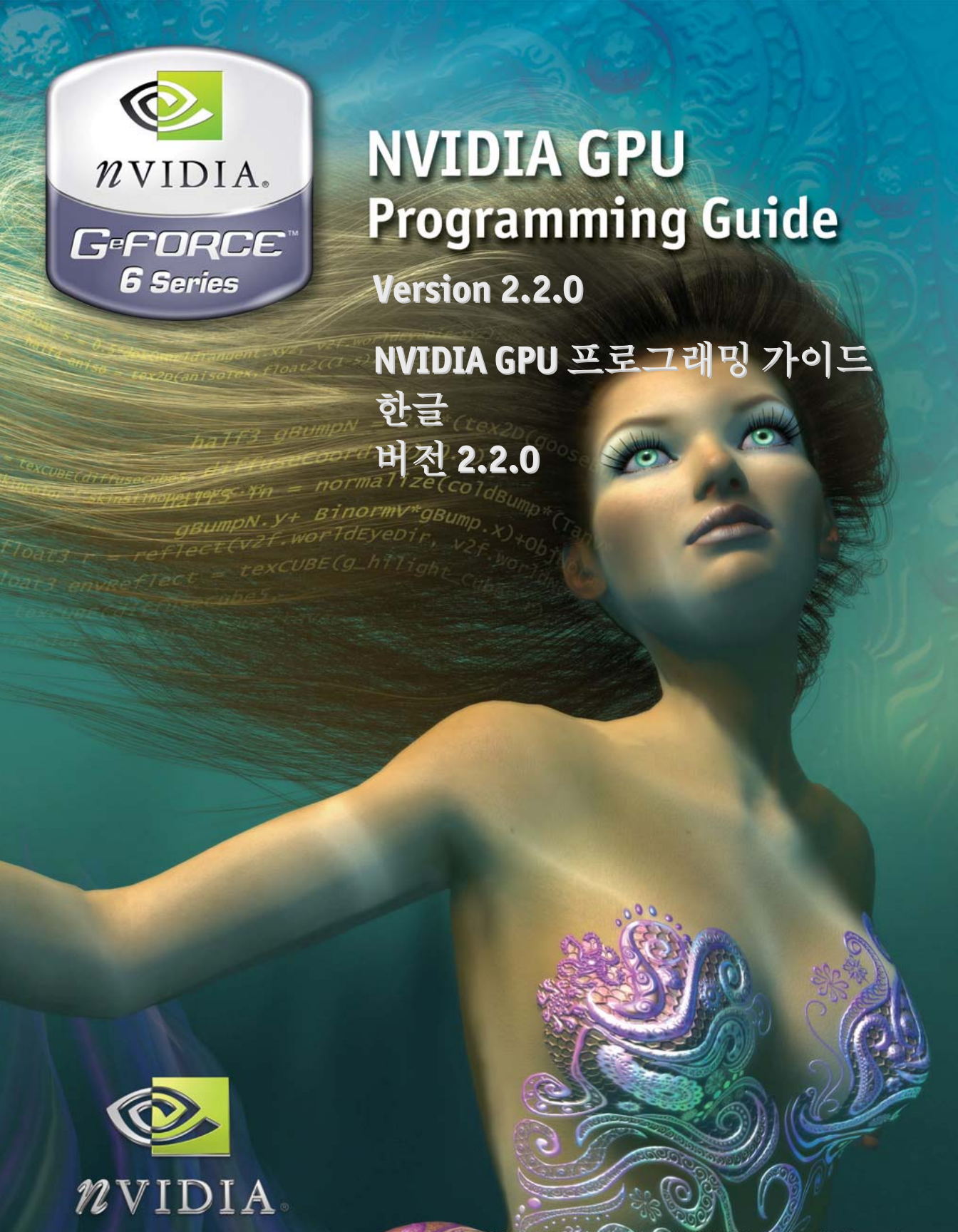

NVIDIA
GeFORCE
6 Series
NVIDIA GPU
Programming Guide
Version 2.2.0
NVIDIA GPU 프로그래밍 가이드
한글
버전 2.2.0
NVIDIA

주요 수정 내역

버전	날짜	변경사항
2.2.0	11/16/2004	일반 맵 형식에 관한 도움말 추가 ps_3_0 성능에 관한 도움말 추가 “일반 도움말” 장 추가
2.1.0	07/20/2004	입체 개발 장 추가
2.0.4	07/15/2004	MRT 항목 업데이트
2.0.3	06/25/2004	멀티 GPU 지원 장 추가
2.0.0	06/01/2004	NV40(GeForce 6 시리즈) 장 추가 “NVIDIA GPU 프로그래밍 설명서”로 제목 변경
1.0.0	07/14/2003	GeForce FX 프로그래밍 설명서

목차

Chapter 1.문서 정보

1.1. 소개

이 설명서는 애플리케이션, 그래픽 API, 그래픽 처리 장치(GPU)에서 최상의 그래픽 성능을 끌어내기 위해 유용하게 사용될 수 있습니다. 이 설명서에 기재된 정보를 이해하면 더 나은 그래픽 애플리케이션을 개발하실 수 있습니다. 도움이나 도움말이 필요하시면 즉시 devsupport@nvidia.com으로 이메일을 보내 주십시오.

이 문서는 다음과 같은 구성으로 되어 있습니다.

- 1 장(이 장)은 문서 내용에 대해 간략하게 소개하고 있습니다.
- 2 장은 흔히 발생하는 장애를 발견하고 해결하여 애플리케이션을 최적화하는 과정을 설명하고 있습니다.
- 3 장은 장애가 확인되었을 때 그 장애를 해결하는 데 도움이 되는 정보를 소개하고 있습니다. 이 정보는 우선 제일 중요한 최적화가 이루어질 수 있도록 분류, 나열되어 있습니다.
- 4 장은 NVIDIA® GeForce™ 6 시리즈와 NV4X 기반 Quadro FX GPU에 대한 유용한 프로그래밍 정보를 소개하고 있습니다. 여기 나온 정보는 기능을 중점적으로 다루지만, 경우에 따라 성능 문제도 해결합니다.
- 5 장은 NVIDIA® GeForce™ FX와 NV3X 기반 Quadro FX GPU에 대한 몇 가지 유용한 프로그래밍 정보를 제공하고 있습니다. 여기에

나온 정보는 기능을 중점적으로 다루지만, 경우에 따라 성능 문제도 해결합니다.

- 6 장은 NVIDIA GPU 에 대한 일반 도움말을 제공하며, 성능, GPU 확인 등과 같이 각기 다양한 주제를 포괄하고 있습니다.
- 7 장은 NVIDIA 의 SLI(Scalable Link Interface) 기술에 관해 설명하고 있습니다. SLI 를 사용하면 멀티 GPU 로 성능을 크게 증대할 수 있습니다.
- 8 장은 입체 게임 지원을 사용하는 방법에 대해여 설명하고 있습니다. 잘 만들어진 스테레오 게임은 스테레오가 아닌 게임보다 역동적이고 시각적인 면에서 훨씬 집중하기 쉽습니다.
- 9 장은 NVIDIA 의 성능 툴에 대해 간략하게 설명하고 있습니다.
- 10 장은 내부 코드명과 공식적인 제품명으로 당사의 GPU 를 기재하여 쉽게 찾아 보실 수 있도록 도와 드립니다.

1.2. 의견 보내기

본 문서에 관한 의견이나 제안사항이 있으시면 devsupport@nvidia.com으로 보내주십시오.

Chapter 2.
애플리케이션 최적화 방법

이 장에서는 그래픽 애플리케이션의 성능 장애를 발견하고 제거하는 일반적인 절차를 간략하게 설명해 드립니다.

2.1. 정확하게 측정하기

여러 가지 편리한 툴을 사용하면 성능을 측정할 수 있을 뿐 아니라, 검사 절차를 거친 믿을 수 있는 성능 표시기로 사용할 수 있습니다. 예를 들어, NVPerfHUD의 노란 선(자세한 내용은 NVPerfHUD 문서 참조)은 프레임당 총 밀리초(ms)를 측정하고 현재 프레임 속도를 표시합니다..

성능 비교의 유효성을 기하려면 다음 사항을 준수하십시오.

- **애플리케이션이 원활하게 작동하고 있는지 검증합니다.** 예를 들어, 애플리케이션이 DiretX Debug 실행 시간에 맞춰 작동하면, 오류나 경고가 발생하지 않습니다.
- **테스트 환경이 유효한지 확인합니다.** 다시 말해, 애플리케이션 릴리즈 버전과 해당 DLL, 그리고 DirextX 최신 버전의 릴리즈 실행 시간을 작동하고 있는지 확인합니다.
- **모든 소프트웨어에 릴리즈 버전(디버그 빌드 아님)을 사용합니다.**
- **모든 표시 설정을 정확하게 했는지 확인합니다.** 보통, 이것은 설정이 기본값으로 되어 있다는 것을 의미합니다. 이방성 필터링과 안티앨리어싱 설정은 성능에 특히 영향을 미칩니다.

- **수직 동기를 해제합니다.** 이렇게 하면 모니터의 재생 속도로 인해 프레임 속도가 제한되지 않습니다.
- **대상 하드웨어에서 구동합니다.** 특정 하드웨어 구성이 제대로 될지 알고 싶다면, 올바른 CPU, GPU 에서 작동하고 시스템 메모리를 적당량으로 합니다. 낮은 사양 시스템에서 높은 사양 시스템으로 이동하면 병목현상는 크게 달라질 수 있습니다.

2.2. 병목현상 찾기

2.2.1. 병목현상에 대한 이해

지금 이 순간, 성능 저하를 나타내는 상황을 알게 되었다고 가정해 봅시다. 그러면 이제 성능 병목현상을 찾아야 합니다. 병목현상은 대체로 화면 내용에 따라 옮겨 다닙니다. 설상가상으로 단 하나의 프레임에서 자주 옮겨 다닙니다. 그래서 "병목현상을 찾는다"는 것은 시나리오에서 가장 제한되게하는 장애를 찾아보자"는 것을 의미합니다. 병목현상을 제거하면 성능을 크게 높일 수 있습니다.

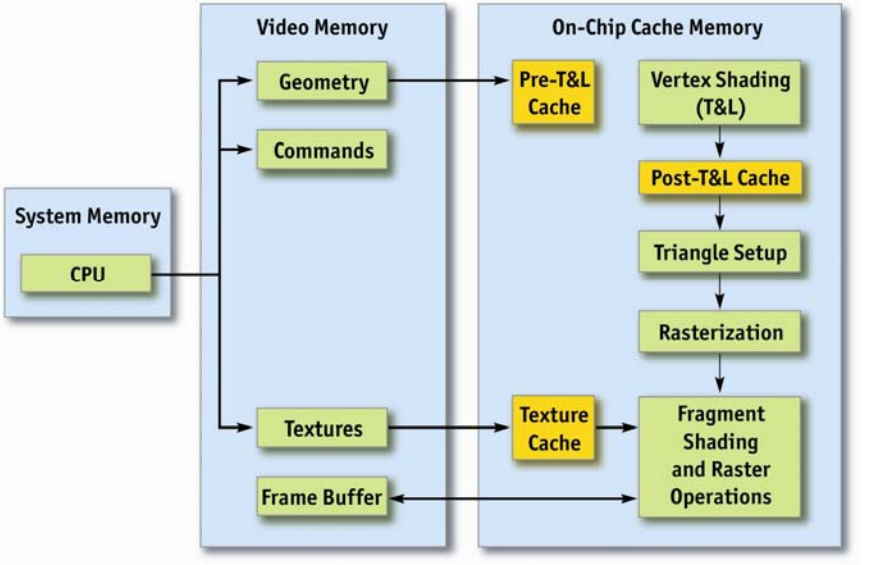


그림 1. 잠재적인 병목현상

CPU, AGP 버스, GPU 수송 단계가 모두 동일하게 로딩되는그림 1 참조) 이상적인 경우라면 어떠한 병목현상도 없을 것입니다. 하지만 불행하게도 그런 이상적인 애플리케이션을 만드는 것은 불가능합니다. 실제로는 어떤 것이 원인이 되어 언제나 성능이 떨어집니다.

병목현상은 CPU 나 GPU 에 존재할 수도 있습니다. NVPerfHUD 의 녹색선(NVPerfHUD 에 대한 자세한 내용은 9.1 참조)은 GPU 가 유휴 시간 동안 몇 밀리초 동안 쉬고 있었는지 알려줍니다. GPU 가 프레임당 단 1 밀리초라도 작동하지 않는다면, 이것은 애플리케이션이 최소한 부분적으로 CPU 제한적이라는 것을 의미합니다. GPU 가 거의 프레임 타임 내내 작동하지 않고 있거나 단 1 밀리초라도 쉬고 있으면, 모든 프레임과 애플리케이션이 CPU 와 GPU 를 동시에 작동시키지 않으며, 그렇게 되면 CPU 가 최대의 병목현상이 됩니다. GPU 성능을 개선하기만 해도 GPU 유휴시간이 늘어납니다.

2.2.2. 기본 테스트

몇 가지 간단한 테스트를 직접 실시하여 애플리케이션의 장애를 찾아낼 수 있습니다. 특수한 툴이나 드라이버가 없어도 되므로, 무엇보다 가장 간편합니다.

