 速度的系统 BIOS，这样才能在同一个系统进行测试。如果帧速率根据 CPU 的速度而做相应地改变，这说明您的应用程序*受到 CPU 的限制*。
- **降低 GPU 的核心时钟（core clock）频率。**您可以通过常用的工具(例如 Coolbits[请参见第 6 章])来进行此操作。如果降低核心时钟频率会相应地降低性能，这说明您的应用程序受顶点着色器、光栅化、或碎片着色器的限制（也就是*受着色器限制*）。
- **降低 GPU 的内存时钟频率。**您可以通过常用的工具(例如 Coolbits[请参见第 6 章])来进行此操作。如果降低内存时钟频率会相应地影响性能，这说明您的应用程序受纹理或帧缓存带宽的限制（*GPU 带宽限制*）。

总之，改变 CPU 的速度、GPU 核心时钟频率和 GPU 内存时钟频率，是迅速确定 CPU 瓶颈或 GPU 瓶颈的简便方法。如果降低 CPU 时钟频率 n 个百分点，导致应用程序性能降低 n 个百分点，那么，该应用程序就是受 CPU 限制的。如果降低 GPU 的核心时钟和内存时钟 n 个百分点，导致应用程序性能降低 n 个百分点，那么，该应用程序就是受 GPU 限制的。

2.2.3. 使用 NVPerfHUD 软件

为了确定 CPU 瓶颈，可以将您的应用程序在一个特殊的驱动程序上运行，该软件叫做“*空硬件*(*Null Hardware*)”驱动程序。除了它事实上并不对 GPU 传送任何任务外，这个驱动程序将像普通驱动程序一样工作（它通过与普通驱动程序完全相同的编码路径运行）。这样，“空硬件”驱动程序便模拟了一个无穷快速的 GPU。*如果“空硬件”驱动程序的性能不如一个普通驱动程序，那么，应用程序就是完全受 CPU 限制的。*

NVPerfHUD 2.0 有一个特殊的模式，可以禁止应用程序所有的绘图调用，因此模拟了一个“空硬件驱动程序”。但是，由于驱动程序不再需要处理和执行任何在绘图调用之前的状态改变，禁止绘图调用还因此减轻了 CPU 的负荷。NVPerfHUD 还有许多其他有用的功能，能帮助您确定软件的性能问题。

如果 NVPerfHUD 显示 GPU 从未闲置，那么应用程序就是受 GPU 限制的。NVPerfHUD 的蓝线显示了驱动程序等待 GPU 的毫秒数，因此能够验证受 GPU 限制的性能。

*《NVPerfHUD 用户指南》*介绍了确定并消除瓶颈、发现并排除故障等的详细方法。下载地址为：
http://developer.nvidia.com/object/nvperfhud_home.html。

2.3. 瓶颈：CPU

如果应用程序是受 CPU 限制的，应使用概要分析来查明什么占用了 CPU 资源。下列模块通常会占用大量的 CPU 资源：

- 应用程序（*可执行程序和相关的动态链接库[DLL]*）
- 驱动程序（*nv4disp.dll, nvoglnt.dll*）
- DirectX Runtime(*d3d9.dll*)
- DirectX 硬件抽象层 (*hal32.dll*)

因为此阶段的目标是减少 CPU 占用率，使 CPU 不再成为瓶颈，所以，查清是什么占用了 CPU 大部分运行时间相对比较重要。通常的方法是：通过较小的最优化改进算法。当然，还需找出占用 CPU 时间最多的程序，以最大限度地提高性能。

接下来，我们需要解决应用程序编码问题，看能否移除或减少编码模块。如果应用程序在文件hal32.dll、d3d9.dll、或nvoglnt.dll上占用了CPU的大量时间，这可能就是应用编程接口（API）的问题了。如果驱动程序占用了CPU的大量时间，可能就需要减少对驱动程序的调用了。提高批处理的规模也有助于减少驱动程序的调用。如需查询有关批处理的详情，请浏览：

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

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

NVPerfHUD 软件也有助于确定驱动程序占用率，该软件能显示驱动程序每帧所花的时间（以红线显示），以及每帧所执行的批处理数目。

当软件性能受 CPU 限制时，还需要检查：

- **应用程序是否锁定了诸如帧缓存器或纹理等资源。**锁定资源可以使 CPU 和 GPU 交替运行，实际上是使 CPU 停转，直到 GPU 回到锁定状态。因此 CPU 正在等待阶段时，不能处理应用程序编码。所以，锁定资源会导致 CPU 的消耗。
- **应用程序是否利用 CPU 保护 GPU。**有时候，绘制小三角形这样的工作会被分配给 CPU 来完成，以节省 GPU 的工作量，但 GPU 此时是闲置的！当软件性能受 CPU 限制时，去除这些 CPU 方面的优化设置实际上会提高性能。
- **考虑把 CPU 的工作转移到 GPU。**您能考虑重新修改运算法则来适应 GPU 的顶点或像素处理器的要求吗？
- **应用着色器增加批处理规模，降低驱动程序资源消耗量。**例如，您可以把两个素材合并到一个着色器中，用一个批处理绘制几何坐标，而无需用两个着色器分别进行两个批处理。在许多情况下，Shader Model 3.0 能有效地把多个批处理合并到一个着色器中，从而同时降低了批处理和绘图资源消耗量。（请参见 4.1 节，了解更多有关 Shader Model 3.0 的情况）

2.4. 瓶颈：GPU

GPU 是依靠数据传输通道的体系架构。如果 GPU 是瓶颈之所在，我们便需要找出哪一个数据传输通道段是最大的瓶颈。如需查询有关各种图形数据传输通道段的详情，请浏览：http://developer.nvidia.com/docs/IO/4449/SUPP/GDC2003_PipelinePerformance.ppt。

NVPerfHUD 软件通过开启或关闭各种 GPU 和驱动程序的功能，将一切变得简单化。例如，该软件能进行MipMap LOD Bias 调整使纹理成四倍解析度。如果性能提高了许多，那说明纹理集成缓存的错失是瓶颈所在。NVPerfHUD 软件通过迫使所有或一部分着色器在单个循环中运行，控制像素着色器的执行次数。

如果您确定了GPU是软件的瓶颈，请用第**Error! Reference source not found.**章中的技巧来提高性能。

Chapter 3.第 3 章
提升图形芯片性能的相关技巧

本章将讲述一些高级技巧，以帮助您在 GeForce FX 及 GeForce 6 系列图形芯片上获得最佳性能。为了便于阅读，这些内容围绕数据传输通道进行组织。每一个小节的内容按照其重要程度粗略地进行了排序，因此您可以知道首先应该采用哪些技巧。

要想对现代图形芯片的数据传输通道性能有一个整体了解，最好参考*《图形芯片宝藏：实时图形的编程技术、技巧及诀窍》*中*“图形数据传输通道性能”*一章。该章介绍了如何确定瓶颈，以及如何处理图形数据传输通道的各部分中潜在的性能问题。

*“图形数据传输通道性能”*一章的内容可以在以下位置免费下载：http://developer.nvidia.com/object/gpu_gems_samples.html。

3.1. 技巧目录

只要使用得当，GeForce FX 和 GeForce 6 系列图形芯片能够达到极佳的性能。本目录列出了提升性能的一些技巧，后面各小节将加以详细介绍。

- **批处理欠佳导致的 CPU（中央处理器）瓶颈**
  - 减少对批处理的使用
    - 使用纹理地图以避免纹理状态的改变。http://developer.nvidia.com/object/nv_texture_tools.html

- 在 DirectX 中使用实例化 API（应用编程接口）以避免像 SetMatrix 这类实例状态发生改变。

- **顶点着色器导致的图形芯片（GPU）瓶颈**
  - 使用索引原语调用
    - 使用 DirectX 9 的网格优化调用 [ID3Dxmesh:OptimizeInplace()或 ID3DXMesh:Optimize()]
    - 如果索引表无法工作，可以使用我们的 NVTriStrip 工具 http://developer.nvidia.com/object/nvtristrip_library.html。

- **像素着色器导致的图形芯片瓶颈**
  - 选择能满足您工作需要的最低版本的像素着色器
    - 在开发自己的着色程序时，您可以使用更高的版本。首先确保它能正常工作，然后找机会降低像素着色程序的版本以实现优化。
  - 如果您需要 ps_2_*功能，请使用 ps_2_a 配置文件（profile）
  - 选择能满足您工作需要的精度最低的数据类型：
    - 最好使用半精度浮点数
  - 尽可能使用上述数据类型：
    - 可变参数
    - 恒定（Uniform）参数
    - 变量
    - 常量
  - 平衡顶点着色器和像素着色器。
  - 如果受到像素着色器的限制，尽可能将可线性化的计算交给顶点着色器完成。
  - 不要将恒定参数应用于（在像素着色器的生命周期中）不发生改变的常量。
  - 尽量使用代数方法减少计算量。
  - 用纹理查找（texture lookup）代替复杂的函数

- 单像素镜面反射光照效果
- 使用 FX 合成器将生成的纹理有计划地烘焙到文件中
- sincos、log、exp 是原生指令，不需要用纹理查找来替换

- **纹理贴图导致的图形芯片瓶颈**
  - 使用 mipmapping 纹理映射
  - 慎用三线过滤（trilinear filtering）和各向异性过滤（anisotropic filtering）
    - 使各向异性过滤的级别和纹理复杂程度相匹配。
    - 使用Photoshop插件改变各向异性过滤的级别，看看更改后的效果如何。
      http://developer.nvidia.com/object/nv_texture_tools.html
      遵循这一经验法则：如果纹理存在干扰，请打开各向异性过滤。
- **光栅化导致的图形芯片瓶颈**
  - 倍速 z-only 和模板渲染
  - Early-z（z 轴精简）优化
- **反锯齿**
  - 如何利用反锯齿功能

3.2. 批处理

3.2.1. 减少批处理的使用

“批处理”是指将几何图集合到一起，使得一个 API 调用就能绘制出许多三角形，而不需要每个三角形都调用一次 API（在最坏的情况下）。每次调用 API 时驱动程序都会占用资源，减少这种消耗的最佳办法就是尽可能少地调用 API。换句话说，就是每次绘制数千个三角形以减少总的绘制调用次数。减少批处理的次数，增大批处理的规模是提高性能的好办法。随着图形芯片的处理能力越来越强大，要达到最佳的渲染速率，有效的批处理变得更加重要。

3.3. 顶点着色器

3.3.1. 使用索引原语调用

使用索引原语调用使得图形芯片可以利用自己的多边形转换和光源处理后顶点高速缓存（post-transform-and-lighting vertex cache）。当图形芯片发现一个转换后的顶点时，它不会再次进行转换，而是利用存储在高速缓存中的转换结果。

在 DirectX 中，您可以使用 ID3DXMesh 类的 OptimizeInPlace()或 Optimize()函数对网格进行优化，并使网格和顶点高速缓存之间的配合更加流畅。

您还可以使用我们自己的NVTriStrip工具来创建经过优化的、适合高速缓存的网格。NVTriStrip是一个独立的程序，其下载地址为：http://developer.nvidia.com/object/nvtristrip_library.html.

3.4. 着色器

高级着色语言提供了一个强大而灵活的工具，使编写着色器程序的工作更加简单。令人遗憾的是，这同时也意味着更容易编写出*慢速*着色器。稍不留神，这些慢速着色器可能会同时到达运行极限，使应用程序异常中止。下面的技巧能防止您为实现简单效果而编写出效率低下的着色器。此外，您还将了解到如何充分利用图形芯片的计算能力。只要使用得当，高端 GeForce FX 图形芯片每个时钟周期能够完成 20 多项操作。而最新的 GeForce 6 系列图形芯片的性能还要强许多倍。

3.4.1. 选择可以正常工作的最低版本的像素着色器

选择能够完成工作的最低版本的像素着色器。例如，当您在每个组件只有 8 位的纹理上进行简单的纹理取码和混合操作时，没有必要使用 ps_2_0 或更高版本的着色器。

3.4.2. 使用 ps_2_a 配置文件（Profile）编译像素着色器

微软公司的 HLSL 编译器（fxc.exe）能根据您要编译的配置文件针对不同的芯片进行优化。如果您使用 GeForce FX 图形芯片并且您的着色器需要 ps_2_0 或更高的版本，您应该使用 ps_2_a 配置文件，它是 ps_2_0 函数的超集，直接配合 GeForce FX 产品系列的图形芯片。编译时使用 ps_2_a 配置文件可能会获得比普通 ps_2_0 配置文件更好的性能。请注意，只有在 2003 年 7 月以后发布的 HLSL 版本才能使用 ps_2_a 配置文件。

总之，您应该使用最新版本的 fxc（以及 DirectX 9.0c 或更新的版本），因为微软公司在每一个版本中都会添加更智能的编译功能，还会解决一些缺陷。对于 GeForce 6 系列图形芯片来说，只要用适当的配置文件和最新的编译器进行编译就足够了。

3.4.3. 选择可以正常工作的精度最低的数据类型

另一个影响性能和影像质量的因素是用于运算和寄存器的数据精度。GeForce FX 和 GeForce 6 系列图形芯片支持 32 位和 16 位的浮点格式（分别称作浮点数和半浮点数），还支持 12 位的定点格式（称作定点数）。浮点数据类型采用的是 s23e8 格式，这非常像 IEEE。半浮点数也像 IEEE，它采用的是 s10e5 格式。12 位的定点数类型包括的范围是 [-2,2]，在 ps_2_0 或更高版本的配置文件中不能使用这种数据类型。在 DirectX 中 ps_1_0 到 ps_1_4 配置文件可以使用定点数类型，在 OpenGL 中 NV_fragment_program 扩展或 Cg 可以使用定点数类型。

这些数据类型的表现会随精度的改变而发生变化：

- 定点数类型最快，适用于低精度运算，例如色彩计算。
- 如果您需要浮点精度，那么半浮点类型的性能要高于浮点类型。谨慎的使用半浮点类型可以将帧速率提高 3 倍，在大多数应用程序中被着色的像素有 99％以上都位于全 32 位计算结果的一个最低有效位（LSB）中！
- 如果您需要尽可能高的精度，请使用浮点数类型。

您可以使用/Gpp 旗标（2003 年 7 月的 HLSL 更新中提供）来强制着色器中的所有部件都使用半浮点精度。当您的着色器能够正常工作并采用了本节中的技巧后，您可以启用这个旗标并看看它对性能和影像质量有

何影响。如果没有错误产生，请继续启用这个旗标。否则请手动降到半浮点精度（/Gpp 旗标为您所能达到的性能设定了一个上限）。

当您使用半浮点数或定点数类型时，请确保是用于可变参数、uniform 参数、变量和常量。如果您使用 DirectX 的汇编语言和 ps_2_0 配置文件，请使用_pp 修饰语以降低计算的精度。

如果您使用OpenGL的 ARB_fragment_program语言时，使用 ARB_precision_hint_fastest选项既可以减少执行时间，还可以降低精度，而使用NV_fragment_program选项则可以控制每条指令的精度。（请浏览 http://www.nvidia.com/dev_content/nvopenglspecs/GL_NV_fragment_program_option.txt）。

许多基于色彩的运算都可以用定点数类型或半浮点数类型的数据来进行，这不会造成任何精度损失（例如 tex2D*diffuseColor 运算）。

在 OpenGL 中 GeForce FX 硬件上，您可以提高以浮点运算为主的着色器的速度，方法是以定点数精度来执行这些运算（例如标准化矢量的点积运算）。

例如，任何标准化的运算结果或色彩都可以是半浮点精度。坐标也可以是半浮点精度，但是在顶点着色器中它们需要进行缩放，使相关的数值接近零。

例如，将数值移动到本地正切空间，然后缩小坐标可以消除由于将大坐标转换成半浮点精度所造成的带状干扰（banding artifacts）。

3.4.4. 使用代数计算来减少运算量

着色器开始正常工作以后，您可以检查一下计算方法，看能否使用数学工具来减少运算量。特别是对于被多个着色器共享的库函数来说这种可能性更大。例如：

- 普通的球面帖图投射常常表示为

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

  这可以展开为：

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

如果投射矢量已经过标准化（请参见 **Error! Reference source not found.** 节和 **Error! Reference source not found.**节），那么，前 3 项之和一定为 1.0。因此这一表达式可重新分解为：

p = sqrt(2 * (Rz + 1)) = 1.414*sqrt(Rz + 1)

- 将这个乘以 1.414 的乘法合并入另一个常量（请参见 **Error! Reference source not found.**节），这可以节省一次点积运算。
- dot(normalize(N)、normalize(L))函数的运算可以更加有效。
  - 它的常规运算方式是(N/|N|) dot (L/|L|)，这需要两个费时的倒数平方根（rsq）计算。
  - 作一些简单的代数计算后结果是：
  - (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))
  - 这只需要一次费时的倒数平方根计算。

3.4.5. 不要把矢量值放入含有多个内插值的标量部件中

在计算中放入太多信息会使编译器难以对您的代码进行有效的优化。例如，如果您要传递一个切线矩阵，不要在 3 q 部件中包含视图矢量（view vector）。下面将对这种错误举例说明：

```
// 错误的代码
tangent = float4(tangentVec, viewVec.x)
binormal = float4(binormalVec, viewVec.y)
normal = float4(normalVec, viewVec.z)
```

视图矢量应该放入第 4 个内插值中。

3.4.6. 不要编写过于通用的库函数

开发员常常编写能被多个着色器共享的函数。例如，投射的计算方法常常是：

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

这种计算方法使投射矢量的计算独立于法矢量（normal vector）或事件矢量（incident vector）的长度。然而，着色器的作者常常希望至少能实现法矢量的标准化，以便执行光照效果的计算。如果是这样，那么点积计算、倒数计算和标量的乘法计算就可以从 reflect()函数中去除。此类优化可以大大提高性能。

3.4.7. 不要计算标准化矢量的长度

过于通用的库函数的一个常见（也是很浪费的）例子就是在本地运算输入矢量的长度。然而，矢量常常在调用函数之前被标准化。编译器检测不到这一点，这意味着实际上每一个像素都执行了计算，计算结果为 1.0。

如果您的库函数必须独立于矢量的长度正确工作，可以考虑在函数中将矢量的长度设为一个标量。这样一来，在调用函数以前先要执行矢量标准化的着色器就能够传递一个数值为 1 的常量（不计算长度能够带来诸多好处），而那些不执行矢量标准化的着色器仍然可以计算长度。

3.4.8. 合并恒定常量（Uniform Constant）表达式

很多开发人员者在他们的像素着色器中计算含有动态常量的表达式。如果表达式中使用了不止一个恒定常量（或一个恒定常量和一个内联常量），通常能够将这些常量合并在一起以提高性能。例如：

```
half4 main(float2 diffuse : TEXCOORD0,
        uniform sampler2D diffuseTex,
        uniform half4 g_OverbrightColor) {
  return tex2D(diffuseTex, diffuse) * g_OverbrightColor * 2.0;
}
```

g_OverbrightColor 可以在 CPU 上预先乘以 2.0，这就不用每一个像素都进行乘法运算，因为每一帧可能有数以百万计的像素。

为了将尽可能多的常数表达式合并，您也许需要对表达式进行分配或分解。此外，您可以在运行着色器之前，使用 HLSL 的预着色器（preshader）在 CPU 上进行预先计算。

另一个常见的例子是计算每一个顶点的 materialColar * lightColor。由于对于某个程序组的所有顶点来说，这个表达式的值是相同的，因此它应该放在 CPU 上计算。

您还应该将矩阵求逆和转置矩阵的计算放在 CPU 而不是图形芯片上进行，因为它们只需要计算一次，而不需要逐顶点或逐碎片进行计算。/Zpr（行优先压缩）和/Zpc（列优先压缩）编译选项有助于将矩阵按照您希望的方式进行储存。

3.4.9. 不要将恒定参数（Uniform Parameter）用于在像素着色器生命周期中不发生改变的常量

开发人员有时候使用恒定参数来传递常用的常量，例如 0、1、255。应该避免这种用法，因为这会使编译器难以辨别常量和着色器参数，因此导致性能下降。

3.4.10. 平衡顶点着色器和像素着色器

取得高性能的关键在于消除瓶颈，这意味着您必须平衡每一条数据传输通道：CPU、AGP 总线，还有图形数据传输通道的级数。使用顶点着色器还是像素着色器取决于几个因素：

- **目标*镶嵌*状况如何？**如果每一帧有几百万个顶点，那您也许想减轻顶点着色器的工作量。当使用多通道（multipass）算法时尤其如此。
- **您要求的分辨率有多高？**如果您希望应用程序在高分辨率下运行，那么，像素着色器很有可能成为瓶颈。因此您应该将更多计算交给顶点着色器去完成。
- **像素着色器有多长？**如果您进行复杂的着色，那么像素着色器可能会成为瓶颈。如果像素着色器编译超过 20 个循环（平均）并占据了大半个屏幕，那您的应用程序可能在 GeForce FX 硬件上受到像素着色器的限制。因此应该尽量将计算工作交给顶点着色器。（请参见 **Error! Reference source not found.**一节的例子）您可以使用我们的 NVShaderPerf 工具来查看着色器使用了多少循环。另外，还要注意，有些新的硬件（例如 GeForce 6 系列）可以使用更复杂的像素着色器程序，而不会很快就受到像素着色器的限制。

3.4.11. 如果受到像素着色器的限制，就把可线性化的计算交给顶点着色器完成

光栅在进行透视角修正处理时会读取每个顶点的数值，并在每个碎片上对这些数值进行插值处理。硬件已经替您完成了这个工作，因此您可以将线性计算交给顶点着色器。您可以为较少的顶点执行这一计算，同时仍然能从像素着色器中获得插值处理的结果。

例如，您可以从立体空间（world space）移动到光照空间（light space）以实现衰减。或者当您做凹凸贴图时可以移动到每个顶点的正切空间，除非您正在将每个像素的投射做成立体贴图。