- **모든 파일 액세스을 배제합니다.** 하드 디스크 접근이 있으면 프레임 속도가 크게 떨어집니다. 이런 상태는 쉽게 알아차릴 수 있습니다. 그냥 컴퓨터의 "하드디스크 사용 중" 램프를 보거나 Windows의 성능 툴, AMD의 CodeAnalyst(http://www.amd.com/us-en/Processors/DevelopWithAMD/0,,30_2252_3604,00.html) 또는 인텔의 VTune (http://www.intel.com/software/products/vtune/)을 사용하여 디스크 성능 모니터 신호를 보기만 하면 됩니다. 명심할 것은, 특히 애플리케이션이 다량의 메모리를 사용할 경우 메모리를 교환하면서 하드 디스크 액세스가 초래될 수도 있다는 점입니다.
- **CPU 에서는 속도가 다른 동일한 GPU 를 작동합니다.** CPU 속도를 조절할 수 있는, 다시 말해 클럭을 낮출 수 있는 기능을 가진 시스템 BIOS 를 찾는 것이 좋습니다. 왜냐하면, 그렇게 하면 시스템 하나로 테스트를 할 수 있기 때문입니다. 프레임 속도가 CPU 속도에 비례하여 바뀐다면, 애플리케이션은 *CPU* 제한적입니다.

- **GPU 의 코어 클럭을 줄입니다.** Coolbit(제 6 장 참조)와 같이 널리 보급된 유틸리티를 사용해서 할 수 있습니다. 코어 클럭이 느려져서 이에 비례하여 성능이 떨어진다면, 애플리케이션은 버텍스 쉐이더, 레스터화, 또는 프래그먼트 쉐이더의 제한을 받습니다(즉, *쉐이더 제한적*).
- **GPU 의 메모리 클럭을 감소합니다.** Coolbit(제 6 장 참조)와 같이 널리 보급된 유틸리티를 사용해서 할 수 있습니다. 메모리 클럭이 느려져서 성능에 영향을 미친다면, 애플리케이션이 텍스처나 프레임 버퍼 대역폭의 제한을 받습니다(*GPU 대역폭 제한적*).

일반적으로, CPU 속도, GPU 코어 클럭, GPU 메모리 클럭을 바꾸면 CPU 에 장애가 있는지, GPU 에 장애가 있는지 쉽고 빠르게 판단할 수 있습니다. CPU 의 클럭이 *n*% 감소하여 *n*%만큼 성능이 저하되면, 애플리케이션이 CPU 제한적입니다. GPU 의 코어와 메모리 클럭이 *n*% 감소하여 성능이 *n*%만큼 저하되면, 애플리케이션이 GPU 제한적입니다.

2.2.3. NVPerfHUD 사용하기

CPU 장애를 검증하려면 *Null Hardware*(Null HW)라는 특수 드라이버로 애플리케이션을 구동합니다. 이 드라이버는 일반 드라이버처럼 작동하지만(일반 드라이버의 모든 동일한 코드 경로를 통과합니다), 단 한가지 실제로 GPU 에 어떤 작업도 넘기지 않는 점이 다릅니다. *Null HW 드라이버의 성능이 일반 드라이버보다 나을 것이 없다면, 애플리케이션은 완전히 CPU 위주입니다.*

NVPerfHUD 2.0 은 애플리케이션의 모든 그리기 요청(draw call)을 압박해서 Null Hardware Driver 를 모방하는 특수한 모드를 갖고 있습니다. 그러나 모든 그리기 요청을 생략하면 CPU 부하량이 줄어듭니다. 드라이버는 각 그리기 요청에 앞서 일어난 상태 변화를 더 이상 처리하고 수행하지 않기 때문입니다. 또한 NVPerfHUD 는 성능 문제를 찾아내는 데 도움을 주는 유용한 기능이 다양하게 있습니다.

NVPerfHUD 에서 GPU 가 전혀 쉬지 않는 것이 보이면, 애플리케이션은 GPU 제한적입니다. NVPerfHUD 의 파란 선은 드라이버가 GPU 를 몇 밀리초 동안 기다리고 있는지를 나타내므로, GPU 위주의 성능을 검증할 수 있습니다.

NVPerfHUD 사용 설명서에는 장애를 발견하고 제거하며 문제를 해결하는 등의 상세한 방법이 포함되어 있습니다. 이것은 http://developer.nvidia.com/object/nvperfhud_home.html에서 확인할 수 있습니다.

2.3. 병목현상: CPU

애플리케이션이 CPU 위주일 경우, 프로파일링을 사용하여 CPU 시간을 차지하는 것이 무엇인지 찾아내십시오. 다음 모듈은 일반적으로 CPU 시간이 상당히 많이 소요됩니다.

- 애플리케이션(*관련 DLL 만큼 제대로 실행*)
- 드라이버(*nv4disp.dll, nvoglnt.dll*)
- DirectX 실행 시간(*d3d9.dll*)
- DirectX 하드웨어 추상 계층(*hal32.dll*)

이 단계의 목적은 CPU 가 더 이상 장애가 되지 않도록 CPU 오버헤드를 감소하는 것이기 때문에, 무엇이 대부분의 CPU 시간을 차지하는가가 비교적 중요합니다. 이때 평범한 도움말을 할 수 있습니다. 바로, 사소한 최적화 대신 알고리즘 개선을 선택하라는 것입니다. 그리고 CPU 를 가장 많이 차지하는 것을 찾아내면 성능을 최대한 높일 수 있습니다.

그런 다음, 애플리케이션 코드를 조사해서 코드 모듈을 제거하거나 줄이는 것이 가능한지 알아봐야 합니다. 애플리케이션이 `hal32.dll`, `d3d9.dll` 즉, `nvoglnt.dll`에서 CPU를 상당량 사용하고 있다면, API 남용을 표시할 수도 있습니다. 드라이버가 CPU를 상당 부분 차지하고 있다면, 드라이버에 전달되는 요청의 수를 줄이는 것이 가능합니까? 배치(batch) 크기를 개선하면 드라이버 요청을 줄이는 데 도움이 됩니다. 일괄 처리에 관한 자세한 내용은 다음 프리젠테이션에서 확인할 수 있습니다.

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

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

NVPerfHUD 또한 드라이버 오버헤드를 찾아내는 데 도움이 됩니다. 프레임당 드라이버에 소요된 시간의 양을 표시할 수 있고(빨간 선으로 나타냄), 프레임마다 그려진 배치의 수를 그래프로 나타냅니다.

성능이 CPU 위주일 때 점검해야 하는 다른 영역은 다음과 같습니다.

- **애플리케이션이 프레임 버퍼나 텍스처와 같은 자원을 Lock 합니까?** 자원을 Lock 하면 CPU 와 GPU 를 순차화하고, 결과적으로 GPU 가 장치를 다시 작동할 준비가 될 때까지 CPU 를 중지시킵니다. 그래서 CPU 는 적극적으로 대기 상태가 되고, 애플리케이션 코드를 처리할 수 없습니다. 따라서 lock 이되면 CPU 오버헤드가 초래됩니다.
- **애플리케이션은 CPU 를 사용하여 GPU 를보호합니까?** 삼각형을 조금 모으면 CPU 작업이 만들어지고 GPU 의 작업이 줄지만, GPU 는 이미 유휴 상태입니다! CPU 위주로 돌아갈 때 CPU 측 최적화를 제거하면 성능이 실제로 증가합니다.
- **CPU 작업을 GPU 에 넘겨주는 것을 고려합니다.** GPU 의 버텍스나 픽셀 처리장치에 들어맞도록 알고리즘을 다시 작성할 수 있습니까?
- **쉐이더를 사용하여 배치 크기를 늘리고 드라이버 오버헤드를 줄입니다.** 예를 들어, 2 개 배치를 자체 쉐이더로 그리지 않고 2 개 자료를 한 쉐이더에 합쳐서 그 기하학적 구조를 하나의 배치로 그릴 수도 있습니다. Shader Model 3.0 은 배치 여러 개를 하나로 붕괴시키는 다양한 상황에서 유용할 수 있으며, 배치와 draw overhead 모두 줄일 수 있습니다. Shader Model 3.0 에 대한 자세한 내용은 4.1 항을 참조하십시오.

2.4. 병목현상: GPU

GPU는 파이프라인으로 정교하게 연결된 아키텍처입니다. GPU가 병목현상 이라면, 어떤 파이프라인 단계가 최대의 장애인지 알아내야 합니다. 그래픽 파이프라인의 다양한 단계에 대한 간략한 설명은 http://developer.nvidia.com/docs/IO/4449/SUPP/GDC2003_PipelinePerformance.ppt를 참조하십시오.

NVPerfHUD 는 다양한 GPU 와 드라이버 기능을 강제로 작동하거나 해제하게 만들어서 작업을 단순화합니다. 예를 들어, 밉맵 LOD 바이어스를 동원하여 모든 텍스처를 2 × 2 로 만들 수 있습니다. 성능이 크게 좋아진다면, 텍스처 캐시 실패가 병목현상입니다. 이와

마찬가지로 NVPerfHUD 는 쉐이더의 전체 또는 일부를 강제로 단일 주기에서 작동하게 함으로써 픽셀 쉐이더 실행 시간에 대한 통제를 허용합니다.

GPU가 애플리케이션의 병목현상이라는 판단이 들면, 3 장에 나온 정보를 사용하여 성능을 개선하십시오.

Chapter 3.
일반 GPU 성능 정보

이 장에서는 GeForceFX 와 GeForce 6 시리즈 GPU 에서 최적의 성능을 구현하 데 도움이 될 만한 매우 유용한 정보를 소개하고 있습니다. 여러분이 찾아보기 편하도록 파이프라인 단계에 따라 정보를 정리해 놓았습니다. 각 하위 항의 정보는 중요도 순서로 정리되어 있습니다. 그러므로 우선 어느 부분에서 노력을 집중해야 하는지 알 수 있을 것입니다.

GPU 파이프라인 성능에 대해 간단하게 알고 싶다면, 라는 *GPU 의 핵심: 실시간 그래픽을 위한 프로그래밍 기법, 정보, 요령(Programming Techniques, Tips, and Tricks for Real-Time Graphics)*이란 책의 *그래픽 파이프라인 성능* 챕터를 참조하십시오. 그래픽 파이프라인에서 나타나는 잠재적인 성능 문제를 어떻게 처리해야 하는지 방법은 물론 장애 확인에 관한 내용이 들어 있습니다.

그래픽 파이프 라인 성능은 http://developer.nvidia.com/object/gpu_gems_samples.html에서 자유롭게 확인할 수 있습니다.

3.1. 유용한 정보 모음

GeForceFX 와 GeForce 6 시리즈 GPU 는 제대로 사용하면 상당히 높은 수준의 성능을 실현할 수 있습니다. 아래는 성능에 관한 유용한 정보를 간략하게 소개한 것으로, 자세한 내용은 하위 항에서 확인할 수 있습니다.

- **일괄 처리가 미흡하면 CPU 병목현상이 발생합니다**
  - 배치를 적게 사용합니다.
    - 텍스처 아틀라스을 사용하여 텍스처의 상태 변화를 줄입니다. http://developer.nvidia.com/object/nv_texture_tools.html
    - DirectX 의 경우, Instancing API 를 사용하여 SetMatrix, 그리고 이와 유사한 인스턴싱 으로 상태 변화를 줄입니다.
- **버텍스 쉐이더는 GPU 병목현상을 초래합니다**
  - Indexed Primitive Call 을사용합니다.
    - Use DirectX 9 의 메시 최적화 요청[`ID3DXMesh:OptimizeInplace()` 또는 `ID3DXMesh:Optimize()`]을 사용합니다.
    - Indexed Primitive Call 이 효과가 없을 경우 NVTriStrip 유틸리티를 사용합니다. http://developer.nvidia.com/object/nvtristrip_library.html
- **픽셀 쉐이더는 GPU 병목현상을 초래합니다**
  - 작업에 잘 맞는 가장 낮은 픽셀 쉐이더 버전을 선택합니다.
    - 쉐이더를 개발할 때, 더 높은 버전을 사용하는 것은 괜찮습니다. 우선 작동하게 해놓은 다음, 픽셀 쉐이더 버전을 낮춰서 최적화할 수 있는 기회를 노립니다.
  - `ps_2_*` 기능이 필요하다면, `ps_2_a` profile 프로파일을 사용합니다.
  - 작업에 잘 맞는 가장 낮은 데이터 정밀도를 선택합니다.
    - half 에서 float 인 것을 선호합니다 .
  - 될 수 있는 한 모든 것에 이 종류를 사용합니다.
    - 다양한 매개변수
    - 일정한 매개변수
    - 변수
    - 상수
  - 버텍스와 픽셀 쉐이더의 균형을 맞춥니다.

- 픽셀 쉐이더의 지배를 받을 경우 선형화할 수 있는 계산을 버텍스 쉐이더에 넘깁니다.
- 픽셀 쉐이더의 주기 동안 바뀌지 않을 상수에 대해 일정한 매개변수를 사용하지 않습니다.
- 대수학을 사용하여 연산을 줄일 수 있는 기회를 노립니다.
- 복잡한 함수를 텍스처 검색으로 바꿉니다.
    - 픽셀마다 정반사 라이팅
    - 프로그램에서 생성된 텍스처를 파일로 만들기 위해 FXComposer 를 사용합니다.
    - 그러나 `sincos`, `log`, `exp`는 원시 명령이므로, 텍스처 검색으로 바꿀 필요가 없습니다.

- **텍스처링은 GPU 병목현상을 초래합니다**
    - 밉매핑을 사용합니다.
    - 밉맵 및 이방성 필터링을 사용합니다.
        - 이방성 필터링의 수준을 텍스처 복잡성에 맞춥니다.
        - 포토샵 플러그인을 사용하여 이방성 필터링 수준을 다양화하고 모양이 어떻게 되는지 확인합니다. http://developer.nvidia.com/object/nv_texture_tools.html

        간단한 원칙 하나를 지키면 됩니다. 바로, 텍스처에 노이즈가 있으면 이방성 필터링을 작동시키는 것입니다.

- **래스터화는 GPU 병목현상을 초래합니다**
    - 2 배속 Z 전용 및 스텐실 렌더링
    - 초기 Z(Z-cull) 최적화

- **안티앨리어싱(Antialiasing)**
    - 안티앨리어싱을 활용하는 방법

3.2. 일괄 처리

3.2.1. 배치를 적게 사용하라

"일괄 처리"(batching)는 삼각형당 한 번의 API 요청을 사용하지 않고, 수많은 삼각형을 한 번의 API 요청으로 그릴 수 있도록 기하학적 구조를 그룹으로 묶는 것을 가리킵니다. API 요청을 할 때마다 드라이버 오버헤드가 있습니다. 그래서 이 오버헤드를 상환하는 가장 좋은 방법은 되도록 API 를 요청하지 않는 것입니다. 다시 말해, 한번에 몇 천 개의 삼각형을 그림으로써 그리기 요청의 총 수를 줄입니다. 규모가 더 큰 배치의 사용 수를 줄이는 것도 성능을 높이는 좋은 방법입니다. GPU 가 강력해질수록, 최적의 렌더링 속도를 달성하기 위해 효과적인 일괄 처리가 더욱 중요해집니다.

3.3. 버텍스 쉐이더

3.3.1. Indexed Primitive Call 사용

Indexed Primitive Call 을 사용하면 GPU 로 변형과 라이팅을 마친 후의 버텍스 캐시를 활용할 수 있습니다. 버텍스가 이미 변형되어 있으면, 두 번 변형하지 않습니다. 그냥 캐시 결과를 사용합니다.

DirextX의 경우, `ID3DXMesh` class의 `OptimizeInPlace()` 또는 `Optimize()` 함수를 사용하여 매시를 최적화하고 버텍스 캐시에 좀더 우호적으로 만들 수 있습니다.

자체 NVTriStrip 유틸리티를 사용하여 최적화된 캐시 우호적 메시를 형성할 수도 있습니다. NVTriStrip는 스탠드얼론 프로그램으로, http://developer.nvidia.com/object/nvtristrip_library.html에서 구할 수 있습니다.

3.4. 쉐이더

높은 수준의 쉐이딩 언어는 쉐이더 작성을 수월하게 하는 강력하면서 유연한 메커니즘을 제공합니다. 불행하게도, 이것은 *느린* 쉐이더를 작성하는 것이 그 어느 때보다 쉽다는 것을 의미합니다. 주의를 기울이지 않으면 끝내 느린 쉐이더 혼자 서도 애플리케이션이 작동을 멈출 수 있습니다. 다음에 나온 정보는 간단한 효과 때문에 비효율적인 쉐이더를 작성하는 것을 피하도록 도움을 줄 것입니다. 거기다 GPU 의 연산 능력을 100% 활용하는 법을 배우게 될 것입니다. 하이엔드 GeForce FX GPU 는 제대로 사용하면 클럭 주기당 20 개가 넘는 작업을 실현할 수 있습니다! 그리고 최신 GeForce 6 시리즈 GPU 는 몇 배 좋은 성능을 실현할 수 있습니다.

3.4.1. 되도록 버전이 낮은 픽셀 쉐이더 선택

작업을 완수할 수 있는 가장 낮은 픽셀 쉐이더 버전을 선택합니다. 예를 들어, 요소당 8 비트밖에 안 되는 텍스처에서 간단한 텍스처 꺼내기와 블렌드 작업을 하고 있다면, `ps_2_0` 이상의 쉐이더를 사용할 필요가 없습니다.

3.4.2. ps_2_a 프로파일을 사용하여 픽셀 쉐이더 컴파일

마이크로소프트의 HLSL 컴파일러(fxc.exe)는 컴파일을 하고 있는 프로파일에 기준하여 칩 최적화를 추가한 것입니다. GeForce FX GPU를 사용하고 있고 쉐이더에 `ps_2_0` 이상이 필요하다면, `ps_2_a` 프로파일을 사용해야 합니다. `ps_2_a` 프로파일은 GeForce FX 계열에 직접 대응하는 `ps_2_0` 기능의 포함집합입니다. `ps_2_a` 프로파일에 컴파일을 하면 `ps_2_0` 프로파일에 컴파일하는 것보다 성능이 더 좋을 것입니다. `ps_2_a` 프로파일은 2003 년 7 월 HLSL 릴리즈부터 사용할 수 있다는 사실에 주의하십시오.

일반적으로, fxc 의 최신 버전(DirectX 9.0c 이상)을 사용해야 합니다. 마이크로소프트가 더 지능적인 컴파일 기능을 추가하고 각 릴리즈로 버그를 수정할 것이기 때문입니다. GeForce 6 시리즈 GPU 의 경우, 적절한 프로파일과 최신 컴파일러로 컴파일을 하는 것만으로도 충분합니다.

3.4.3. 되도록 데이터 정밀도가 가장 낮은 값을 선택

성능과 품질에 영향을 미치는 또 다른 요소는 작업과 레지스터에 쓰이는 정밀도입니다. GeForce FX와 GeForce 6 시리즈 GPU는 32 비트와 16 비트의 부동 소수점 형식(각각 float와 half라 칭함)과 12 비트의 고정 소수점 형식(fixed라 칭함)을 지원합니다. float형의 데이터는 `s23e8`형식과 더불어 IEEE와 매우 유사합니다. half 또한 IEEE와 유사하며, `s10e5` 형식으로 되어 있습니다. 12 비트의 fixed 유형은 [-2,2]의 범위를 포괄하며, `ps_2_0` 이상의 프로파일에서 사용할 수 없습니다. fixed 유형은 DirextX에서 `ps_1_0~ps_1_4` 프로파일과 함께, OpenGL에서는 `NV_fragment_program` extension 또는 Cg와 함께 사용할 수 있습니다.

이처럼 여러 유형의 성능은 정확성에 따라 다양합니다.

- fixed 유형은 가장 빠르며, 컬러 연산과 같은 정확성이 낮은 계산에 쓰입니다.
- 부동 소수점 정확성을 원하면, half 유형이 flat 유형보다 높은 성능을 제공합니다. half 유형을 제대로 사용하면 프레임 속도를 3 배 높일 수 있으며, 애플리케이션 대부분의 경우 렌더링한 픽셀의 99% 이상을 32 비트 결과의 1 LSB(최하위비트) 이내로 할 수 있습니다!
- 되도록 정확성이 높아야 한다면, float 유형을 사용하십시오..

`/Gpp` 플래그(2003 년 7 월 HLSL 업데이트에서 사용 가능)를 사용하여 쉐이더에 있는 모든 것을 강제로 1/2 정밀도로 할 수 있습니다. 쉐이더를 작동시키고 여기 나온 정보를 그대로 따른 다음, 이 플래그를 가동하여 플래그가 성능과 품질에 영향을 미치는 것을 확인하십시오. 아무런 오류가 발생하지 않으면, 이 플래그를 가동 상태로 두십시오. 그렇지 않으면, 유익할 때마다 1/2 정밀도로 하십시오(`/Gpp`는 작업할 수 있는 상위의 성능 범위를 지원합니다).

half 또는 fixed 유형을 사용할 때, 다양한 매개변수, 일정한 매개변수, 변수, 상수에 사용하는지 확인하십시오. `ps_2_0` 프로파일과 함께 DirectX의 어셈블리 언어를 사용하고 있다면, `_pp` 변경자를 사용하여 계산의 정확성을 낮추십시오.

OpenGL `ARB_fragment_program` 언어를 사용하고 있다면, 실행 시간을 최소화하고 싶을 경우 `ARB_precision_hint_fastest` 옵션을

사용하고 명령 기준으로 정밀도를 조절하고 싶을 경우 `NV_fragment_program` 옵션을 사용합니다(http://www.nvidia.com/dev_content/nvopenglspecs/GL_NV_fragment_program_option.txt 참조).

컬러 기반의 작업은 대부분 정밀도의 손실 없이 fixed 또는 half 데이터 유형으로 수행할 수 있습니다(예: `tex2D*diffuseColor` 작업).

OpenGL 의 GeForce FX 하드웨어에서, 여러분은 부동 소수점 정밀도로 작업(정규화된 벡터의 도트 곱 등)을 함으로써 부동 소수점 작업으로 대부분 구성된 쉐이더의 속도를 높일 수 있습니다.

예를 들어, 표준화의 결과는 컬러와 마찬가지로 1/2 정밀도일 수 있습니다. 위치는 정밀도를 1/2 로 할 수 있지만, 관련한 값이 제로에 가까워질 수 있도록 버텍스 쉐이더로 비율을 조절해야 할 수도 있습니다.

예를 들어, 값을 로컬 탄젠트 공간으로 이동한 다음 위치를 낮추면, 매우 넓은 위치가 1/2 정밀도로 변환될 때 보이는 줄무늬 흔적을 제거할 수 있습니다.

3.4.4. 대수학을 사용하여 연산 줄이기

일단 쉐이더를 실행했으면, 연산을 보고서 수학적 자질을 사용하여 연산을 간단하게할 수 있는지 생각해 보십시오. 이것은 여러 개의 쉐이더에 걸쳐 공유하는 라이브러리 함수에 특히 효과적입니다.

- 일반적인 구체 맵 영사는 다음과 같은 식으로 표현합니다.

  $p = sqrt(Rx^2 + Ry^2 + (Rz + 1)^2)$

  이것을 전개하면 다음과 같습니다.

  $p = sqrt(Rx^2 + Ry^2 + Rz^2 + 2Rz + 1)$

  반사 벡터가 정상화되면(3.4.8 과 3.4.6 항 참조), 처음 세 가지 식의 합계는 1.0 이 됩니다. 그러면 이 식은 다음과 같이 인수로 재분해할 수 있습니다.

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

- 1.414 를 곱한 다음 다른 상수로 포개어(3.4.8 항 참조), 도트 곱을 저장합니다.

- 도트(normalize(N), normalize(L))는 훨씬 더 효율적으로 연산할 수 있습니다.
  - 보통 (N/|N|) dot (L/|L|)로 연산할 수 있습니다. 이때 값비싼 2 가지 rsq(reciprocal square root)가 필요합니다.
  - 대수학을 약간 적용하면 다음과 같은 식이 나옵니다.
  - (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))
  - 이때 한 가지 값비싼 rsq 작업이 필요합니다.

3.4.5. 여러 Interpolant 의 스칼라 요소에 벡터 값을 포함시키지 않음

계산 하나에 너무 많은 정보를 끼워넣으면 컴파일러가 코드 효율성을 최적화하는 것이 어려워질 수 있습니다. 예를 들어, 탄젠트 행렬을 전달하고 있을 경우, 뷰 벡터를 3 q 요소에 포함시키지 마십시오. 이와 같은 실수는 아래 나와 있습니다.

```
// Bad practice
tangent = float4(tangentVec, viewVec.x)
binormal = float4(binormalVec, viewVec.y)
normal = float4(normalVec, viewVec.z)
```

대신에 뷰 벡터를 네 번째 interpolant 에 위치시킵니다.

3.4.6. 지나치게 일반적인 라이브러리 함수를 사용하지 않음

쉐이더 여러 개에 걸쳐 공유하는 함수는 매우 일반적으로 작성할 때가 많습니다. 예를 들어, 반사는 종종 다음과 같이 연산합니다.

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

이런 식으로 작성한 반사 벡터는 일반 벡터나 입사 벡터의 길이와 독립적으로 연산할 수 있습니다. 그러나, 쉐이더 작성자는 라이팅 계산을 수행하기 위해 최소한 일반 벡터가 순서대로 정규화되기를 원할

때가 자주 있습니다. 이런 경우라면, `reflect()`에서 도트 곱, 역곱, 스칼라 곱을 제거할 수 있습니다. 이와 같은 최적화를 통해 성능을 크게 증진시킬 수 있습니다.

3.4.7. 정규화된 벡터의 길이를 계산하지 않음

지나치게 일반적인 라이브러리 함수의 흔한(그리고 값비싼) 예는 입력 벡터의 길이를 계산하는 것입니다. 그러나, 이 벡터는 함수를 호출하기에 앞서 정규화될 때가 많습니다. 컴파일러는 이것을 감지하지 못합니다. 이는 곧, 1.0 을 계산하기 위해 픽셀당 상당한 산술 작업이 수행된다는 것을 의미합니다.