3.4.12. 使用标准库函数 mul()

mul()标准库函数可以代替手工进行矩阵乘法。当应用程序在内插值中传递矩阵时这可以避免可能出现的行优先/列优先的问题。

3.4.13. 用 D3DTADDRESS_CLAMP (或 GL_CLAMP_TO_EDGE)代替 saturate()以取得附属纹理坐标（Dependent Texture Coordinates）

使用 saturate()会为某些图形芯片增加额外的工作量。如果把得到的运算结果作为纹理坐标，最好使用纹理硬件来把纹理坐标限制在[0..1]范围内，而不要在着色器中这样做。

3.4.14. 首先使用低位内插值

如果您首先使用低位的纹理坐标集（TEXCOORD 集），性能将得到明显提升。先使用 TEXXCOORD0，然后再向上使用 TEXCOORD1、TEXCOORD2，依次类推。

3.5. 纹理贴图

3.5.1. 使用 Mipmapping 纹理映射

为了防止缩小的纹理造成“闪光”状干扰，请始终在应用程序中使用 mipmapping 纹理映射。您将看到更佳的画质、工作更出色的纹理高速缓存，还有更高的性能。实现这一切只需要多使用 33%的内存，这种交换很划算。特别是 3D 纹理，它能从 mipmapping 中得到很大好处，我们已经看到在启用 mipmapping 之后其性能提高了 30%至 40%。

在创建 mipmap 的时候，不要简单地使用 box filter 来生成越来越小的 mipmap，应该使用 Gaussian 或 Mitchell 过滤器得到更多的样本，这样得到的结果质量更好。在创建 mipmap 时会多花一点时间，您可以让程序在运行时表现得更好。我们的 Photoshop 插件（NVIDIA 纹理工具套件的一部分）可以快速创建高质量的 mipmap。这个套件的下载地址为：http://developer.nvidia.com/object/nv_texture_tools.html。

3.5.2. 慎用三线过滤和各向异性过滤

三线过滤和各向异性过滤都能提高图像质量，但都会导致性能下降。尽量只在必要时使用三线过滤和各向异性过滤。一般来说，含有大量高对比度细节的纹理需要使用它们。对于各向异性过滤，您需要考虑纹理的方向。如果您知道纹理将向浏览者倾斜（例如地板纹理），可以提高该纹理的各向异性过滤级别。对于多层纹理表面，每一层都应该进行适当的过滤。

我们的 Adobe Photoshop 插件有助于确定要使用的各向异性过滤的级别。这个工具允许您尝试不同的过滤级别并观看其视觉效果。它的下载地址为：http://developer.nvidia.com/object/nv_texture_tools.html。画家可能希望使用该工具来帮助他们决定哪些纹理需要各向异性过滤或三线过滤。

3.5.3. 用纹理查找代替复杂的函数

纹理是将复杂函数进行编码的最佳方法，可以把它们看做能被编入索引的多维阵列。GeForce FX 产品系列能够有效地访问纹理，其资源占用率通常和算术运算相同。您可以使用我们的 FX 合成器工具来为这类优

化制作原型。FX 合成器的下载地址为：
http://developer.nvidia.com/FXComposer。

如果能够对纹理中复杂的算术运算序列进行编码，就可以提高其性能表现。请记住，有些复杂的函数（例如 log 和 exp）是 ps_2_0 或更高版本的配置文件中的微指令，因此不需要为了优化性能而对其进行编码。

3.5.3.1. 逐像素光照处理

使用 2D 纹理

一个要用到纹理的常见情形是逐像素光照处理。您可以使用 2D 纹理，编索引时将（N dot L）放在一个轴上，（N dot H）放在另一个轴上。在每一个（u,v）位置上纹理都会编码：

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

这是标准的 Blinn 光照模型，包括对漫射和反射条件的限定。

使用 1D 的 ARGB 纹理

使用以（N dot H）为索引的 1D 的 ARGB 纹理是一个有用的诀窍。纹理在每个通道中用不同的指数为（N dot H）编码。例如，它可能编码：

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

然后，每一项都被分配一个指数为 4 的加权常量，该常量和这些值相混合，为着色提供了一个单色反射值。这种方法的好处是它能在 GeForce4 级别的硬件上发挥作用，而且还很灵活，能使用不同的外观。

使用 3D 纹理

您还可以使用 3D 纹理将反射求幂添加到混合中。前两个轴使用上一节讲述的 2D 纹理技巧，第三个轴将反射指数编码（光泽）。

但是请记住，纹理太大可能会影响高速缓存的性能。我们建议您只将最常用的指数进行编码。

3.5.3.2. 矢量的标准化

如果您正在编写 ps_1_*着色器，请使用标准化立体贴图将矢量快速标准化。如果要求的质量更高，您可以使用两个 16 位带标记的立体贴图，一个用于 x 和 y，另一个用于 z。

另一种优化是基于这样一个事实：在实践中，被标准化的矢量 V 的长度经常接近 1，这是因为它们是以内插值替换的或者是经过过滤的法线。这意味着当 x=1 时您可以使用 1 / sqrt(x)的泰勒展开式（Taylor expansion）的第一项求 1 /||V||的近似值：

1 / sqrt(x) ~ 1 + ½ (1 - x)

因此：

V / ||V|| = V / sqrt(||V||^2) = V + ½ V (1 - ||V||^2)

这个公式可以写成两条汇编语言指令：

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

其中 r0 包含 V，r1 的最后一个值包含 V / ||V||。

由于寄存器修饰语 x2 的关系，上面的代码只对 ps1_4 有效。*对于版本较低的像素着色器，可以使用下面的公式：*

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

这里假设 r0 包含½ (V + 1)，这很少造成任何限制，因为 V 常常需要经过从[-1, 1]到[0, 1]的范围压缩并被传递给像素着色器。

GeForce 6 系列图形芯片有一个特殊的半浮点精度标准化单元，该单元可以在一个着色器循环中标准化一个 16 位浮点矢量，而不占用任何系统资源。利用这项功能，只要在一个 16 位浮点矢量上执行一次标准化，编译器就将生成一条 nrmh 指令。

要想进一步了解标准化，请参见我们的《标准化探索》和《凹凸贴图的压缩》白皮书。下载地址为：
http://developer.nvidia.com/object/normalization_heuristics.html
http://developer.nvidia.com/object/bump_map_compression.html

3.5.3.3. sincos()函数

尽管先前有很多建议，但 GeForce Fx 产品系列和后来的图形芯片都在硬件中对一些复杂数学函数提供了支持。sincos 就是这样一个便利的函数，它可以同时计算一个数值的正弦和余弦。

3.6. 性能表现

3.6.1. 倍速 Z-Only 和模板渲染

当只渲染深度和模板数值时，GeForce FX 和 GeForce 6 系列图形芯片能够以双倍的速度进行渲染。要启用这种特殊的渲染模式，您必须遵守以下规则：

- 禁用色彩写入功能（Color writes）
- 活动深度-模板表面未经过多重采样
- 未对任何碎片使用 Texkill
- 未对任何碎片使用深度替换（oDepth, texm3x2depth, texdepth）
- 禁用阿尔法测试（Alpha test）
- 任何活动纹理中都未使用色彩抠像（color key）技术
- 禁用所有用户裁减切面技术（clip planes）

3.6.2. Early-Z 优化

Early-z 优化（有时也叫做“z 轴精简”）技术通过避免渲染被遮蔽的表面来提高性能。如果被遮蔽的表面使用了大量的着色器程序，那么 z 轴精简技术将节省很多计算时间。要想利用 z 轴精简技术，请遵循以下的指南：

- 不要创建有洞的三角形（即避免阿尔法测试或 texkill 指令）
- 不要修改深度（即允许图新芯片使用以内插值替换的深度值）

违反以上规则将会导致图形芯片在进行 Early-z 优化时使用的数据无效，还会导致 z 轴精简被禁用，直到深度缓冲器被再次清空。

3.6.3. 先规定深度

要想利用上述两项功能来提升性能表现，最好的办法就是“先规定深度”。我们的意思是您应该在第一个处理流程中使用倍速深度渲染来绘制场景（不着色），这样建立的表面距离浏览者最近。然后您可以再次渲染场景，但要使用完全着色。Z 轴精简将会自动剔除看不到的碎片，因此节省了着色计算。

“先规定深度”这一操作要求具有自己的渲染层，但如果很多被遮蔽的表面进行过大量渲染，那这个方法可以提高性能表现。当三角形很小时，倍速渲染的效果会差一些，同时小三角形还会降低 z 轴精简的效果。

另一个相关的技术是延迟着色（Deferred Shading），您可以在 NVSDK7.1 及以后的版本中应用到该技术。

3.6.4. 内存分配

为了减低应用程序对显存的占用率，分配着色器和渲染目标的最佳方法是：

1. 先分配渲染目标
    - 根据宽距（宽度*位每像素[width * bpp]）来安排分配顺序。
    - 根据使用频率为不同的宽距组排序。应该首先分配渲染频率最高的表面。
2. 创建顶点和像素着色器
3. 加载剩余的纹理

3.7. 反锯齿技术

GeForce FX 和 GeForce 6 系列拥有功能强大的反锯齿引擎。启用反锯齿引擎能够获得最佳的显示质量，所以建议您启用反锯齿功能。

如果您没有反锯齿功能，又需要使用这项功能，请与我们联系，我们将非常乐意与您共同探讨这个问题并帮助您找到解决方案。

一个问题是使用具有后处理效果的反锯齿技术，目前，这一问题在

DirectX9.0b 及以上的版本中已得到解决。StretchRect()指令可以将后台缓存复制到具有多重采样的脱屏场景中。

例如，如果在一个 100X100 后台缓存区启用 4 倍多重采样，处理器实际上在内部创建了一个 200X200 后台缓存区和深度缓存区来实现反锯齿效果。如果创建一个 100X100 的脱屏场景，StretchRect()指令会将整个后台缓存放置到脱屏表面（off-screen surface），图形芯片就可以把消除锯齿后的缓存过滤到脱屏缓存区。

然后就在 100X100 场景上进行渲染和进行其他的后处理，然后回到后台主缓存区。

您无法将多重采样的 Z 缓存应用到非多重采样的渲染对象上，其原因是，200X200 后台缓存区显示分辨率和 100X100 处理后的显示分辨率不匹配。

Chapter 4.第 4 章
GeForce 6 系列编程技巧

本章将介绍多项十分有用的技巧，帮助您充分利用 GeForce 6 系列和 Quadro FX 4XXX 系列图形芯片的强大性能。虽然大部分内容主要是关于芯片功能的，但是有些还是会影响到性能表现。

4.1. 支持 3.0 着色器模型

在高级顶点和像素着色器技术方面，微软公司 DirectX9.0 引入了几项新标准，即 2.0 版本和 3.0 版本。2.0 版本的着色器模型硬件已于 2002 下半年发布，现在推出的大多数图形芯片都支持 2.0 及以上版本的着色器模型。2.0 版本的着色器模型可应用高级光照技术和动画技术，但是着色程序长度和复杂程度有限，这相应限制了所能实现的效果。

由于程序开发人员不满足于 2.0 像素着色器和 2.0 顶点着色器所固有的局限性，所以他们开始采用更新、更高级的 3.0 版本着色器模型。3.0 版本的着色器模型在像素和顶点着色器处理方面的多个领域中都有所突破。