라이브러리 함수가 벡터의 길이에 독립적으로 정확하게 이루어질 경우, 길이를 이 함수의 스칼라 매개변수로 만드는 것을 고려해 보십시오. 그런 식으로, 함수를 호출하기 전에 벡터를 정규화하는 쉐이더는 1.0 이라는 상수값을 전달할 수 있으며(길이를 계산하지 못하는 것으로 인한 모든 이점을 제공함), 벡터를 정규화하지 못하는 쉐이더는 길이를 계산할 수 있습니다.

3.4.8. 일정한 상수식을 포갬

많은 개발자들은 픽셀 쉐이더에 동적 상수를 포함하고 있는 식을 계산합니다. 표현식에 일정한 상수(일정하고 일렬로 된 상수) 또는 가 하나 이상 사용되면, 상수를 함께 포개서 성능을 증대하는 방법이 있습니다. 예를 들면 다음과 같습니다.

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

`g_OverbrightColor` 는 CPU 에서 2.0 을 미리 곱한 다음, 프레임당 수백만 개에 이르는 픽셀에 픽셀당 곱을 저장할 수 있습니다.

가능한 한 많은 상수 표현식을 접기 위해서 수식을 분배하거나 인수로 분해할 필요가 있을 수도 있습니다. 그리고, HLSL 프리쉐이더를 사용하여 쉐이더가 작동하기 전에 CPU 에서 사전 계산을 수행할 수 있습니다.

또 다른 흔한 예를 들자면, 각 버텍스에서 `materialColor * lightColor` 를 연산하는 것입니다. 이 표현식은 주어진 배치의 모든 버텍스에 대해 동일한 값을 갖기 때문에, CPU 에서 계산해야 합니다.

또한 역행렬을 계산해서 GPU 가 아니라 CPU 에서 행과 열을 바꿔야 합니다. 버텍스나 프래그먼트를 기준으로 하지 않고 한번 계산하는 것만 필요하기 때문입니다. `Zpr` (pack row-major)와 `Zpc` (pack column-major) 컴파일러 옵션은 행렬을 원하는 대로 저장하는 데 유용합니다.

3.4.9. 픽셀 쉐이더의 주기 동안 바뀌지 않을 상수에 일정한 매개변수를 사용하지 않음

개발자들은 가끔 일정한 매개변수를 사용하여 0, 1, 255 와 같이 흔히 쓰는 상수를 넣습니다. 이런 관행은 지양해야 합니다. 컴파일러가 상수와 쉐이더 매개변수를 구분하는 것이 어려워서 성능을 떨어뜨리기 때문입니다.

3.4.10. 버텍스 쉐이더와 픽셀 쉐이더의 알맞은 균형 찾기

높은 성능을 실현하는 것은 곧, 장애를 제거하는 것을 일컫습니다. 또한 이는 파이프라인의 모든 요소, 즉 CPU, AGP 버스, 그래픽 파이프라인의 단계를 균형있게 만들어야 한다는 것을 의미합니다. 버텍스 쉐이더나 픽셀 쉐이더를 사용하고자 할 때 다음과 같은 몇 가지 요인을 고려해야 합니다.

- **객체가 어떻게 짜여져 있습니까?** 프레임마다 수백만 개의 버텍스를 사용하고 있을 경우, 버텍스 쉐이더에 가해지는 부하량을 덜고 싶을 수도 있습니다. 이것은 멀티패스 알고리즘을 사용하고 있을 경우 특히 그렇습니다.
- **목표하는 해상도는 얼마나 됩니까?** 애플리케이션이 더 높은 해상도에서 구동되기를 기대한다면, 픽셀 쉐이더가 장애물로 등장할 가능성이 높습니다. 그래서 여러분은 버텍스 쉐이더에 더 많은 연산을 넘기고 싶을 수도 있습니다.
- **픽셀 쉐이더는 얼마나 깁니까?** 복잡한 쉐이딩을 하고 있다면, 아마도 픽셀 쉐이더가 장애물이 될 것입니다. 픽셀 쉐이더가 (평균적으로) 20 주기 넘게 컴파일되어 화면의 절반 이상을 차지하고 있을 경우,

애플리케이션이 GeForce FX 하드웨어에서 픽셀 쉐이더의 지배를 받게 될 것입니다. 그러니, 연산을 버텍스 쉐이더로 옮길 수 있는 기회를 노려 보십시오. (예를 들어 3.4 항 참조하십시오.) NVShaderPerf 툴을 사용하여 쉐이더가 몇 개 주기를 사용하고 있는지 알아낼 수 있습니다. 또한 GeForce 6 시리자와 같은 새로운 하드웨어는 픽셀 쉐이더에 지배되기 전에 복잡한 픽셀 쉐이더를 허용한다는 것을 명심하십시오.

3.4.11. 픽셀 쉐이더의 지배를 받는 경우 선형화할 수 있는 계산을 버텍스 쉐이더에 넘김

래스터라이저는 버텍스당 값을 가져가서 프래그먼트마다 삽입하며, 아울러 원근 수정의 원인이 되기도 합니다. 선형 계산을 버텍스 쉐이더로 옮김으로써, 당신 대신 이것을 미리 처리하는 하드웨어를 활용하십시오. 더 적은 수의 버텍스에 대해 계산하고 삽입된 결과를 픽셀 쉐이더에 수령할 수도 있습니다.

예를 들어, 여러분은 감쇠를 위해 실제 세상 공간에서 빛의 공간으로 이동할 수 있습니다. 혹 범퍼 매핑을 하고 있다면, 입방체 맵(cube map)에 픽셀당 반사를 하고 있지 않는 한, 버텍스마다 탄젠트 공간으로 이동할 수 있습니다.

3.4.12. `mul()` 표준 라이브러리 함수 사용

행렬 곱을 직접 수행하지 않고 `mul()` 표준 라이브러리 함수를 사용하십시오. 그러면, 애플리케이션이 interpolant 에 행렬을 전달할 때 나타날 수 있는 일부 문제를 피할 수 있을 것입니다.

3.4.13. 종속 텍스처 좌표에 `saturate()` 대신 `D3DTADDRESS_CLAMP` (또는 `GL_CLAMP_TO_EDGE`) 사용

`saturate()`를 사용하면 일부 GPU 에 대한 비용이 늘어날 수 있습니다. 고정된 결과를 텍스처 좌표로 사용할 경우, 쉐이더에서 하지 않고 텍스처 좌표를 [0..1] 범위로 고정하는 텍스처 하드웨어의 능력을 사용하는 것이 바람직합니다.

3.4.14. 숫자가 낮은 Interpolant 를 먼저 사용

숫자가 낮은 텍스처 좌표(TEXCOORD 집합)를 먼저 사용하면 성능이 좋아질 것입니다. 처음에 TEXCOORD0 로 시작해서 TEXCOORD1, TEXCOORD2 등 위로 이동합니다.

3.5. 텍스처링

3.5.1. 밉매핑 사용

축소된 텍스처가 "물결치는" 결과를 초래하는 것을 막으려면,애플리케이션에서 언제나 밉매핑을 사용합니다. 이미지 품질을 높이고 텍스처 캐시 행동을 개선할 수 있을 것입니다. 메모리를 불과 33% 사용하고서 이 모든 것을 얻는 것이면 꽤 괜찮은 방법입니다. 특히 3D 텍스처는 밉매핑으로 크게 이점을 얻을 수 있습니다. 밉매핑을 가동하고서 성능이 30~40% 증대되는 것을 우리는 보았습니다.

밉맵을 만들 때에는 점점 더 작은 밉맵을 생성하려고 무턱대고 박스 필터를 사용하지 마십시오. 대신에, 가우스 필터나 미첼 필터를 사용하여 샘플을 더 많이 얻으십시오. 그러면, 더 나은 품질 결과를 얻을 수 있을 것입니다. 그러나 밉맵을 만드는 사전 처리에 조금만 더 시간을 할애하면, 실행하는 중에도 애플리케이션의 모습을 지속적으로 개선할 수 있습니다. 포토샵 플러그인(NVIDIA 텍스처 툴 프로그램 일습의 일부)이 여러분을 위해 빠르게 고품질의 밉맵을 만들 수 있습니다. 프로그램 일습은 http://developer.nvidia.com/object/nv_texture_tools.html에서 볼 수 있습니다.

3.5.2. 밉맵 및 이방성 필터링 사용시 신중할 것

밉맵 및 이방성 필터링은 둘 다 이미지 품질을 개선하는 데 도움이 되지만, 성능 저하를 초래하는 단점이 있습니다. 꼭 필요할 때에만 밉맵 및 이방성 필터링을 사용합니다. 일반적으로, 콘트라스트가 큰 세부묘사가 많은 텍스처에 사용하고 싶을 것입니다. 이방성 필터링의 경우, 텍스처의 방향을 고려하고 싶을 수도 있습니다. 보는 사람 입장에서 텍스처가 기울어져 있다면(예를 들어, 마루 텍스처), 텍스처에

대한 이방성 필터링의 수준을 높이십시오. 텍스처가 여러 개인 표면의 경우에는 각기 다른 레이어에 대한 필터링 수준을 적절하게 조절해야 합니다.

Adobe Photoshop 플러그인은 이방성 필터링의 수준을 결정하는 데 도움이 됩니다. 이 툴을 사용하면 서로 다른 필터링 수준을 시도하여 시각적인 효과를 확인할 수 있습니다. http://developer.nvidia.com/object/nv_texture_tools.html에서 툴을 받아볼 수 있습니다. 귀사의 아티스트들은 어떤 텍스처가 이방성 필터링이나 밉맵 필터링을 필요로 하는지 알기 위해 이 툴의 도움을 원하고 있을 수도 있습니다.

3.5.3. 복잡한 함수를 텍스처 검색으로 바꾸기

텍스처는 복잡한 함수를 부호화하는 멋진 방법입니다. 텍스처가 OTF(on-the-fly) 방식으로 표시할 수 있는 다차원 배열이라고 생각해 보십시오. GeForce FX 계열은 텍스처에 효율적으로 접근할 수 있으며, 비용이 산술적 작업과 같을 때가 많습니다. FX Composer 툴을 사용하면 이런 종류의 최적화를 원형화할 수 있습니다. FX Composer는 http://developer.nvidia.com/FXComposer에서 구할 수 있습니다.

텍스처에서 복잡한 일련의 산술 작업을 부호할 수 있으면 그럴 때마다 언제나 성능을 개선할 수 있습니다. 명심할 것은, log 와 exp 와 같은 일부 복잡한 함수는 `ps_2_0` 이상의 프로파일의 마이크로명령이므로, 성능이 최적일 때에는 텍스처에서 부호화할 필요가 없습니다.

3.5.3.1. 픽셀당 라이팅

2D 텍스처 사용

픽셀당 라이팅을 통해 우리는 흔히 텍스처의 유용성을 발견할 수 있습니다. 한 축을 `(N dot L)`으로 표시하고 또 다른 축을 `(N dot H)`으로 표시하는 2D 텍스처를 사용할 수 있습니다. 각 위치(u, v) 에서 텍스처가 부호화됩니다.

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

이것은 발산 조건과 정반사 조건에 대한 클램핑을 비롯하여 표준 Blinn 라이팅 모델입니다.

1D ARGB 텍스처 사용

(N dot H)라고 표시된 1D ARGB 텍스처를 사용하면 편리합니다. 텍스처는 (N dot H)를 각 채널의 다양한 지수로 부호화합니다. 예를 들면 다음과 같습니다.

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

그런 다음, 이 값을 혼합하여 쉐이딩에 대한 흑백 반사 값을 전달하는 4 요소의 가중 상수를 각 자료에 할당합니다. 이런 방식의 장점은 GeForce 4-클래스 하드웨어에서 작동할 수 있고 다양한 외관을 갖출 수 있을 만큼 유연하다는 사실입니다.

3D 텍스처 사용

3D 텍스처를 사용하여 혼합(mix)에 정반사 지수화를 추가할 수도 있습니다. 처음 두 축은 이전 항목에서 설명한 2D 텍스처 기법을 사용하고 세 번째 축은 정반사 지수(광택)를 부호화합니다.

하지만 텍스처가 너무 크면 캐시 성능에 지장이 올 수도 있다는 것을 기억하십시오. 가장 많이 사용하는 지수만 부호화하고 싶을 것입니다.

3.5.3.2. 벡터의 정규화

만약 여러분이 ps_1_* shader 쉐이더를 쓰고 있다면, 정규화 입방체 지도를 사용하여 벡터를 신속하게 정규화하십시오. 품질을 더 높이고 싶다면 16 비트의 입방체 맵 두 개를 사용할 수 있습니다. 이 중 하나는 x 와 y 를 위한 것이고, 나머지 하나는 z 를 위한 것입니다.

또 다른 최적화는 정규화된 벡터 V 가 사실상 그 기준이 1 에 가깝다는 사실을 근거로 합니다. 벡터 V 는 삽입되거나 여과된 표준이기 때문입니다. 이것은 곧, Taylor 확장의 첫 번째 조건(x=1 일 때 / sqrt(x))에 의하여 $1 / ||V||$ 에 근접할 수 있습니다.

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

그래서 이렇게 됩니다.

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

이 공식은 2 가지 어셈블리 명령어로 쓸 수 있습니다.

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

r0 은 V 를 포함하고 r1 의 최종 값은 V / ||V||.을 포함합니다.

_x2 레지스터 변경자 때문에 코드는 오직 ps_1_4 의 경우에만 유효합니다. 버전이 낮은 픽셀 쉐이더의 경우, 다음 공식을 대신 사용할 수 있습니다.

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

이것은 r0 이 ½ (V + 1)을 포함하고 있다는 것을 가정하고 있습니다. ½ (V + 1)는 V 가 [-1, 1]에서 [0, 1], 픽셀 쉐이더로 거리를 압축해서 전해야 할 때 거의 제약이 되지 않습니다.

GeForce 6 시리즈 GPU 는 쉐이더 주기 중에 무료로 fp16 벡터를 정규화할 수 있는 특수한 1/2 정밀도 정규화 장치를 갖고 있습니다. 이 기능을 활용하여 fp16 분량으로 정규화를 그냥 수행하십시오. 그러면 컴파일러가 nrmh 명령을 생성할 것입니다.

정규화에 관한 자세한 내용은 Normalization Heuristics and Bump Map Compression 백서를 참조하십시오. 다음 링크에서 확인할 수 있습니다.

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

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

3.5.3.3. sincos() 함수

앞서 말한 도움말에도 불구하고, GeForce FX 계열과 이후 나온 GPU 는 하드웨어의 일부 복잡한 수학 함수를 지원합니다. 그런 기능 중에 편리한 한 가지가 sincos 함수입니다. 이 함수를 사용하면 사인 값과 코사인 값을 동시에 계산할 수 있습니다.

3.6. 성능

3.6.1. 2 배속 Z 전용 스텐실 렌더링

오직 깊이나 스텐실 값을 렌더링할 때 GeForce FX 와 GeForce 6 시리즈 GPU 는 2 배속으로 렌더링 작업을 합니다. 이 특수한 렌더링 모드를 가동하려면 다음 규칙을 따라야 합니다.

- 컬러 쓰기를 해제합니다.

- 실행 가능 상태의 깊이-스텐실 표면은 견본을 여러 개로 하지 않습니다.
- Texkill 은 어떠한 프래그먼트에도 적용하지 않았습니다.
- 깊이 교체(`oDepth`, `texm3x2depth`, `texdepth`)는 어떠한 프래그먼트에도 적용하지 않았습니다.
- 알파 테스트를 해제합니다.
- 실행 가능 상태의 텍스처에서 컬러 키를 사용하지 않습니다.
- 사용자 클립핑 면을 가동하지 않습니다.

3.6.2. 초기 Z 최적화

초기 Z 최적화(간혹 "z-cull 이라고 칭함)는 이미 랜더링된 표면의 렌더링을 피함으로써 성능을 개선합니다. 랜더링된 표면에 값비싼 쉐이더를 적용하면, z-cull 은 상당한 연산 시간을 절약할 수 있습니다. z-cull 을 활용할 때에는 다음 지침을 따릅니다.

- 안에 구멍이 있는 삼각형을 형성하지 마십시오(다시 말해, 알파 테스트나 texkill 을 피하십시오).
- 깊이를 수정하지 마십시오(다시 말해, GPU 가 삽입된 깊이 값을 사용하도록 허용하십시오).

이 규칙을 위반하면 GPU 가 초기 최적화에 사용하는 데이터를 무효화하고 깊이 버퍼가 다시 초기화될 때까지 z-cull 을 해제할 수 있습니다.

3.6.3. 먼저 깊이를 규정

앞서 언급한 2 가지 성능 기능을 활용하는 가장 좋은 방법은 "우선 깊이를 규정하는 것"입니다. 이는 곧, 여러분이 2 배속 깊이 렌더링을 사용하여 첫 단계로 (쉐이딩 없이) 화면을 그려야 한다는 것을 의미합니다. 그러면 보는 사람에게 가장 가까운 표면이 만들어집니다. 이제 여러분은 화면을 다시 렌더링할 수 있지만, 쉐이딩은 완벽합니다. z-cull 은 보이지 않는 프래그먼트를 자동으로 골라낼 것입니다. 따라서 쉐이딩 연산에 들이는 시간을 줄일 수 있습니다.

우선 깊이를 규정하려면 자체 렌더링 패스가 필요하지만, 많은 봉쇄된 표면에 값비싼 쉐이딩을 적용하면 성능이 좋아질 수 있습니다. 2 배속 렌더링은 삼각형 크기가 작을수록 효율성이 떨어진다. 그리고 작은 삼각형은 z-cull 효율성을 저해할 수 있습니다.

또 다른 관련 기법은 Deferred Shading 입니다. NVSDK 7.1 이상에서 볼 수 있습니다.

3.6.4. 메모리 할당

애플리케이션 포화상태(thrashing) 비디오 메모리의 가능성을 최소화하기 위해, 쉐이더와 렌더 타겟을 할당하는 가장 좋은 방법은 다음과 같습니다.

1. 먼저 렌더 타겟을 할당합니다
   - 할당 순서를 피치 기준(width * bpp)으로 정렬합니다.
   - 사용 빈도에 따라 서로 다른 피치 그룹을 정렬합니다. '가장 자주'로 판단되는 표면을 먼저 할당합니다.
2. 버텍스와 픽셀 쉐이더를 형성합니다
3. 남은 텍스처를 로드합니다

3.7. 안티앨리어싱(Antialiasing)

GeForce FX 계열과 GeForce 6 시리즈는 강력한 안티앨리어싱 엔진을 달고 있습니다. 이들은 안티앨리어싱을 가동했을 때 최상을 능력을 발휘합니다. 따라 애플리케이션에 안티앨리어싱을 작동할 것을 권합니다.

안티앨리어싱을 가동했을 때 작동하지 않는 기법을 사용해야 할 경우 저희에게 연락 주십시오. 기꺼이 여러분과 문제를 논의하고 해결책을 찾도록 도움을 드리겠습니다.

DirectX9.0b 이상의 경우 지금 현재 해결되지 않는 한 가지 문제는 후처리 효과를 가진 안티앨리어싱을 사용하는 것입니다. `StretchRect()` 호출은 멀티샘플링과 연계하여 백 버퍼를 오프스크린 텍스처에 복사할 수 있습니다.

예를 들어, 100 × 100 백 버퍼에서 4x 멀티샘플링을 가동할 경우, 안티앨리어싱을 수행하기 위해 드라이버가 실제로 내부적으로 200 × 200 백 버퍼와 깊이 버퍼를 형성합니다. 애플리케이션이 100 × 100 오프스크린 텍스처를 형성할 경우, 전체 백 버퍼를 오프스크린 표면에 `StretchRect()`할 수 있으므로, GPU 는 안티앨리어싱 처리한 버퍼를 오프스크린 버퍼로 여과합니다.

그런 다음, 발광 효과와 기타 후처리 효과는 100 × 100 텍스처에서 수행할 수 있고 그 후 메인 백 버퍼에 다시 적용할 수 있습니다.

실제 버퍼 크기(200 × 200)와 애플리케이션의 뷰(100 × 100) 간의 해상도 부조화는 멀티샘플 버퍼를 비멀티샘플 렌더 타겟에 첨가할 수 없는 이유가 됩니다.

Chapter 4.
GeForce 6 시리즈 프로그래밍 정보

이 장은 GeForce 6 시리즈와 Quadro FX 4xxx GPU 의 능력을 완벽하게 사용하는 데 도움이 되는 몇 가지 유용한 정보를 소개하고 있습니다. 일부는 성능에 영향을 미칠 수도 있지만, 여기 나온 정보는 대부분 기능을 중심으로 하고 있습니다.

4.1. Shader Model 3.0 지원

Microsoft DirectX 9.0 은 고급 버텍스 및 픽셀 쉐이더 기술(버전 2.0 과 버전 3.0)에 대한 몇 가지 새로운 표준을 소개했습니다. Shader Model 2.0 하드웨어는 2002 년 말부터 보급되었으며, 오늘날 제공되는 GPU 의 대부분은 Shader Model 2.0 이상을 지원합니다. Shader Model 2.0 은 고급 라이팅과 애니메이션 기법에 유용한 기술을 포함하고 있지만, 쉐이더 프로그램 길이와 복잡성이 한정되어 있습니다. 그래서 실현할 수 있는 효과의 충실도가 한정됩니다.

개발자들은 Pixel Shader 2.0 와 Vertex Shader 2.0 에 내재하는 제약사항을 극복하려고 하면서 더 새롭고 발전된 Shader Model 3.0 을 채택하기 시작했습니다. 이 쉐이더 모델은 몇 가지 분야에서, 특히 픽셀 쉐이더 처리와 버텍스 쉐이더 처리에서 모두 발전을 이루었습니다.

4.1.1. Pixel Shader 3.0

다음은 Pixel Shader 2.0 과 3.0 의 주된 차이점을 간략하게 설명한 기능 요약입니다.

픽셀 쉐이더 기능	Shader 2.0	Shader 3.0	설명
쉐이더 길이	96	65535+	더 복잡한 쉐이딩, 라이팅, 절차 자료를 허용합니다
동적 분기	없음	있음	관련이 없는 픽셀의 경우 복잡한 쉐이딩을 건너뜀으로써 성능을 절약합니다
쉐이더 안타알리아싱	지원하지 않음	내장형 도함수 명령	개발자들은 어떤 함수라도 화면 공간 도함수를 계산할 수 있으므로, 쉐이딩 빈도나 오버 샘플링을 조정해서 결과물을 제거할 수 있습니다.
후면 레지스터	없음	있음	단일 패스에서 양면 라이팅을 허용합니다
삽입된 컬러 형식	8 비트의 정수 최소값	32 비트 부동 소수점 최소값	범위와 정밀도 컬러가 더 높아서 버텍스 수준에서 매우 동적인 범위 라이팅이 가능합니다
여러 개의 렌더 타겟	선택적	4 개 필요함	고급 라이팅 알고리즘을 사용하여 필터링과 버텍스 작업을 면할 수 있고, 그래서 최소한의 비용으로 라이트를 늘릴 수 있습니다
안개와 정반사	8 비트의 함수 최소값	주문형 fp16-fp32 쉐이더 프로그램	Shader Model 3.0 은 개발자에게 정반사/안개 연산과 고정 함수에 대한 완벽하고 정확한 지배권을 발휘합니다
텍스처 좌표 계수	8	10	픽셀당 입력량이 높아서 특히 스킨의 경우 좀더 사실적인 렌더링이 가능합니다

4.1.2. Vertex Shader 3.0

Vertex Shader Model 2.0 에서 3.0 으로 이동할 때 개발자들이 즐기는 비슷한 핵심 기능 목록이 있습니다.

버텍스 쉐이더 기능	Shader 2.0	Shader 3.0	설명
쉐이더 길이	명령어 256 개	명령어 65535 개	명령어가 많아지면 더 자세한 인물 라이팅과 애니메이션이 가능합니다
동적 분기	없음	있음	관련이 없는 버텍스의 경우 애니메이션과 연산 작업을 건너뜀으로써 성능을 아낄 수 있습니다
버텍스 텍스처	없음	최대 4 개 텍스처의 검색 횟수 무제한	이동 매핑, 입자 효과를 허용합니다
인스턴싱 지원	없음	필요함	단 하나의 명령어로 다양한 객체를 많이 그릴 수 있습니다

4.1.3. 동적 분기(Dynamic Branching)

Shader 3.0 모델(버텍스와 픽셀)의 한 가지 중요한 기능은 동적 분기(Dynamic Branching)입니다. 간단하게 말해서, 쉐이더 작성자는 이 기능을 사용하여 진정한 루프와 조건문을 형성할 수 있습니다. 예를 들어, 누구나 특정한 수의 버텍스 라이트를 통해 계속 반복하는 쉐이더를 작성하여 어떤 것이 특정한 버텍스에 영향을 미칠 수 있는지 결정한 다음, 관련 라이트의 색인을 픽셀 쉐이더에 전달할 수 있습니다. 픽셀 쉐이더는 이 '라이트 색인'을 사용하여 어떤 라이트 매개변수를 적용할지 결정할 수 있습니다. 그러면 픽셀 쉐이더는 작동 중인 라이트에 걸쳐 반복하고, 일단 모든 라이트를 처리하면 동적 분기를 사용하여 초기에 쉐이더에서 나갈 수 있습니다.

대부분의 라이트 유형은 빛과 마주하는 면인 객체의 앞면에만 적용됩니다. 따라서 버텍스 동기와 픽셀 동기를 모두 사용하여, 빛을 외면하는 것처럼(새로운 '후면 레지스터' 사용) 쉐이더가 감지하는 라이트의 처리를 건너뛸 수 있습니다. 그러면 상당한 처리 시간을 절약하고 쉐이더의 속도를 높일 수 있습니다. 이와 같은 고속화는 많은 유사 알고리즘은 물론 인물 뼈대 애니메이션의 처리를 건너뛰는 데 사용할 수 있습니다.