4.1.1. 3.0 像素着色器

下表列出了 2.0 像素着色器和 3.0 像素着色器之间的主要功能差别：

像素着色器功能	2.0 着色器	3.0 着色器	描述
着色器程序长度	96	65535+	可以处理更多复杂的着色、灯光和过程材质
动态分支功能	无	有	可以跳过无关像素的复杂着色处理，提升性能
着色器反锯齿功能	不支持	内建衍生指令	开发人员可以计算任何函数的屏幕空间衍生值，调整着色器频率或通过超采样（over-sampling）消除干扰
隐面寄存器	无	有	支持单通道双面光照效果
内插值色彩格式	最小 8 位整数	最小 32 位浮点	更大范围和更高精度的色彩，支持顶点高动态范围的光照效果
多渲染目标	可选	必须 4 个	支持高级光照逻辑算法，以节约过滤和顶点着色工作——以最少的消耗，取得最佳效果
雾化和反射	最小 8 位固定功能	自定义 fp16-fp32 着色器程序	3.0 着色器模型允许开发人员全面、准确地控制反射和雾化效果计算，以往的版本中使用固定功能
纹理坐标点计数器	8	10	更多的“逐像素”输入，渲染的真实感更强，尤其适合皮肤材质的渲染时

4.1.2. 3.0 顶点着色器

下表列出了从 2.0 版本顶点着色器模型转变为 3.0 版本顶点着色器模型时，开发人员可用的主要功能：

顶点着色器功	2.0 着色器	3.0 着色器	描述

能			
着色器程序长度	256 条指令	65535 条指令	指令越多，角色的光照效果和动画效果越细腻逼真
动态分支功能	无	有	可以条过无关定点的动画和计算，提升运算速度
顶点纹理	无	任何数量的搜索，最高为 4 个纹理	支持位移贴图和粒子效果
支持实例功能	无	必须	使用单个命令可以绘制多个不同的对象

4.1.3. 动态分支功能

3.0 着色器模型（顶点和像素着色器）都具有一个主要功能——动态分支功能。简言之，该功能允许允许着色创建人员在着色程序中建立真实循环语句和条件语句。例如，可以编写一个通过一定数量的顶点光照点的含循环语句的着色器，并决定哪些可能影响某个特定顶点，然后再将每个相关光照点的索引数据传送到像素着色器。像素着色器将利用这些“光照索引”确定使用哪些光照参数。然后，像素着色器将对有效光照进行循环操作，一旦所有的光照点处理完成，像素着色器就使用动态分支功能退出着色器。

大部分的光照效果仅仅适用于对象的前方——正对光源的那一面，因此，您可以同时使用顶点分支功能和像素分支功能跳过一些着色器检测为远离光源的光照点的处理（利用新的隐面寄存器），这样可以节省大量处理时间，并提高着色器效率。角色骨骼动画的处理和其他许多类似计算工作都可以运用上述加速技巧。

4.1.4. 代码维护更简捷

由于游戏引擎变得越来越复杂，所以经常要为每个着色器创建许多不同的版本，以便使其适合 2.0 像素着色器程序长度的限制。这样就加大了程序代码维护难度，增加了着色程序编写时间和层装载时间，而且运行时会耗用大量宝贵的系统内存。但是 3.0 着色器模型解决了这一问题，通过大量运用循环程序和分支程序，游戏引擎内可以编写一个单顶点、单像素着色器，该着色器包含相应的静态和动态分支语句，在运行时选择正确执行路径，从而极大地简化了着色合成激增问题。

4.1.5. 实例功能

3.0 版本着色器模型的另一个重要功能就是支持微软 DirectX®的实例应用编程接口（API）。目前，游戏场景中能够显示的不同对象的数量有限，这不是因为图形芯片功率的问题，而是由于 CPU 在存储或提交同一对象略有不同的实例时，通常会消耗大量资源。例如森林由许多相似的树组成，但是每棵树的位置、高度、树叶色彩等方面各不相同。为了表现这些变化，开发人员要么需要存储很多棵单独的树木原型（每颗树只是稍有不同），要么需要耗用大量资源来渲染状态变化，对每棵树进行旋转、缩放、着色和布置。

实例功能允许开发人员先存储一棵树，然后再存储几个其他定点数据流，定义每个实例的色彩、高度和树枝大小等。例如，一棵 1,000 顶点的树木模型会包含顶点位置和法线数据等，而 200 个元素的顶点数据流则包含顶点位置、色彩和高度等数据。实例功能允许开发人员提交单个绘图指令，利用基本树木模型相同的数据去渲染 200 棵树中的每一棵，然后再通过每个实例数据流渲染树木的其他不同特性。

如需查询实例功能代码样例，请浏览：
http://download.nvidia.com/developer/SDK/Individual_Samples/samples.html

4.1.6. 小结

总的来说，DirectX9.0 下的 3.0 版本着色器模型在易用性、性能表现和着色器先进程度方面都向前迈出了一大步，动态分支功能可以使程序及早运行，使很多算法加速，同时在图形引擎和图形工具方面简化了着色器编码路经。因此，实例功能可以在很低的 CPU 和内存消耗下，完成极其复杂的处理。

4.2. sRGB 解码功能

sRGB 解码是一种格式，它可以利用伽马转换（gamma conversion）来取得近乎绝对的更高精确度，这种格式可以模拟人类视觉系统。另外，sRGB 还支持 DXT 压缩格式，这使得应用程序可从增加色彩真实度和降低存储容量两个方面受益。

在 GeForce 6 系列的图形芯片中，有时候您可能希望使用浮点格式，使用浮点格式有以下几个优点：

- 整个范围内的更高精度
- 更丰富的线性动态范围
- 帧缓存线性合成

对纹理来说，由于浮点格式对路径和带宽要求更小的内存，所以 sRGB 在许多情况下大量采用浮点格式。

4.3. 单阿尔法（Alpha）合成

GeForce 6 系列图形芯片允许您定义色彩和阿尔法（Alpha）通道的单独合成功能，您可以更灵活地选择纹理阿尔法（Alpha）通道中的存储值。其中的一个用途就是在保存阿尔法（Alpha）通道的同时可以调整色彩通道。

4.4. 支持的纹理格式

下表列出了 GeForce 6 系列图形芯片所支持的纹理格式：

整数格式	2D	立体	3D	MIP	过滤器	SRGB	渲染器	合成器	顶点
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

浮点格式	2D	立体	3D	MIP	过滤器	SRGB	渲染器	合成器	顶点
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

阴影贴图	2D	立体	3D	MIP	过滤器	SRGB	渲染器	合成器	顶点
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. 浮点纹理

GeForce 6 系列的图形芯片为浮点纹理提供了更好的支持。下表列出了支持 16 位材质（fp16）和 32 位材质(fp32)的浮点纹理的各种功能。

纹理材质类型	最近邻过滤	双线性和三线性过滤	各向异性过滤	支持 Mipmap	3D 纹理	立体贴图	非两次方纹理
16 位	支持	支持	支持	支持	支持	支持	支持
32 位	支持	不支持	不支持	支持	支持	支持	支持

4.5.1. 局限性

请注意，浮点纹理不支持 R16F 格式，而是支持 G16R16F 格式。另外，您只能合成到 A16B16G16F16F 表面，而不是到 G16R16F 或 R32F 表面，而且也只支持 G16R16F 纹理的过滤器。

4.6. 多渲染目标（MRTs）

GeForce 6 系列图形芯片支持多渲染目标（MRTs），所谓多渲染目标就是允许一个像素着色器将着色数据写入 4 个不同的渲染目标。一旦像素着色器需要计算 4 个以上的的浮点值并将这些中间值存储在纹理中时，多渲染目标功能将十分有用。

多渲染目标（MRT）的典型用途就是同时计算粒子物理属性、位置和速度参数，以及处理类似的图形芯片算法。延迟着色是多渲染目标的另一项功能，它可以同时计算和存储多个 float4 值：它可以计算所有的材质特性，例如，表面特性，散射和反射材质特性等，并可以将这些材质特性分别存储在单个纹理中。在随后的通道中进行多光源场景处理时，这些材质特性将十分有用。

在 DirectX 程序代码的 NumSimultaneousRTs 表明了图形芯片能够同时渲染的目标数目。GeForce 6 系列图形芯片可以同时渲染 4 个目标。利用 SetRenderTarget(index, pRenderTarget)应用编程接口（API）指令，可以启用多渲染目标功能。这一指令将在渲染目标纹理中传送的目标附加到给定的渲染索引中。然后，像素着色器将利用 oC0、 oC1、oC2、和 oC3 输出寄存器将这些数据输出到这些附加渲染目标纹理上。最后，请记住将 1 到 3 渲染索引目标重置为 NULL，以关闭多渲染目标功能。

多渲染目标功能同时也限制了图形芯片的其他功能的使用。最重要的是，硬件加速反锯齿功能无法用于多渲染目标功能。而且，所有渲染目标必须具有同样的宽度、高度和深度。对 GeForce 6 系列图形芯片而言，这意味着可以在 32 位格式（例如，A8R8G8B8、X8R8G8B8、G16R16F 和 R32F）中可以自由混合匹配；而在在每个 64 位格式（例如，使用 A16R16G16B16F 格式）中则可以使用多达 4 个渲染目标；同样，在每个 128 位格式（例如，使用 A32R32G32B32F 格式）中也可以使用多达 4 个渲染目标。

另外，只有在多渲染目标（MRT）中才可以有阿尔法（alpha）合成、阿尔法（alpha）测试、雾化和抖动效果等后期像素着色器合成功能。要实现这些效果需要设置 D3DMISCCAPS_MRTPOSTPIXELSHADERBLENDING 程序代码，并且要将 USAGE_QUERY_POSTPIXELSHADERBLENDING 渲染格式设置为是（yes）才行。

GeForce 6 系列芯片支持除 R32F、G16R16F 和 A32R32G32B32F 之外的所有多渲染目标格式，因此，特别要注意 GeForce 6 系列图形芯片（除 GeForce6200 之外）一定支持 A16R16G16B16F 浮点格式渲染目标的后期合成。DirectX 规范会对多渲染目标后期合成操作进行进一步的修改：使用多渲染目标（MRT）功能进行渲染时不管其抖动状态，只是注重在渲染目标为零时（1 至 3 渲染目标就假设关闭雾化）的雾化状态，而阿尔法测试只是利用 oC0.a 值来确定是否丢掉全部 4 个渲染目标的像素。

最后，使用多渲染目标还存在性能方面的问题。使用多渲染目标功能时，会占用大量与帧缓存相关的带宽，尤其是在更大的渲染深度时。例如，渲染 4 个 A32R32G32B32F 表面占用的图帧带宽是渲染 1 个 A8R8G8B8 表面的 16 倍。另外，GeForce 6 系列在同时使用 3 个或以下的渲染目标时具备性能友好场景。

因此，请遵循以下几点建议：只有在需要时，才使用多渲染目标功能，例如，使用多渲染目标存储多个通道时。最大限度地减少渲染目标数目和位深度，例如，对数据进行打包处理，不浪费阿尔法处理通道等。最后一个就是尽早分配多渲染目标功能的渲染目标（请参阅 3.6.4 章节的“分配内存”）。

另外，将多渲染目标的输出数据分成 3 组或以下，同时不增加处理通道的总数。例如，如果处理器在要渲染一个随后要输出到 4 个多渲染目标的环境通道时，那么就先输出这些目标中的一个，然后再输出另外 3 个。如果渲染目标中的一个能够以比其他渲染目标更低的精度存储，这样做尤其有效，并且也容易单独计算其他的渲染目标（如材质散射纹理贴图）。您可以在SDK7.1 版本中延迟着色这一部分获取更多的信息，或下载我们的延迟着色演示文稿：
ftp://download.nvidia.com/developer/Movies/NV40-LowRes-Clips/Deferred_Shading.avi.