4.1.4. 더욱 간편한 코드 유지보수

게임 엔진이 복잡해질수록 Pixel Shader 2.0 프로그램 길이 제한에 맞추기 위해 쉐이더 버전이 다양하게 만들어지게 마련입니다. 그러면 코드 유지보수 작업, 쉐이더 컴파일 시간, 수준 로딩 시간이 늘어날 뿐만 아니라 실행 시간에 중요한 시스템 메모리를 차지합니다. Shader Model 3.0 은 포괄적인 반복 실행과 동기를 통해 이 문제를 처리하므로, 실행 시간의 정확한 실행 경로를 선택하기 위해 엔진이 적절한 정적/동적 동기를 포함하여 단 하나의 버텍스 쉐이더와 단 하나의 픽셀 쉐이더를 작성하는 것이 가능합니다. 그래서 쉐이더 조합 급증이 크게 단순해집니다.

4.1.5. 인스턴싱(Instancing)

Shader Model 3.0 의 또 다른 핵심 기능은 Microsoft DirectX Instancing API 에 대한 지원입니다. 현재, 게임은 화면에 표시할 수 있는 고유 객체의 수 면에서 제약이 있습니다. 그래픽 성능 때문이 아니라, 동일하지만 조금씩 차이를 보이는 많은 객체를 저장하거나 제출하는 등의 CPU 측 오버헤드 때문입니다. 예를 들어, 숲은 서로 비슷한 나무들로 이루어져 있지만, 각 나무는 위치가 다르고 높이가 다르고 잎의 색깔이 다릅니다. 원하는 변화를 추가하기 위해 개발자들은 똑같이 생겼지만 약간 다른 나무 여러 개를 만드는 것과 나무 하나하나의 크기를 조정하고 색을 바꾼 다음 위치를 정해야 합니다.

인스턴싱을 사용하면 프로그래머가 트리 하나를 저장한 다음 몇 가지 다른 버텍스 데이터 스트림을 저장하여 인스턴스당 컬러, 높이, 지류 크기 등을 지정할 수 있습니다. 예를 들어, 1000 버텍스의 트리 모델 하나에는 버텍스 위치와 노말이 포함되며, 200 요소의 버텍스 스트림에는 위치, 컬러, 높이가 포함됩니다. 프로그래머는 인스턴싱을 사용하여 그리기 요청을 한번 제출할 수 있습니다. 여기서 기본적인 트리 형태의 경우 동일한 데이터를 사용하여 200 개의 트리를 만들지만, 그런 다음 인스턴스당 스트림에서 다양화됩니다.

인스턴싱 코드 샘플은
http://download.nvidia.com/developer/SDK/Individual_Samples/samples.html
에서 확인할 수 있습니다.

4.1.6. 요약

요컨대, DirectX 9.0 Shader Model 3.0 은 사용 편의성, 성능, 쉐이더 복잡성 측면에서 한 단계 앞선 중요한 프로그램입니다. 동적 분기는 초반기 기회를 포함하는 많은 알고리즘을 가속화하며, 이와 동시에 그래픽 엔진과 툴의 쉐이더 코드 경로를 단순화하기도 합니다. 마지막으로, 인스턴싱은 CPU 와 메모리 오버헤드가 매우 낮은 경우 심한 복잡성을 허용합니다.

4.2. sRGB 인코딩

sRGB 인코딩은 인간의 시각 계통을 모방하여 제로에 가까운 정밀도를 제공하기 위해 감마 변환을 사용한 형식입니다. 그리고 DXT 압축 형식과 함께 작동하여, 애플리케이션은 색상 충실도의 개선과 저장 크기의 감소에서 모두 이점을 얻을 수 있습니다.

GeForce 6 시리즈 GPU 에서, 여러분은 경우에 따라 부동 소수점 형식을 사용하고 싶어할 수도 있습니다. 부동 소수점 형식이 다음과 같은 장점을 제공하기 때문입니다.

- 전체 범위에 걸쳐 정밀도가 높아짐
- 규모가 훨씬 커진 선형 동적 범위
- 프레임 버퍼의 경우 선형 블렌딩

텍스처에는 부동 소수점보다 sRGB 를 훨씬 더 선호하는 편입니다. sRGB 가 메모리 면적과 대역폭 사양이 적기 때문입니다.

4.3. 별도의 알파 블렌딩

GeForce 6 시리즈 GPU 에서 여러분은 컬러와 알파에 대해 별도의 블렌딩 기능을 지정할 수 있습니다. 그래서 예를 들어 텍스처의 알파 채널에 저장된 값의 경우, 유연성이 더 좋습니다. 이것과 관련한 한 가지 용도는 알파를 보존하는 동시에 컬러 채널을 모듈화하는 것일 것입니다.

4.4. 지원되는 텍스처 형식

다음 표는 GeForce 6 시리즈 GPU 에서 지원하는 텍스처 형식을 나타낸 것입니다.

정수 형식	2D	입방체	3D	MIP	필터	sRGB	렌더	블렌드	버텍스
R8G8B8	N	N	N	N	N	N	N	N	N
A8R8G8B8	Y	Y	Y	Y	Y	Y	Y	Y	N
X8R8G8B8	Y	Y	Y	Y	Y	Y	Y	Y	N
R5G6B5	Y	Y	Y	Y	Y	Y	Y	Y	N
X1R5G5B5	Y	Y	Y	Y	Y	Y	Y	Y	N
A1R5G5B5	Y	Y	Y	Y	Y	Y	N	N	N
A4R4G4B4	Y	Y	Y	Y	Y	Y	N	N	N
R3G3B2	N	N	N	N	N	N	N	N	N
A8	Y	Y	Y	Y	Y	N/A	N	N	N
A8R3G3B2	N	N	N	N	N	N	N	N	N
X4R4G4B4	N	N	N	N	N	N	N	N	N
A2B10G10R10	N	N	N	N	N	N	N	N	N
A8B8G8R8	N	N	N	N	N	N	N	N	N
X8B8G8R8	N	N	N	N	N	N	N	N	N
G16R16	Y	Y	Y	Y	Y	N	N	N	N
A2R10G10B10	N	N	N	N	N	N	N	N	N
A16B16G16R16	N	N	N	N	N	N	N	N	N
A8P8	N	N	N	N	N	N	N	N	N
P8	N	N	N	N	N	N	N	N	N
L8	Y	Y	Y	Y	Y	N	N	N	N
L16	Y	Y	Y	Y	Y	N	N	N	N
A8L8	Y	Y	Y	Y	Y	N	N	N	N
A4L4	N	N	N	N	N	N	N	N	N

float 형식	2D	입방체	3D	MIP	필터	sRGB	렌더	블렌드	버텍스
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

그림자 맵	2D	입방체	3D	MIP	필터	sRGB	렌더	블렌드	버텍스
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. 부동 소수점 텍스처

GeForce 6 시리즈 GPU 는 부동 소수점 텍스처의 경우 한층 발전한 지원을 제공합니다. 다음 표는 요소 기준 16 비트(fp16)과 요소 기준 32 비트(fp32) 부동 소수점 텍스처에 대해 지원하는 다양한 기능을 나타낸 것입니다.

텍스처 요소 유형	가장 가까운 필터링	이선/삼선 필터링	이방성 필터링	밉맵 지원	3D 텍스처	입방체 맵	Non-power-of-2 Textures
16 비트	있음	있음	있음	있음	있음	있음	있음
32 비트	있음	없음	없음	있음	있음	있음	있음

4.5.1. 제약사항

우리는 R16F 형식을 지원하지 않는다는 점에 주의하십시오. 그러니 대신 G16R16F 를 사용하십시오. 그리고 G16R16F 또는 R23F 표면이 아니라 그냥 A16B16G16R16F 표면에 혼합할 수 있습니다. 그러나 G16R16F 텍스처에서는 필터링을 지원하지 않습니다.

4.6. MRT(Multiple Render Target)

GeForce 6 시리즈 GPU 는 MRT 를 지원합니다. 그래서 픽셀 쉐이더가 최대 4 개 타겟까지 데이터를 작성할 수 있습니다. 픽셀 쉐이더가 4 개 이상의 float 값을 연산하고 텍스처에 이 즉각적인 결과를 저장해야 할 때마다 MRT 가 유용합니다.

MRT 의 용례에는 위치와 속도를 동시에 연산하는 입자물리학과 유사 GPGPU 알고리즘이 포함되어 있습니다. 이연 쉐이딩(deferred shading)은 여러 개의 float4 값을 동시에 연산하고 저장하는 또 다른 기법입니다. 이 기법은 모든 가령, 표면 노말, 발산, 정반사 물질 속성과 같은 모든 물질 속성을 연산하고 이것을 별도의 텍스처에 저장합니다. 이러한 속성은 화면에 여러 개의 조명을 연속 패스로 비출 때 사용됩니다.

DirectX caps bit `NumSimultaneousRTs` 는 그래픽 장치가 동시에 얼마나 많은 렌더 타겟을 렌더링할 수 있는가를 나타냅니다. GeForce 6 시리즈 GPU 는 caps bit 가 4 입니다. MRT 를 가동할 때에는 `SetRenderTarget(index, pRenderTarget)` API 호출을 이용하십시오. 이 호출은 통과된 렌더 타겟 텍스처를 주어진 렌더 색인으로 고정합니다. 그런 다음 픽셀 쉐이더는 `oC0`, `oC1`, `oC2`, `oC3` 출력 레지스터를 이용하여 결합된 렌더 타겟 텍스처에 출력합니다. MRT 렌더링을 해제하려면 0 부터 3 까지 색인의 렌더 타겟을 NULL 로 재설정해야 한다는 것을 기억하십시오.

MRT 는 다른 GPU 기능을 제한합니다. 무엇보다도, 하드웨어에서 가속화된 안티앨리어싱은 MRT 렌더 타겟에 맞지 않습니다. 게다가, 모든 렌더 타겟은 너비, 높이, 비트 깊이가 동일합니다. GeForce 6 시리즈의 경우, 이것은 곧, 누구나 자유롭게 32 비트 형식 안에서 마음껏 조합하거나 각각 64 비트인 렌더 타겟(즉, `A16R16G16B16F` 형식을 이용함)을 최대 4 개까지, 또는 각각 128 비트인 렌더 타겟(즉, `A32R32G32B32F` 형식을 이용함)을 최대 4 개까지 사용할 수 있다는 것을 의미합니다.

게다가, `D3DMISCCAPS_MRTPOSTPIXELSHADERBLENDING` caps bit 가 설정되고 `USAGE_QUERY_POSTPIXELSHADERBLENDING` 의 렌더 형식을 조회해서 사용가능하면, 포스트 픽셀 쉐이더 혼합 작업인 알파 블렌딩, 알파 테스트, 안개효과, 디더링은 MRT 에만 이용할 수 있습니다.

GeForce 6 시리즈 칩은 모든 MRT 형식에 대해 이 기능을 지원합니다. 하지만, `R32F`, `G16R16F`, and `A32R32G32B32F` 는 제외합니다. 따라서 GeForce 6 시리즈 GPU(GeForce 6200 은 제외)가 `A16R16G16B16F` 부동 소수점 렌더 타겟에서 혼합 후작업을 지원한다는 사실을 특히 명심하십시오. DirectX 규격은 MRT 포스트 혼합 후작업의 행동을 수정합니다. MRT 렌더링은 디더링 상태를 무시하고 렌더 타겟 제로의 경우에만(안개 상태를 존중하며 렌더 타겟 1 부터 3 까지는 안개가

해제된 것처럼 행동함), 알파 테스트는 `oC0.a` 값만 이용하여 렌더 타겟 픽셀 4 개를 모두 버릴 것인지 여부를 결정합니다.

마지막으로, MRT 를 이용하면 성능에 영향이 미칩니다. MRT 는 특히 더 넓은 비트 길이를 이용할 때 더 많은 프레임 버퍼 대역 비용이 듭니다. 예를 들어, 4 개의 `A32R32G32B32F` 표면으로 렌더링을 하면 단 한 개의 `A8R8G8B` 으로 렌더링하는 것과 비교하여 프레임 버퍼 대역이 16 배 필요합니다! 게다가, GeForce 6 시리즈는 3 개 이하의 렌더 타겟을 동시에 이용할 때 좋은 성능을 발휘합니다.

따라서 다음에 나오는 일반적인 성능 도움말을 적용하자면 이렇습니다. 필요할 때에만, 다시 말해 MRT 가 여러 개의 패스를 저장할 때에만 MRT 를 이용하십시오. 가령 데이터를 단단히 고정하고 알파 채널을 낭비하지 않음으로써, 렌더 타겟과 비트 길이의 숫자를 최소화하십시오. 그리고 반드시 MRT 렌더 타겟을 조기에 할당하십시오(3.6.4 항 메모리 할당 참조).

또한 MRT 출력을 3 개 이하 그룹으로 분리해도 패스의 총 수가 늘어나지 않을 때, 그렇게 분리하십시오. 예를 들어, 애플리케이션이 주위 패스를 렌더링하고 그런 다음 MRT 4 개로 나오는 패스를 렌더링할 경우, 주위 패스 동안 타겟 하나를 출력한 다음 MRT 3 개로만 출력하는 것을 고려하십시오. 그렇게 하면, 타겟 하나를 다른 타겟보다 정밀도를 낮게 하여 저장할 수 있을 경우에 특히 유익하며, 다른 타겟과 독립적으로 편리하게 연산할 수 있습니다(예: 물질 발산 텍스처 맵). 여러분은 SDK 7.1 버전에서 또는 ftp://download.nvidia.com/developer/Movies/NV40-LowRes-Clips/Deferred_Shading.avi에서 이연 세이딩 데모 클립을 내려 받아 이연 쉐이딩에 대해 더 많을 것을 배울 수 있습니다.

4.7. 버텍스 텍스처링

GeForce 6 시리즈 GPU 는 버텍스 텍스처링을 지원하지만, 버텍스 텍스처링은 일정한 RAM 과 같이 취급하지 않습니다. 버텍스 텍스처는 매우 일정한 읽기와 달리 데이터를 가져올 때 지연을 초래하고 이후 텍스처를 꺼낸 결과를 이용하기에 앞서 지연을 갖추는 많은 산술 작업이 이어집니다. 버텍스 텍스처는 거대하게 나열된 상수의 대체물이 아닙니다. 부족한 버텍스당 데이터에 맞춰 만든 것입니다. 그래서 버텍스마다 버텍스 텍스처 꺼내기는 얼마 되지 않습니다.

버텍스 텍스처에 관해서는 http://developer.nvidia.com/object/using_vertex_textures.html에 있는 Using Vertex Textures 백서를 읽으면 많이 알아볼 수 있습니다.

4.8. 일반적인 성능 도움말

GeForce 6 시리즈 아키텍처에는 효율성과 다양성을 고려하여 개선한 것들이 많습니다. 여기에 그 기능을 활용해볼 수 있도록 몇 가지 정보를 소개합니다.

- **쓰기 마스크와 혼합을 이용합니다.** GeForce 6 시리즈 쉐이더 아키텍처는 (공동 발행과 이중 발행을 통해) 각기 다른 장치에 4 요소 벡터의 위치를 잡을 수 있습니다. 그러면 쉐이더 효용이 높아집니다. 쓰기 마스크와 혼합을 이용하면 컴파일러에서 이런 종류의 스케줄 기회를 파악하는 데 도움이 됩니다.
- **가능할 때마다 부분 정밀도를 이용합니다.** GeForce 6 시리즈 GPU 의 경우 부문 정밀도를 이용하는 데에는 2 가지 이유가 있습니다. 첫째, GeForce 6 시리즈는 쉐이더에 특수한 무료 fp16 정규화 정치가 있기 때문입니다. 그래서 16 비트의 부동 소수점 정규화가 다른 연산과 병렬로 매우 효율적으로 이루어질 수 있습니다. 이것을 활용하고 싶으면, 그냥 적당할 때마다 프로그램에 부문 정밀도를 이용하십시오. 둘째, 부문 정밀도는 레지스터 압력을 줄여서 잠재적으로 성능을 증대하는 효과가 있기 때문입니다.
- **분기가 상당한 일관성을 보이면 동적 분기를 이용합니다.** 4.1.3 에서 언급한 것과 같이, 동적 분기는 코드를 신속하고 편리하게 구축할 수 있도록 도와줍니다. 그러나 최적의 작동을 기하려면, 분기가 상당한 일관성을 보여야 합니다(예를 들어, 대략 30 x 30 픽셀의 지역에 걸쳐).

4.9. 노말맵

노말맵 스토리지가 애플리케이션의 문제라면, 장치 길이 탄젠트 공간 노말의 노말맵 압축은 요소 하나를 저장할 필요성이 없는 반구체 매핑 기법을 이용하여 GeForce 6 CPU 에서 할 수 있습니다.

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

이렇게 하면 3-5 픽셀 쉐이더 명령으로 컴파일되므로 이 기법을 통해 성능을 개선할 수도 있습니다. 하지만 이는 픽셀 쉐이더 명령어를 기존의 다른 쉐이더 명령어와 함께 공동 발행할지 여부와 텍스처를 꺼내는 것이 장애가 되는지 여부에 따라 다릅니다.

반구체 매핑 기법을 따르기로 결정할 경우, GeForce 6 GPU 에서 선호하는 텍스처 형식은 DirectX 의 `D3DFMT_V8U8` 와 OpenGL 의 `GL_LUMINANCE8_ALPHA8` 입니다. 이 형식은 16 비트/픽셀이며, 이를 통해 2:1 의 전혀 손해 없는 압축이 이루어집니다.

반구체 매핑은 유연성을 떨어뜨리지 않습니다. 양성(+)의 장치 길이 노말만이 생성되기 때문입니다. 음성(-) 값(객체 공간 노말 매핑 등)이나 비장치 노말(http://developer.nvidia.com/object/mipmapping_normal_maps.html에 설명되어 있는 안티앨리어싱 기법 등)에 의존하는 기법은 가능하지 않을 것입니다.

노말맵에 대한 자세한 내용은 http://developer.nvidia.com/object/bump_map_compression.html에 있는 *Bump Map Compression* 백서를 참조하십시오.

낮은디테일 모델을 높은디테일 모델처럼 보이게 만드는 저고품질의 노말맵을 만들려면, NVIDIA Melody를 이용하십시오. 낮은디테일 모델을 로딩한 다음 높은디테일 레퍼런스 모델을 로딩하고 "노말 맵 만들기" 버튼을 누른 다음 Melody가 신속하게 작업을 처리하는것을 지켜보기만 하면 됩니다. Melody는 http://developer.nvidia.com/object/melody_home.html에 가면 있습니다.

Chapter 5.
GeForce FX 프로그래밍 요령

이 장에서는 GeForce FX 계열의 기능을 완벽하게 이용하는 데 도움이 되는 몇 가지 유용한 정보를 소개하고 있습니다. 일부는 성능에 영향을 미칠 수도 있지만, 여기 나온 정보는 대부분 기능이 중심입니다.

5.1. 버텍스 쉐이더

강력한 GeForce FX 엔진은 GeForce FX 5900 에서 초당 2 억 개가 넘는 삼각형을 달성할 수 있습니다. 이 엔진은 동적 분기를 지원합니다. 그래서 조명, 뼈 등의 수와 종류를 기준으로 하여 쉐이더 작성자가 분기할 수 있습니다.

각각의 활성 분기 쓰레드는 전체 실행을 느리게 만듭니다. 그래서 중요한 버텍스 계산이 절약되거나 API 를 통해 원시 배치 크기가 늘어납니다.

5.2. 픽셀 쉐이더 길이

본래 GeForce FX 는 Direct3D 에서 패스당 512 개의 픽셀 명령어를 처리할 수 있고 OpenGL 에서 1024 개의 명령어를 처리할 수 있습니다. DirectX 의 경우, `ps_2_a` 또는 `ps_2_x` 프로파일로 컴파일하십시오.

OpenGL 의 경우, GLSL, Cg(`arbfp1` 또는 `fp30` 프로파일로 압축함)를 이용하거나 직접 `ARB_fragment_program` 확장을 이용할 수 있습니다.

Quadro FX 카드는 패스당 2048 개의 픽셀 명령어를 처리할 수 있습니다.

5.3. DirectX 픽셀 쉐이더

최신 DirectX 9 `ps_2_0` 이상의 쉐이딩 모델은 기본 설정상 24 비트 이상의 정밀도에서 수학과 일시적인 작업을 연산해야 합니다(이 경우 GeForce FX 계열은 32 비트 float 유형을 이용합니다). 애플리케이션은 16 비트의 부동 소수점 정밀도를 달성하기 위해 어셈블리에서 a `_pp` 변경자를 지정할 수 있습니다.

HLSL 이나 Cg 를 이용하면 이 작업이 매우 간편합니다. 32 비트 정밀도의 경우 변수를 float 로 지정하고 16 비트 정밀도의 경우 half 로 지정합니다.

우선 가장 편리한 방식으로 쉐이더를 작성한 다음 필요한 만큼 레지스터 용도와 1/2 정밀도로 최적화할 것을 권합니다.

고정 소수점 블렌딩은 고정 함수 텍스처 블렌딩과 `ps_1_0` – `ps_1_4` 쉐이더 모델의 텍스처 산술 부분에 주로 쓰입니다.

DirectX 에서 여러분은 `ps_2_0` 를 통해 고정 소수점 정밀도를 요청할 수 있습니다. 프로그램을 `ps_1_1` – `ps_1_4` 에 맞출 수 있다면, 고정 소수점 쉐이딩 하드웨어의 증대된 유용성 때문에 더 빠르게 작동할 것입니다.

5.4. OpenGL 픽셀 쉐이더

`ARB_fragment_program` 확장은 최소한 24 비트의 부동 소수점 정밀도를 요구합니다. 정밀도를 제어하기 위해 `ARB_fragment_program` 소스 코드 맨 위에 다양한 플래그를 둘 수 있습니다.

- `NV_fragment_program`. 제어와 성능이 최고조일 때 half(16 비트 부동 소수점)와 fixed(12 비트 고정 소수점) 형식을 명시적으로 이용할 수 있습니다.

- `ARB_precision_hint_fastest`. Unified Compiler 는 실행할 때 각 쉐이더 작업에 대해 적절한 정밀도를 결정하여, 시각적 산물을 최소로 한 상태에서 전체 성능을 증대할 것입니다.
- `ARB_precision_hint_nicest`. 전체 프로그램이 float 정밀도로 작동할 수 있도록 강요합니다.

5.5. 16 비트 부동 소수점 이용

많은 개발자들은 예전에 half 상태에서 작업하지 않았으며, float 를 "괜찮은 편인" 형식이고 범위와 정밀도와 관련하여 문제가 거의 없다고 생각하고 있습니다. half 는 10 개의 가수(假數) 비트와 5 비트의 지수를 지니고 있습니다. 그리고 float 는 23 비트의 가수와 8 비트의 지수를 지니고 있습니다. 각 가수 비트는 눈금자의 V 표시로 간주될 수 있습니다. 이것은 형식의 정밀도가 최고치라는 것을 나타냅니다. half 의 경우, 각 가수 비트간의 정밀도는 0.1%입니다. 지수 값은 눈금자의 길이로 간주될 수 있습니다. 지수가 높아지면서 눈금자가 길어질수록, 눈금자의 각 V 표시는 길이가 더 기다는 것을 나타냅니다. 이런 식으로 부동 소수점 형식은 정밀도와 범위를 자동으로 바꿉니다.

half 의 경우, -2048 부터 2048 까지 정수를 정확하게 나타낼 수 있지만, 분수 비트는 남아 있지 않습니다. 픽셀 쉐이더에서 뷰 또는 실제 공간의 계산을 수행할 경우, 정밀도가 부족할 수 있습니다. 예를 들어, 캐릭터가 4096 지점에 있고 조명이 4097 지점에 있을 경우, 두 캐릭터는 동일한 16 비트 부동 소수점 수로 나타냅니다. 이 값을 빼면 여러분은 영점이 됩니다. 그런 다음 결과를 제곱하고 정규화하면 `INF` 가 나옵니다. 이 대안책이 원래 쉐이더보다 간단하고 질서정연하고 심지어 빠르기까지 합니다. 행렬과 벡터 공제 작업을 버텍스 쉐이더로 옮기십시오.

처음에 GeForce FX GPU 를 이용할 때 조명 계산과 관련한 문제는 추적해보면 결국 half 형식의 오용일 때가 많습니다. half 형식은 컬러 때문에 영화에 광범하게 사용되긴 하지만 게임의 공통 형식이기 때문에 이것은 놀라운 일이 아닙니다.

버텍스 쉐이더는 최소한 float 지원을 요구합니다. 그래서 대규모의 세상 공간과 뷰 공간을 쉽게 처리할 수 있습니다. 또한, 우선 선형 계산이 버텍스 쉐이더에 왜 속하는지는 쉽게 알 수 있습니다. 그렇다면, 버텍스

단위로만 계산하고 GPU 에 의해 무료로 반복 적용할 수 있는데도 왜 각 픽셀에서 재계산합니까?

따라서 우리는 버텍스를 버텍스 쉐이더의 조명 공간이나 탄젠트 공간으로 옮기고 그렇게 나온 지점을 픽셀 쉐이더로 넘길 것을 권합니다. 한 가지 멋진 방법은 버텍스 쉐이더의 버텍스 위치에서 조명의 위치를 빼는 것입니다. 그런 다음 버텍스를 공히 상수로 기준화하여 값을 제로에 가깝게 하고 프래그먼트 쉐이더에서 벡터를 정규화합니다. (일정한 기준은 정규화된 결과에 영향을 미치지 않는다는 사실을 명심하기 바랍니다.)

어쨌든 목표는 탄젠트 공간에 조명을 수행하는 것일 때가 많습니다. 이 작업은 좌표 체계를 버텍스에 집중시키기 때문에 특히 유용합니다. 제로에 가까운 상태에서 작업하면 함께 작업할 수 있는 비트 기호가 주어지고, 따라서 더불어 가수 비트를 갖게 됩니다.

일반적으로 CPU 에서는 반복되는 계산을, 버텍스 쉐이더에서는 선형 계산을, 픽셀 쉐이더에서는 비선형 계산을 수행합니다.