4.7. 顶点纹理贴图

GeForce 6 系列图形芯片还支持顶点纹理功能，但是顶点纹理不应看作连续的 RAM。顶点纹理在提取数据时不是真正的连续读取，而是会产生等待时间。因此使用顶点纹理的最佳方法就是先进行纹理提取，然后进行逻辑算法计算，这样能在使用纹理提取前避免等待时间。

顶点纹理不是用来**代替大量的常量的阵列**，而是用于减少顶点数据，这样每个顶点只有少量的顶点纹理需要提取数据。

如需了解更多有关顶点纹理的情况，请阅读使用顶点纹理白皮书，网址是：
http://developer.nvidia.com/object/using_vertex_textures.html.

4.8. 总的性能建议

GeForce 6 系列的架构拥有许多更有效、更全面的改进功能，下面是一些小技巧，可以帮助您充分利用这些功能：

- **使用写入掩码和调和器：**GeForce 6 系列着色器架构能在不同的向量（交叉和双向）上完成 4 材质向量的一部分，这样就能改善着色器的使用。通过使用写入掩码和调和器可有助于编译程序识别这些进度条件语句的类型。
- **尽可能使用部分精度：**在 GeForce6 系列图形芯片中使用部分精度有两个原因：一个是 GeForce 6 系列在着色器中有个特殊的自由 fp16 标准化单元，这样可以实现 16 位浮点标准化，从而可以与其他运算保持有效同步。要使用好这个特点，只需要在程序中尽可能适当使用部分精度。另一个原因是使用部分精度可以缓解寄存器的存储压力并进而提高性能。
- **在分支语句比较连贯时使用动态分支功能：**正如 1.1.3 节中所提到的，动态分支功能可以使程序代码更简捷，更容易实施。但是为了达到最佳状态，分支语句应相当的连贯（如超过大约 30X30 像素的区域）。

4.9. 法线图

如果法线图存储对您的应用是一个问题，利用半球重映射技术消除以下组成中的一个，达到法线图对单位长度正切空间法线在 GeForce 6 图形芯片上的压缩：

N.z = sqrt(1 – N.x*N.x – N.y*N.y);

这个式子编译成 3-5 像素着色器指令，所以，这个技术可能会带来性能改进，这取决于这些指令是否能与其他（以前存在的）着色器指令协同发布，以及提取纹理是否成为瓶颈。

如果您决定继续使用半球重映射技术， GeForce 6 图形芯片的纹理结构格式是 DirectX 取 D3Dfmt_V8U8，OpenGL 取 GL_LUMINANCE8_ALPHA8。这些格式是 16 位/像素，它可以提供 2：1 的无损压缩。

请注意，半球重映射技术在某种程度上确降低了灵活性，因为只生成正的单位长度法线。那些依赖负值（比如对象空间法线映射）或非单位法线（比如 http://developer.nvidia.com/object/mipmapping_normal_maps.html里描述的反锯齿技术）则不可能生成。

如需查询更多有关法线图的详情，请浏览 http://developer.nvidia.com/object/bump_map_compression.html，参阅我们的*凹凸贴图压缩*白皮书。

为了生成高质量的法线图，使低*聚合*模型看起来象高聚合模型，应使用 NVIDIA Melody。只需加载您的低聚合工作模型，然后加载您的高聚合参考模型，点击“生成法线图”按钮，并看着调节器顺利进行工作。如需查询有关调节器的资料，请浏览：http://developer.nvidia.com/object/melody_home.html。.

Chapter 5.第 5 章
GeForce FX 编程技巧

本章将列举一些有用的技巧，帮助您充分运用 GeForce 6 系列的强大性能。虽然大部分内容主要是关于芯片功能的，但是有些还是会影响到性能表现。

5.1. 顶点着色器

GeForce FX 的顶点引擎功能强大，在 GeForce FX 5900 图形芯片上，每秒钟可以完成 2 亿多个三角形。它支持完全的动态分支（dynamic branching）功能，该功能允许着色器作者根据光线、骨架（bones）以及其他元素的数量和种类进行分支。

由于每一个活动的分支线程都会降低程序执行的整体速度，因此通过应用编程接口（API）进行分支只适用于两种情况：节省大量的顶点运算或增加基本批处理的规模。

5.2. 像素着色器的长度

如果使用 ps_2_a 或 ps_2_x 配置文件进行编译，在 DirectX 中 GeForce FX 每次能执行 512 条 Direct3D 像素指令或 1,024 条 OpenGL 像素指令。在 OpenGL 中，您可以使用 GLSL、Cg（使用

arbfp1 或 fp30 配置文件进行编译），也可以直接使用 ARB_fragment_program 扩展。

Quadro FX 显卡每次能处理 2,048 条像素指令。

5.3. DirectX 专用的像素着色器

在默认情况下，最新的 DirectX 9 ps_2_0 和更高版本的着色模型要求以 24 位或更高的精度进行数学或临时量的计算（在这种情况下，GeForce Fx 系列要求使用 32 位浮点数）。应用程序可以在汇编程序中规定一个_pp 修饰语，以达到 16 位浮点精度。

在使用 HLSL 或 Cg 时，这项任务变得非常简单——要想使用 32 位精度就把您的变量声明为浮点数，要想使用 16 位精度就声明为半浮点数。

我们建议先用最方便的方法编写着色器，然后针对寄存器使用和半浮点精度的要求进行优化。

定点合成主要用于固定功能纹理合成管线，以及 ps_1_0 至 ps_1_4 着色器模型的纹理运算部分。

在 DirectX 中，您不能通过 ps_2_0 获得定点精度。如果程序能够使用 ps_1_1 至 ps_1_4，那么其运行速度通常能加快，因为这增加了对定点着色硬件的使用。

5.4. OpenGL 专用的像素着色器

默认情况下 ARB_fragment_program 扩展需要不低于 24 位的浮点精度。为了控制精度，您可以在 ARB_fragment_program 源代码的顶部加上各种旗标：

- NV_fragment_program。该旗标允许公开使用半浮点格式（16 位浮点数）和定点（12 位定点）格式，进行最大控制和性能优化。
- ARB_precision_hint_fastest。统一编译器（Unified Compiler）将在每个着色器运行时为其选择适当的精度，其好处是既提高了整体性能，又可以将视觉干扰降到最低限度。

- ARB_precision_hint_nicest。强制整个程序以浮点精度运行。

5.5. 使用 16 位浮点数

很多开发人员以前从未使用过半浮点数，在他们看来浮点数只是一种“能够工作”的格式，却很少考虑其值域和精度。半浮点数有 10 个尾数位和 5 个指数位，浮点数有 23 个尾数位和 8 个指数位。每个尾数位都可以被看作标尺上的刻度标记，它规定了该格式的最大精度。在半浮点数中，每个尾数位之间的精度是 0.1%。指数位的值可以被看作标尺的长度，指数位的值越高，标尺也就越长，而标尺上每个刻度所代表的单位长度也就越大。这就是浮点数格式能自动以精度换算值域的原理所在。

您只能使用半浮点数来表示－2048 到 2048 之间的整数，没有小数位。如果您要在像素着色器中进行 view-或 world-space 运算，那么半浮点精度可能不够用。例如，如果角色位于 4096，而光线位于 4097，两个角色均使用相同的 16 位浮点数表示。当您提取这些数值的时候将会得到 0。接着对结果开平方和标准化将得到 INF，工作区将变得简单、美观甚至比原来的着色器更快。将矩阵和矢量的减法运算移入顶点着色器中。

首次使用 GeForce FX 图形芯片时遇到的光照问题通常是由于错误使用半浮点格式所引起的。这不足为怪，虽然半浮点格式大量应用于影片的色彩，但在游戏中并不常用。

顶点着色器至少要支持浮点数，这样它们才能轻松处理大的世界和视觉空间。同时也不难理解为什么把线性计算先交给顶点着色器来进行——当某些计算可以逐顶点进行然后由图形芯片进行重复计算时，何必仍然要逐像素进行重新计算？

因此，我们建议在顶点着色器中将顶点移入光照空间（light space）或正切空间，并将得到的坐标传递给像素着色器。有个很好的办法，就是从顶点坐标中减去光线坐标，然后用一个常数统一测量矢量，使它的值接近 0，最后在碎片着色器中（fragment shader）对该矢量进行标准化处理。

我们的目的常常是在正切空间中进行光照运算。这时候上述方法特别有用，因为它使您将坐标系的中心置于顶点上。让矢量的值接近 0 还让您可以使用符号位，这就为您增加了一个可用的尾数位。

总之，在 CPU 中进行常量运算，在顶点着色器中进行线性运算，在像素着色器中进行非线性运算。

5.6. 支持的纹理格式

整数格式	2D	立体	3D	MIP	过滤器	SRGB	渲染器	合成器	顶点
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

浮点格式	2D	立体	3D	MIP	过滤器	SRGB	渲染器	合成器	顶点
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

阴影贴图	2D	立体	3D	MIP	过滤器	SRGB	渲染器	合成器	顶点
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

* 在 DX9.0c 中进行曝光时不能使用纹理外包（wrapping）或 mipmapping 纹理映射功能

5.7. 在 DirectX 中使用 ps_2_x 和 ps_2_a

GeForce FX 还支持数种 ps_2_0 中没有的功能，包括倒数计算（通过 DDX 和 DDV）、更长的着色器和预测支持。通过高级着色语言使用该功能有两种方法。第一种方法是使用 HLSL 或 Cg 和 ps_2_x 配置文件进行编译，这样做可以使用上述功能，但无法检查特定的功能位（capability bits）组。更好的方法是使用 ps_2_a 配置文件，该配置文件完全符合 GeForce FX 系列的着色能力，所生成的代码也更加完善。

5.8. 使用浮点数渲染目标

GeForce FX 系列支持 64 位和 128 位的 4 矢量要素（four-component）浮点渲染目标，还支持 32 位的 1 矢量要素或 2 矢量要素浮点渲染目标，以及经过 mipmap 处理的纹理。

在 OpenGL 下对浮点纹理进行曝光是通过 NV_float_buffer 扩展（expose）来进行的，使用 NV_fragment_program 的 pack/unpack 指令还可以把多个低精度矢量要素放入精度更高的单个矢量要素中。应用程序可以在普通的 RGBA8 纹理上使用 pack/unpack 指令模拟 NEAREST_MIPMAP_NEAREST 浮点立体帖图（cubemap）和 3D 体积纹理（volume texture）（该支持仅限于单个 fp32 或两个 fp16 矢量要素）。

在 GeForce FX 硬件上，浮点渲染目标不支持混合，每个纹理元素大于 32 位的浮点纹理也不支持 mipmapping 或过滤。

5.9. 法线帖图

GeForce FX、GeForce4 和 GeForce3 图形芯片有专用的硬件，用于对来自于 2 矢量要素法线帖图进行法线帖图半球再映射（remapping）。对于这些图形芯片，您应该使用 CxV8U8 格式，因为这样做比从着色器中获取 Z 轴更快。

如需查询更多有关法线帖图的详情，请参阅凹凸帖图压缩白皮书，下载地址为：
http://developer.nvidia.com/object/bump_map_compression.html.

要创建能使低聚（low-poly）模型看起来像高聚（high-poly）模型的高质量法线帖图，请使用NVIDIA Melody。方法很简单：先加载能正常工作的低聚模型，然后加载高聚参考模型，再点击“生成法线帖图”，调节器（Melody）就能完成剩余的工作了。调节器（Melody）的下载地址为：http://developer.nvidia.com/object/melody_home.html。

5.10. 新的芯片和架构

GeForce FX 架构现在广泛应用于各种型号和价格的图形芯片。该系列的每款产品都具有相同的顶点着色和像素着色功能。唯一的区别在于内部性能，但这对于开发人员来说是看不到的。这些功能能够让开发人员轻松地针对 GeForce FX 或更新的版本扩展自己的游戏。例如，仅通过几何的细节层次（LOD）或屏幕分辨率处理游戏的扩展性。

GeForce 6 系列图形芯片具有更快的浮点性能，之所以继续支持半浮点数，是因为半浮点数具有更好的性能。与定点类型相比，这些芯片对于浮点和半浮点类型、浮点渲染目标、浮点纹理还具有更好的正交性。

5.11. 小结

在整个行业中，从长着色器程序到真正的倒数计算，GeForce FX 和 GeForce 6 系列架构都具有最灵活的着色器能力。然而，在 GeForce FX 硬件上，纯浮点着色器的运行速度不如定点和浮点相结合的着色器。

对于很多着色器来说，在 GeForce FX 架构上达到最佳性能的最好方法也许是将 ps_1_*和 ps_2_*混合使用。例如，在进行逐象素光照计算时，可以用 ps_1_1 着色器计算漫射光，使用 ps_1_4 或 ps_2_0 着色器计算另一通道的反射光，这样计算速度可能会更快。

Chapter 6.第 6 章
总则

本章将介绍可用于多个图形芯片产品系列的图形芯片一般编程要求。

6.1. 识别图形芯片

过去，开发人员常常（通过 Windows 系统）查询图形芯片设备的 ID 以确定所使用的图形芯片的型号，而这些设备 ID 都是一成不变地增加。然而，随着 GeForce 6 产品系列的问世，这种情况将不会再出现了。因此，我们建议您利用程序设计代码（在 DirectX 中）或者扩展字符串（在 OpenGL 中）来确定您正使用的图形芯片的功能。如果您正在使用 OpenGL 渲染器字符串，应注意并非所有基于 NV40 的芯片名称中都含有“FX”（它们通常命名为 GeForce 6XXX 或 Quadro FX3400）。

设备 ID 经常被开发人员用来减少辅助指令。一旦弄错了设备 ID，您就必须重新**创建辅助指令**。通常，当我们创建一个新的图形芯片时，许多应用程序并不能识别、运行这个图形芯片。

一个未被足够重视的关键问题是**一个未被识别的设备 ID 不能为您提供任何信息**。在未识别设备 ID 之前，请勿采取任何过激行为。

设备 ID 的唯一用途就是在正确识别该 ID 后，采取行动来解决相关问题。

一部分游戏程序错误地将 GeForce 6 系列的图形芯片识别为 TNT 系列的图形芯片，或者不能正确识别设备 ID，而导致 GeForce 6 系列的图形芯片不能正常使用。NV4X 系列芯片是迄今为止功能最为强大的图形芯片，由于一些游戏程序编码十分糟糕而导致无法运行该图形芯片，这简直是一场恶梦。

另外，设备 ID 也**不能替代**程序设计代码和扩展字符串。由于各种原因，程序设计代码也会逐渐发生变化。通常情况下，由于设计规格更加严格、明晰，或者由于某种驱动器或硬件设备在维护成本或维护难度方面的原因，程序设计代码也会一段时间有效，一段时间失效。

渲染目标和纹理格式有时也会失效，所以一定要进行检查支持情况。
如果对设备ID有任何疑问，请发送电子邮电到我们的开发协调组：devrelfeedback@nvidia.com。
如需查询当前所有NVIDIA图形芯片的设备ID列表，请浏览：http://developer.nvidia.com/object/device_ids.html.

6.2. 硬件阴影贴图

NVIDIA GeForce 3 图形芯片及更高版本的 NVIDIA 硬件就已经支持 OpenGL 和 DirectX 中的硬件阴影贴图功能了。“硬件阴影贴图”表示我们已经使用特殊的晶体管进行专门的阴影贴图深度对比和更大比例过滤操作。由于这项功能能够极其有效地显示更高品质的过滤后的阴影贴图轮廓，所以我们建议您好好利用这项功能。由于硬件阴影映射使用晶体管，所以如果您尝试使用高于 PS-2-0 的运算法则来仿效我们的阴影贴图，将会损失一部分显示性能和画面质量。

如果您想在游戏引擎中使用阴影贴图，请浏览：

- 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 *透视阴影贴图*
- *透视阴影贴图：《GPU宝典》：编程技术、技巧和实时图形诀窍中的“填充和注意事项”* (http://developer.nvidia.com/GPUGems)

Simon Kozlov在*《GPU宝典》*的“透视阴影贴图：填充和注意事项”一章中，对现在仍在使用的透视阴影贴图提出了一些改进措施。我们将Kozlov提出的这些改进措施运用到了我们自己的显示引擎中，结果我们发现在实际环境中效果非常好。这一点在SDK 7.0 及以上版本中表现出来。详情请浏览：http://download.nvidia.com/developer/SDK/Individual_Samples/samples.html。

在 DirectX 中，您可以按照以下方法创建硬件阴影贴图：

1) 使用 D3DUSAGE_DEPTHSTENCIL 创建一个纹理。
2) 其格式应设置为 D3DFMT_D16、D3DFMT_D24X8（或 D3DFMT_D24S8，但您无法在着色器中使用模板位）。
3) 使用 GetSurfaceLevel(0)获得 IDirectDrawSurface9 界面。
4) 在 SetDepthStencilSurface()中将表面指针设置为 Z 缓存。
5) DirectX 也要求您设置一个色彩渲染目标，但是您也可以通过将 D3DRS_COLORWRITEENABLE 设置为零来禁用色彩编写。
6) 将阴影构造几何点渲染到阴影贴图 Z 缓存中。
7) 通过这个步骤保存完成投射视图矩阵。
8) 转变渲染目标和 Z 缓存到主场景缓存。
9) 将阴影贴图纹理附加到采样器上，并按以下公式设置纹理坐标点：

   V’ = Bias(0.5/TexWidth, 0.5/TexHeight, 0) *
   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

通过 CPU 连接建立矩阵点，并且通过顶点着色器或者固定功能传输通道的纹理矩阵来实现矩阵连接的转换。

10) 如果使用固定功能数据传输通道或 PS-1.0-1.3，应将投射旗标设置为 D3DTTFF_COUNT4 | D3DTTFF_PROJECTED。

11) 如果使用 1.4 或以上像素着色器时，应从阴影贴图采样器中提取投设的纹理。

12) 硬件使用阴影贴图纹理坐标投射的 X 和 Y 坐标点来查找纹理。

13) 对比阴影贴图深度值和纹理坐标点 Z 方向投射值，如果纹理坐标深度坐标值比阴影贴图深度值大，那么提取的纹理值就认为是 0（在阴影中），反之，则认为是 1。

14) 如果在阴影贴图采样器中启用 D3DFILTER_LINEAR，硬件就会进行 4 深度对比，双线性过滤与一个采样所占用的资源是**一样的——**但效果会更好。

利用该值来调节灯光效果。

早期版本（45.23 或以前）的 NVIDIA 驱动程序没有明确进行阴影贴图投影，这在 52.16 及以后的版本中得到了解决——编程人员现在需要明确设置适当的纹理阶段旗标，尤其是使用 PS2.0 着色模型进行阴影贴图的编程人员必须明确提供一个投射纹理寻址方法（例如，tex2Dproj(ShadowMapSampler, IN.TexCoord0.rgb)。可以手动模拟 W-Divide，，但随后的非投射纹理寻址却不起作用！例如，tex2D(ShadowMapSampler, IN.TexCoord0/IN.TexCoord0.w)就不起作用。

同样，在使用 PS1.1-1.3 的着色器模型时，52.16 及更高版本的驱动程序现在也需要对阴影贴图的纹理阶段采样明确设置投射旗标（例如，SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_PROJECTED)。

注意：在 61.71 及更高版本中的 ForceWare 中，任何纹理指令（tex2D、tex2Dlod、等等）都可以正确使用阴影贴图。

SDK中有在DirectX和OpenGL中设置硬件阴影贴图的实例，详情请浏览：

http://download.nvidia.com/developer/SDK/Individual_Samples/samples.html。

Chapter 7.第 7 章
NVIDIA SLI 与多图形芯片的性能

本章将为您提供几个非常有用的技巧，使您能够在使用多图形芯片配置（如NVIDIA公司的NVIDIA SLI技术）时，获得最佳的图形显示质量。如需查询更多详情，请浏览：ftp://download.nvidia.com/developer/presentations/2004/GPU_Jackpot/SLI_and_Stereo.pdf.

7.1. 什么是 SLI?

SLI（“可扩展连接接口”的缩写）允许多个图形芯片同时工作以获得更高的性能。基于 SLI 技术的主板称之为“PCI Express”主板，它带

有两个 x16 物理信道，每个插槽可配置一个“PCI Express”图形显卡。当两个图形显卡通过一个外置的桥式连接器连通后，驱动程序能自动识别该配置并进入“SLI Multi-GPU”模式。在“SLI Multi-GPU”模式下，驱动程序将两个显卡配置为一个独立的设备：也就是说，所有的图形处理程序将这两个图形芯片视为一个独立的逻辑设备。

由于驱动程序将渲染工作分割给两个物理图形芯片进行处理，使得这个独立的逻辑设备的运行速度最高可达到单个图形芯片的 1.9 倍。但是需要注意的是，SLI 模式运行时并不具有双倍的视频内存，比如，插入两个 256M 内存的图形显卡仍然最多只有 256MB 的视频内存。这是因为驱动程序在一般情况下会在两个图形芯片复制所有的视频内存，也就是说，在任何情况下，图形芯片 0 和图形芯片 1 中的内存数据都是完全一样的。

在 SLI 系统中运行应用程序时，由驱动程序决定运行何种模式：兼容模式、可变*轮流？*帧渲染器模式（AFR）、或分割帧渲染器模式（SFR）。

兼容模式仅仅用在单个图形芯片进行渲染时（也就是说，第二个图形芯片一直处于闲置状态）。在这种模式下，不会有任何性能提高，但是能够确保应用程序可以在 SLI 模式运行。

在 AFR 模式下，驱动程序在图形芯片 0（GPU 0）上渲染 n 帧，在图形芯片 1（GPU 1）上渲染 n+1 帧，在 GPU 0 上渲染 n+2 帧，依此类推。只要每个都是独立帧（即每帧图像没有相同数据），AFR 这时效率最高，这是因为包括逐顶点，光栅化和逐像素在内的所有渲染都要在图形芯片之间平均分割，如果在帧之间包含有共同数据（例如，重新使用先前渲染至纹理的数据），这些数据就必须在图形芯片间传输，而这种数据传输导致了部分通讯损耗，所以无法真正达到 2 倍的增速。

对于 SFR，驱动程序将每帧图像的顶部部分分配给 GPU 0，而将底部部分分配给 GPU 1，并且在每帧的顶部和底部部分间寻求平衡量：如果由于顶部渲染量少于底部渲染量时，GPU 0 这时没有被充分利用，那么驱动程序就把顶部部分分割得比底部大一些，从而使两个图形芯片都充分发挥作用。由于 GPU 0 和 GPU 1 可以分别处理场景的顶部和底部图像，故而可以避免在两个图形芯片上处理所有顶点图像。

SFR 模式也需要数据共享，例如，渲染至纹理操作就是这样。这是因为 AFR 在一般情况下通讯损耗比较少，而且在顶点负载分配的平衡性

方面比 SFR 更为理想。所以 AFR 模式是优先选用的模式。然而，有些时候却不能使用 AFR 模式。比如，如果应用程序将帧缓存的最大数量限制为小于两个时，就不能使用 AFR 模式。

目前开发的驱动程序（66.93 以上的版本）默认使用兼容模式。现在应用程序开发人员可以在驱动程序中创建一个特殊文件，比如，在显示驱动控制面板中，点击 GeForce 选项，选择“显示性能和质量设置（Performance & Quality Settings）”选项，点击“添加文件（Add Profile）”输入应用程序执行文件的名称，然后启用“显示高级设置”选项，并选中启用“Multi-GPU”。

以后的驱动程序将会加入更多方便的控制面板和应用编程接口（API）选项，以便设置各种不同的 SLI 模式。

下面列出了在 SLI 系统中取得最佳性能的一些注意事项。

7.2. 避免 CPU 成为系统瓶颈

如果您的应用程序对 CPU 的依赖性很强，那么使用功能强大的显卡并不能起到多大的作用。要使用多显示芯片（multi-GPU）功能，就必须避免 CPU 成为其瓶颈。本编程指南中的第 2 章讲述了如何检测和避免 CPU 成为系统瓶颈的事宜。

同样，如果您将帧速率手动设定为一个任意的固定值，那么帧速率就不会超过该设定值，所以要避免出现这种情况。

7.3. 默认关闭 VSync

启用 VSync 将人为把帧速率降低到显示器的刷新速率。由于多图形芯片具有更好的图形显示性能，所以，多图形芯片设定达到的帧速率可能会比显示器的刷新速率要快的“多”。如果关闭 VSync，就可以充分发挥多图形芯片设定的帧速率。

三倍缓存并不是获取更高帧速率的解决方法：三倍缓存仅仅分配一个额外的后台缓存以供图形适配器进行渲染。

有了三倍缓存，图形适配器就能不间断地渲染多达 3 个缓存。但是如果图形适配器始终以比显示器刷新率更高的速率完成渲染，那么后台缓存的数量就变没什么意义了，显示器刷新率就将继续限制整体帧速率。

更为糟糕的是，三倍缓存还有两个致命弱点：

1. 三倍缓存消耗更多的视频内存，（在屏幕分辨率较高和启用反锯齿功能的情况下，额外消耗的视频内存数量将十分重要）
2. 由于大量的图像帧处于运行渲染命令和在屏幕上显示之间的过程内，所以三倍缓存会增加延迟。

所以，默认关闭 VSync 是唯一可能的解决方案。要在 DirectX 内关闭 VSync，在调用 IDirect3D9::CreateDevice()时，D3DPRESENT_PARAMETERS 结构必须设置为 D3DPRESENT_INTERVAL_IMMEDIATE。

7.4. 延迟限制在最小 2 帧内

DirectX 允许驱动程序有最大 3 帧的缓存，使用缓存帧能够保证 CPU 和 GPU 相互独立运行，达到最佳性能。另一方面，缓存帧越多，从发出命令到在屏幕上显示花的时间就越多。一般都不希望有显示延迟，因为人眼能够感觉到的延迟时间大约为 30 毫秒（视测试场景而定）。

一些游戏可以人为设置缓存帧数，例如，锁定后台缓存大小，强制设置 CPU 和 GPU 的硬件同步性。要锁定后台缓存，首先要停止运行 CPU，清除所有缓存，然后再停止运行 GPU，最后将所有系统锁定于闲置状态，这样缓存帧数就为零了。

通过这种方式终止运行系统对系统性能有很大的损伤，应尽量避免，尤其是在多图形芯片设定中更是如此。

另一种比较好的方法就是在每帧结束的命令行（比如 DirectX 事件查询）中插入标记。然后在运行额外的渲染命令之前，检查这些标记的消耗情况，以限制缓存帧的数量，这样延迟就会限制在 1 到 3 帧之间。

多图形芯片系统对限制缓存帧的数量特别敏感。一般来讲，在具有 n 个图形芯片的系统中，至少需要 n 个缓存帧，才能实现最佳性能。

但是奇怪的是，这并没有增加延迟，因为双图形芯片通常的运行速度是单图形芯片的两倍。比如，在双图形芯片系统（dual GPU system）中缓存 2 帧（每帧渲染需要 15 毫秒）与在单图形芯片系统中缓存 1 帧（渲染需要 30 毫秒），延迟同样都是 30 毫秒。

如果您必须把缓存帧数量限定为 GPU 的最低数量，建议您检查运行程序的 GPU 的具体数量。nvCPL.dll 应用编程接口（API）提供了查询模式，可以查询出 SLI 模式中当前使用的图形芯片的数量，尤其是在 NvCplGetDataInt(NVCPL_API_NUMBER_OF_SLI_GPUS, &number) 中，该函数可以返回系统启用 SLI 功能后的图形芯片数量。文件 NVControlPanel_API.pdf 和 NVSDK 实例文件 NvCpl 有详细的描述。

7.5. 在所有帧中完全更新所有渲染目标纹理

多图形芯片系统的效率与图形芯片共享数据的多少成反比。理想条件下，就是 GPU 完全不共享数据，也就没有了同步消耗，这样系统效率就能发挥到最佳状态。

要将数据共享量减少到最低，那么每个渲染帧都应与前面的所有帧相互独立，特别是使用渲染至纹理技术时，所有使用的渲染目标纹理的都适用于同样的一帧。然而，应避免每隔一帧更新渲染目标，应该每一帧使用相同的渲染目标作为纹理。

如果确实需要跳过渲染目标更新，才能加快单个图形芯片系统的渲染速度，那么可以修改多图形芯片的逻辑算法。比如，测试应用程序是否使用了多图形芯片系统，如果是，则在每一帧都需要更新渲染目标（提高视觉真实度），或者在同一行内每跳过两帧就更新两帧的渲染目标。

还有一种方法可供选择，先渲染后使用渲染效果在 SLI 系统也有效，它可以避免一个图形芯片等待另一个 GPU 的渲染操作后才能进行工作。

7.6. 清除渲染目标和帧缓存

由于硬件无法得知应用程序在渲染目标中是否会更新每个像素，如果渲染目标在使用前没有清除掉，渲染结果就必须在多图形芯片系统中的

GPU 间进行共享。所以，在使用之前清除渲染目标可以让驱动程序和硬件知道不需要进行同步操作。

7.7. 在 D3DPOOL-MANAGED 中分配顶点缓存

多图形芯片系统在 GPU 之间共享顶点缓存，在 Directx 系统中 D3DPOOL_MANAGED 内分配的顶点缓存，与在 D3DPOOL_DEFAULT 内分配缓存相比，减少了相关联的传输损耗，这种损耗的减少对动态顶点缓存最有效，对部分更新的动态顶点缓存尤其有效。

Chapter 8.第 8 章
立体游戏开发

本章将讲述 NVIDIA 立体渲染的工作原理，以及如何在您的应用程序中充分发挥其优势。

8.1. 为何关注立体？

游戏开发商在不断追求游戏逼真度的同时，常常会忽视一个重要因素：在真实世界中，人们是通过双眼来观察事物的。虽然模拟立体成像（在显示屏上而非真实世界）并不是一个巨大的市场，但许多玩家非常喜欢便宜的立体眼镜及 NVIDIA 的立体驱动程序为游戏带来的特殊现场感受。

另外，在游戏开发过程中从三维立体角度来观察这个游戏会给您带来很多好处。您能立即发现看起来不真实的事物。别忘了动态视差会体现出类似的立体视觉效果，但如果使用者移动的话，立体镜会立刻感知他们看到的东西，并通过动态视差获取深度信息。

在开发游戏时使用立体成像是极具竞争优势的；您可以在视觉缺陷嵌入游戏以前将其更正。这当然也会给玩立体游戏的人带来全新的视觉体验。

8.2. 立体工作原理

NVIDIA 3D 立体驱动程序支持基于 DirectX 和 OpenGL 游戏的全屏立体成像。3D 立体驱动支持红与蓝互补渲染以及适用于立体眼镜的翻页成像。通过兼容的硬件，您会看到具有深度感的图像。请注意立体驱动程序版本必须与显示驱动程序版本匹配才能体现出效果。

当启用立体驱动程序进行 3D 游戏时，场景通过两个视觉点进行渲染；这两个视觉点分别模拟真实的视觉点，就像人的左眼和右眼。它与固定功能渲染及顶点着色器同时工作。

为达到更准确的立体效果，开发人员需要考虑以下事宜。

8.3. 影响立体效果的负面因素

这里我们列出了通常会对立体效果有负面影响的因素，并提供了相应对策。

8.3.1. 错误深度的渲染

这是您首先应当关注的问题。3D 立体驱动程序通过深度来获得立体效果，因此，如果对象的深度不正确的话，透过立体成像后的效果就会与场景格格不入。

- 将背景图像、天空盒、天空顶放于尽量远的深度。否则，三维世界看起来就会像在一个小盒子里。
- 将物品界面（HUD）项目置于适当的 3D 深度。如果您有名称标签悬浮在对象之上的话，将他们置于这个对象的 3D 深度，这样显示的立体效果比将他们放于视觉截面附近的平面的深度要好。
- 尽量将 HUD 渲染到场景的深远位置，该技巧对立体效果的体现同样很有帮助。这样您就能在场景的其他部分获的更好的深度效果，同时在注视 HUD 时不会引起视觉疲劳。
- 激光瞄准镜，十字准线和指针在 3D 世界里只有置于他们指向的对象的深度时看起来才会正确。当深度不正确时，人的视线聚焦在一

个深度，而指针却在另一深度，这几乎不可能使用；使用者会看到两个指针，而它们都没有指向正确的位置。

- 应该在对象本身的深度进行高亮处理，而不是在屏幕空间中。

8.3.2. 广告牌效果

在常规的 3D 中，广告牌效果看起来平淡低劣，而在立体场景中效果甚至更差。在常规的 3D 中，当您移动时仍能看见广告牌。但是在立体场景中，即使场景是静态的，您也会马上出现问题。大多数广告牌在立体场景中看起来都特别平淡，所以尽量用真实几何图形进行代替，即使是低分辨率的几何图形看起来效果也会更好。

对于具有粒子效果的广告牌（火花、烟雾、灰尘等），其效果也有差异。您最好在立体场景中测试您的应用程序，测试其视觉效果以判断其质量是否足够好，并确保这些广告牌在 3D 中有实际意义的深度。

8.3.3. 后期处理与屏幕空间效果

2D 屏幕空间效果会极大损害立体效果。像模糊发光、bloom 过滤器、基于图像的动态模糊都属于这一类。这类效果是在把 3D 几何图形进行质地渲染，然后把 2D 屏幕方格列进行屏幕渲染时产生的。具有质地的几何图形的深度就不再是立体效果所应有的深度，因此在 3D 立体里效果不佳。

您应当为在立体里游戏的玩家提供关闭这些效果的选项，并将几何图形渲染至后台缓存。

8.3.4. 在 3D 场景中使用 2D 渲染

任何作为 2D 进行渲染的对象并不真正拥有 3D 深度，因此将其置于屏幕深度。这在 3D 立体场景中看起来会非常平淡。如果您在 HUD 中混有 2D 和 3D 的话，就会导致深度不一致而引起视觉疲劳。再次强调，在 3D 中把一切对象都置于合适的深度，然后启用立体效果进行测试。

8.3.5. 子视角渲染

当把比如画中画显示，车镜，或屏幕上角的小地图这样的子视角渲染至屏幕时，您必须在渲染前设定视口覆盖这部分区域。这样可以避免特殊的立体效果与屏幕的其他区域混淆。

8.3.6. 用复杂方块更新屏幕

如果您想只改变屏幕部分内容而不更新屏幕的其他部分，这可能导致立体场景中产生看起来很奇怪的渲染。仅仅渲染每一帧中所有可视的对象即可。

8.3.7. 用大量的分隔来解决碰撞

如果您想通过将对象彼此分隔解决碰撞问题的话，请确保不要将他们分隔得太远。在普通 3D 中如果移动这看起来效果很差，在立体中会马上显示出来，这会使对象看起来好象悬浮在地面之上。

8.3.8. 改变场景中不同对象的深度段

将场景分割成多个深度段会导致立体效果出现扭曲，某些对象看起来会变短或拉长。为达到最好的立体效果，所有对象都应在协调一致的深度段进行渲染。

8.3.9. 不提供顶点深度数据

在为软件转换和光照发送顶点数据以进行 D3D 渲染时，应包含 RHW 深度信息，以实现立体效果。

8.3.10. 在窗口模式下渲染

只有当您的应用程序在全屏模式下，NVIDIA 的 3D 立体才能正常工作。如果您的应用程序不支持全屏模式，那么玩家将无法体验 3D 立体效果。

8.3.11. 阴影

在立体中使用全屏阴影色格渲染模板阴影将不能正常工作。但在立体中对场景里有阴影的对象以阴影色适当的深度进行再渲染则工作正常。阴影帖图正常，而只要阴影投射的深度合适，投射阴影也正常。

8.3.12. 软件渲染

NVIDIA 3D 立体驱动程序自动支持 DirectX 和 OpenGL。如果您使用其他应用编程接口来渲染 3D 几何图形的话，在立体中将无法渲染。

8.3.13. 手动写入渲染目标

不要锁定渲染目标，也不要直接写入；这样做会绕过立体驱动程序。

8.3.14. 很暗或很高的对比度场景

在使用 3D 立体眼镜后，很暗的场景会变得更暗。提供亮度或伽玛（gamma）调节将有助于解决这一问题。在极亮或极暗的对象上使用极亮的对象可能会产生重影，从而损害立体效果。在立体场景中测试您的游戏可以迅速判断是否存在这个问题。

8.3.15. 顶点间有小空隙的对象

网眼的小空隙在立体渲染时会变得更明显。请注意使您的网眼紧密，并且在立体场景中测试，以避免这一问题。

8.4. 改善立体效果

以下建议将有助于您制作出具有超群效果的立体游戏。

8.4.1. 在立体中测试您的游戏

获得极佳 3D立体效果的最佳方法就是在立体场景内测试您的游戏。这样，很多问题都会凸现出来，而且能被轻松修复。您可通过 IODisplay(http://www.i-glasses.com)获得低成本的立体工具包。NVIDIA 3D立体驱动程序同样支持红与蓝互补立体模式，如果您身边有一副纸质 3D眼镜的话，测试起来就更容易了。

8.4.2. 获得“飞出屏幕”的效果

您可以在自己的游戏中设计一些非常接近最近平面的对象，但是不要与视截面的边沿相交。这种设计能让您得到强烈的“飞出屏幕”的立体效

果。悬浮球体、太空飞船、飞行的人物等都很有可能产生这种效果。在立体场景中，这些看起来都非常真切、奇妙。

8.4.3. 使用细致的几何图形

为了使在 3D 中的对象更逼真，您可以使用更多的多边形。当然，这个策略永远都是正确的，在立体中也不例外。尽可能在所有地方都使用多边形——例如建筑物、植物、树木、人物...

8.4.4. 提供交替视角

让用户能够在您的游戏中选择视觉点，或者至少能控制视角。第一人称视角、第三人称视角、俯视视角等，总有某个视角在立体中的效果会更好一些。

8.4.5. 查找当前游戏中的问题

一旦安装了 3D 立体驱动程序，您可以查看它的控制面板，并打开“立体游戏设置”选项。在这个页面上，您可以看到自己的游戏是否被列出和我们已经发现的问题。您的 NVIDIA 客户关系联络人也能帮助您解决问题。

8.5. 立体应用编程接口

为了更好地支持立体效果，目前，我们正在开发两个单独的应用编程接口：

- **立体 BLT 应用编程接口—— 以 3D 显示预渲染立体图像**
  当启用立体显示时，允许显示预创建的左/右图像。
- **I 立体应用编程接口——实时控制立体渲染**
  供您查询和控制立体驱动程序的聚焦点、立体分离度设置和其他细节。这个应用编程接口由一套标题文件和库文件组成，您能使用它在自己的游戏中实时控制这些设置。这些修改能立即生效并且能在每帧中改变。

- **OpenGL 四组缓冲立体功能。**这可用于 NVIDIA Quadro 图形芯片系列中，它不需要特殊的立体驱动程序，并且也可以在窗口模式下工作。

8.6. 更多信息

如需查询更多详情，请与您的NVIDIA客户关系代表联系，或发送电子邮件至3DStereoDev@nvidia.com，了解更多信息或立体应用程序接口的最新资料。

另外，您还可以通过以下网址在线查询最新的立体信息：
http://developer.nvidia.com/object/3D_Stereoscopic_Dev.html

Chapter 9.第 9 章
性能工具概览

本章将描述几个能帮助您识别和解决性能瓶颈的工具。

9.1. NVPerfHUD

NVPerfHUD 能提供 4 个信息图表，均显示在 DirectX 应用程序上方。这些信息图表显示了应用程序的重要统计数据，这些数据能帮助您找出潜在的瓶颈。这些信息图表以心率监护仪格式显示示例数据。从右向左滚动，您就能看到最近的 256 帧数值。

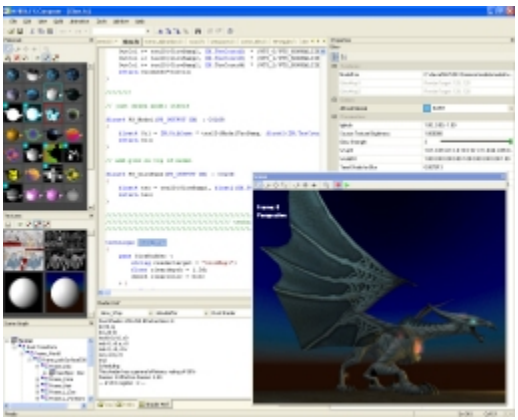
您可以在NVIDIA开发人员网站上找到NVPerfHUD：http://developer.nvidia.com/object/nvperfhud_home.html.

9.2. NVShaderPerf

NVShaderPerf 命令行工具所采用的技术与 FX 合成器（FX Composer）中的着色器性能面板（Shader Perf panel）技术相同。它支持以 HLSL、GLSL、Cg、!!FP1.0、!!ARBfp1.0、ps_1_x 以及 ps_2_x 等语言编写的 DirectX 和 OpenGL 着色器。您可以获得着色器在整个 GeForce 6 产品系列和 GeForce FX 图形芯片中的性能表现报告，包括循环计数、寄存器使用率和 GPU 利用率等。

NVShaderPerf的下载地址为：
http://developer.nvidia.com/object/nvshaderperf_home.html。

9.3. FX 合成器



FX 合成器具有独特的实时预览功能和优化功能，开发人员可以在集成式开发环境中开发出性能更高的着色器。设计 FX 合成器的目的是为了帮助编程人员更轻松地开发并优化着色器，同时为电脑美术创作人员提供一个直观的图形化用户界面（GUI），使他们能为特殊场景定制着色器。

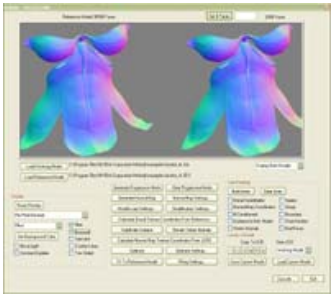
FX 合成器使您能通过高级分析和优化功能对着色器的性能进行微调：

- 允许针对顶点和像素着色器的性能微调工作
- 模拟整个 GeForce 6 系列和 GeForce FX 图形芯片的性能
- 获得纹理查询表的预算功能

- 提供经验性能表现度量，例如，GPU 循环计数、寄存器使用率、利用率和每秒帧数（FPS）
- 优化提示，能指出性能瓶颈

您可以从下列地址下载最新版的FX合成器：
http://developer.nvidia.com/fxcomposer。

9.4. NVIDIA Melody



要创建能让低聚（low-poly）模型看似高聚（high-poly）模型的高画质法线贴图（normal maps），请使用NVIDIA Melody。只需载入低聚工作模型，然后载入高聚参考模型，点击“生成法线贴图”按钮，调节器（Melody）就会既快又好地完成工作。调节器的下载地址为：
http://developer.nvidia.com/object/melody_home.html。

9.5. 开发人员工具问题与反馈

我们期望收到您对这些工具的反馈意见。请将您的看法或关心的问题发送至：sdkfeedback@nvidia.com。