5.6. 지원되는 텍스처 형식

정수 형식	2D	입방체	3D	MIP	필터	sRGB	렌더	블렌드	버텍스
R8G8B8	N	N	N	N	N	N	N	N	N
A8R8G8B8	Y	Y	Y	Y	Y	Y	Y	Y	N
X8R8G8B8	Y	Y	Y	Y	Y	Y	Y	Y	N
R5G6B5	Y	Y	Y	Y	Y	Y	Y	Y	N
X1R5G5B5	Y	Y	Y	Y	Y	Y	Y	Y	N
A1R5G5B5	Y	Y	Y	Y	Y	Y	N	N	N
A4R4G4B4	Y	Y	Y	Y	Y	Y	N	N	N
R3G3B2	N	N	N	N	N	N	N	N	N
A8	N	N	N	N	N	N/A	N	N	N
A8R3G3B2	N	N	N	N	N	N	N	N	N
X4R4G4B4	N	N	N	N	N	N	N	N	N
A2B10G10R10	N	N	N	N	N	N	N	N	N
A8B8G8R8	N	N	N	N	N	N	N	N	N
X8B8G8R8	N	N	N	N	N	N	N	N	N

G16R16	Y	Y	Y	Y	Y	N	N	N	N
A2R10G10B10	N	N	N	N	N	N	N	N	N
A16B16G16R16	N	N	N	N	N	N	N	N	N
A8P8	N	N	N	N	N	N	N	N	N
P8	N	N	N	N	N	N	N	N	N
L8	Y	Y	Y	Y	Y	N	N	N	N
L16	Y	Y	Y	Y	Y	N	N	N	N
A8L8	Y	Y	Y	Y	Y	N	N	N	N
A4L4	N	N	N	N	N	N	N	N	N

float 형식	2D	입방체	3D	MIP	필터	sRGB	렌더	블렌드	버텍스
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

그림자 맵	2D	입방체	3D	MIP	필터	sRGB	렌더	블렌드	버텍스
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

* 래핑(wrapping) 또는 밉매핑 기능이 없는 DX9.0c 에 있음

5.7. DirectX 에서 ps_2_x 와 ps_2_a 이용

GeForce FX 는 ps_2_0 기능 이외에 미분 계산(DDX 와 DDY 를 통해), 더 긴 쉐이더, 서술 지원을 비롯하여 몇 가지 기능을 지원합니다. 높은 수준의 쉐이딩 언어에서 2 가지 방식으로 이 기능을 이용할 수 있습니다. 한 가지 방식은 ps_2_x 프로파일과 함께 HLSL 또는 Cg 를 이용하여 컴파일하는 것입니다. 이 프로파일은 앞서 설명한 기능을 이용하지만 특정한 능력 비트 집합과 대조하지 않습니다. 더 나은 방법은 ps_2_a 프로파일을 이용하는 것입니다. 이 프로파일은 GeForce FX 계열의 쉐이딩 기능에 정확하게 들어맞으며 더욱 최적화된 코드를 만들어냅니다.

5.8. 부동 소수점 렌더 타겟 이용

GeForce FX 계열은 64 비트와 128 비트로 된 4 요소 부동 소수점 렌더 타겟과 1 요소 및 2 요소 32 비트 부동 소수점 렌더 타겟과 밉매핑한 텍스처를 지원합니다.

OpenGL 의 경우, 부동 소수점 텍스처는 `NV_float_buffer` 확장을 통해 드러나며, 여러 개의 저저밀도 요소는 `NV_fragment_program` 에 있는 팩/언팩 명령어를 이용하여 정밀도가 더 큰 단 하나의 요소에 편입될 수 있습니다. 애플리케이션은 일반적인 RGBA8 텍스처에 있는 팩/언팩 명령어를 이용하여 `NEAREST_MIPMAP_NEAREST` 부동 소수점 입방체 맵과 볼륨 텍스처를 모방할 수 있습니다(지원은 단 하나의 fp32 또는 2 개의 fp16 요소에 한정됩니다).

GeForce FX 하드웨어의 경우, 부동 소수점 렌더 타겟은 텍셀(texel)당 32 비트보다 큰 블렌딩과 부동 소수점 텍스처를 지원하지 않으며, 밉매핑이나 필터링을 지원하지 않습니다.

5.9. 노말맵

GeForce FX, GeForce4, GeForce3 GPU 는 전용 하드웨어를 2 요소 노말맵의 노말맵 반구체 매핑을 할수있게 추가했습니다. 이 GPU 의 경우, CxV8U8 형식을 사용해야 합니다. 이 부분의 경우 쉐이더에서 Z 를 유도하는 것보다 이 형식을 사용하는 것이 더 빠를 수도 있기 때문입니다.

노말맵에 관한 자세한 내용은 http://developer.nvidia.com/object/bump_map_compression.html에 있는 Bump Map Compression 백서를 참조하십시오.

5.10. 신형 칩과 아키텍처

GeForce FX 아키텍처는 이제 다양한 모델과 가격대에서 만날 수 있습니다. 이 계열의 모든 제품은 동일한 버텍스와 픽셀 쉐이딩 기능을

갖고 있습니다. 유일한 차이점은 개발자 눈에는 보이지 않는 내부 성능 기능입니다. 이 기능 덕분에 개발자들은 가령 유일한 기학학적 구조 LOD 나 화면 해상도를 통해 확장성을 처리하는 방식으로 GeForce FX 이상을 목표로 하여 게임을 설계할 수 있습니다.

GeForce 6 시리즈 GPU 는 훨씬 빠른 float 성능을 갖고 있지만, 높은 성능 때문에 half 를 계속 지원하고 있습니다. 또한 이 GPU 는 float/half 유형, 부동 소수점 렌더 타겟, 고정 소수점 텍스처와 비교되는 부동 소수점 텍스처의 경우 더욱 수직을 이룹니다.

5.11. 요약

GeForce FX 와 GeForce 6 시리즈 아키텍처는 긴 쉐이더 프로그램에서부터 진정한 미분 계산에 이르기까지 업계에서 가장 유연한 쉐이더 능력을 갖고 있습니다. 그러나 GeForce FX 하드웨어의 경우, 순수한 부동 소수점 쉐이더는 고정 소수점 쉐이더와 부동 소수점 쉐이더의 조합만큼 빠르게 작동하지 않습니다.

대부분의 쉐이더에서, GeForce FX 아키텍처에서 최상의 성능을 실현하는 가장 좋은 방법은 `ps_1_*`와 `ps_2_*` 쉐이더를 합쳐서 이용하는 것일 수도 있습니다. 예를 들어, 픽셀당 조명의 경우 `ps_1_1` 쉐이더에서 발산 조명 조건을, `ps_1_4` 또는 `ps_2_0` 쉐이더에서 또 다른 패스의 정반사 조건을 이용하는 것이 더 빠를 수도 있습니다.

Chapter 6.
일반 도움말

이 장에서는 여러 개의 GPU 계열에서 활용해볼 수 있는 프로그래밍 GPU 에 관한 일반 도움말을 소개하고 있습니다.

6.1. GPU 확인

과거에는 개발자들이 어떤 GPU 에서 작동하고 있는지 알아보기 위해 (Window 를 통해) GPU 의 장치 ID 를 조회하는 일이 잦았습니다. 역사적으로 장치 ID 는 완만한 증가세를 보였습니다. 그러나, GeForce 6 시리즈 GPU 는 그렇지 않습니다. 따라서 구동 중인 GPU 의 기능을 설정하려면 caps bit(DirectX 의 경우) 또는 확장 스트링(OpenGL 의 경우)에 의존할 것을 여분에게 권합니다. OpenGL 의 렌더러 스트링을 이용하고 있다면, NV40 기반의 칩은 모두 이름에 FX 가 붙는다는 것을 잊지 마십시오("GeForce 6xxx" 또는 Quadro FX 3400"로 칭함).

장치 ID 는 개발자들이 지원 호출을 줄이고자 종종 사용합니다. 장치 ID 를 잘못 다루면 대신에 **지원 호출이 이루어집니다**. 새로운 GPU 를 형성할 때, 많은 애플리케이션이 인식하지 못하고 구동에 실패합니다.

아무리 강조해도 지나치지 않은 한 가지 중요한 사실은 장치 ID 를 인식하지 못하면 **어떠한 정보도 알아내지 못한다는 점입니다**. 하지만 장치 ID 를 인식하지 못한다고 극단적인 조치를 취하지는 마십시오.

유일하게 장치 ID 를 합리적으로 사용하는 것은 ID 를 **인식하면** 그때 조치를 취하는 것입니다. 그리고 여러분도 해결하고자 하는 특수한 기능이나 문제가 있다는 것을 알고 있습니다.

어떤 게임은 GeForce 6 시리즈 GPU 에서 작동하지 않습니다. GPU 를 TNT 급 GPU 로 인식하지 못하거나 장치 ID 를 감지하지 못하기 때문입니다. 칩 중에서 NV4X 세대가 가장 뛰어나기 때문에 이러한 행동은 악몽 같은 상황을 연출하지만, 어떤 게임의 경우 코딩이 미흡하기 때문에 작동하지 않습니다.

장치 ID 또한 cap 과 확장 스트링의 **대용이 아닙니다**. cap 은 여러 가지 원인 때문에 시간의 흐름과 함께 변합니다. 대부분, cap 은 시간이 흐르면 가동되겠지만, 사양이 강화되고 명확해지거나 특정 드라이버나 하드웨어 기능을 유지하는 어려움이나 비용 때문에 해제되기도 합니다.

렌더 타겟과 텍스처 형식 또한 때때로 해제됩니다. 따라서 지원을 반드시 확인하십시오.

장치 ID에 문제가 있을 경우, devrelfeedback@nvidia.com으로 Developer Relations 그룹에 연락하십시오.

현재 모든 NVIDIA GPU의 장치 ID 목록은 http://developer.nvidia.com/object/device_ids.html에 있습니다.

6.2. 하드웨어 그림자 맵

GeForce 3 GPU 이상의 NVIDIA 하드웨어는 OpenGL 과 DirectX 에서 하드웨어 그림자 매핑을 지원합니다. “하드웨어 그림자 매핑”(hardware shadow mapping)은 그림자 맵 깊이 비교와 % 근접 필터링 작업을 특별히 수행하기 위한 전용 특수 트랜지스터가 있다는 것을 의미합니다. 여러분에게 이 기능을 이용할 것을 권합니다. 매우 효율적으로 품질이 더 좋게 필터링한 그림자 맵 모서리를 만들어내기 때문입니다. 전용 트랜지스터는 하드웨어 그림자 매핑을 위해 존재하기 때문에, ps_2_0 이상으로 그림자 매핑 알고리즘을 모방하려고 할 경우 성능과 품질이 떨어집니다.

게임 엔진에 그림자 맵를 이용하고 있을 경우 다음을 참고할 수 있습니다.

- 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
- *Perspective Shadow Maps* from SIGGRAPH 2002
- *Perspective Shadow Maps: Care and Feeding* in *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics* (http://developer.nvidia.com/GPUGems)

Simon Kozlov의 *GPU Gems*에서 "Perspective Shadow Maps: Care and Feeding" 챕터는 실제로 원근 그림자 맵을 사용할 수 없게 하는 몇 가지 개선사항을 설명하고 있습니다. 우리는 Kozlov의 챕터에 있는 개념을 가져다가 자체 엔진에 개념 증명으로 구현했으며, 이 개념이 실제 상황에서도 잘 들어맞는다는 것을 발견했습니다. 이 사례는 SKD 버전 7.0 이상에서 이용할 수 있습니다. http://download.nvidia.com/developer/SDK/Individual_Samples/samples.html 에서 볼 수 있습니다.

Direct X 의 경우, 다음과 같은 방식으로 하드웨어 그림자 맵을 형성할 수 있습니다.

1) 용도 `D3DUSAGE_DEPTHSTENCIL` 로 텍스처를 만듭니다.
2) 형식은 `D3DFMT_D16, D3DFMT_D24X8` (또는 `D3DFMT_D24S8` 이지만, 쉐이더에서 스텐실 비트를 이용할 수 없습니다)여야 합니다.
3) `IDirectDrawSurface9` Interface 를 얻기 위해 `GetSurfaceLevel(0)`을 이용합니다.
4) `SetDepthStencilSurface()`에서 표면 지시기를 Z 버퍼로 설정합니다.
5) DirectX 의 경우, 컬러 렌더 타겟도 설정해야 하지만, `D3DRS_COLORWRITEENABLE` 를 제로로 설정해서 컬러 쓰기를 해제할 수 있습니다.
6) 그림자 투사 기하학을 그림자 맵 z 버퍼로 렌더링합니다.
7) 이 과정에서 쓰인 오프더뷰 투영 행렬을 저장합니다.
8) 렌더 타겟과 z 버퍼를 메인 화면 버퍼로 전환합니다.
9) 그림자 맵 텍스처를 샘플러에 고정하고 텍스처 좌표를 다음과 같이 설정합니다.

```
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
```

행렬은 CPU 에 연결될 수 있으며, 연이은 변형은 버텍스 쉐이더에 또는 고정 소수점 파이프라인의 텍스처 행렬을 이용하여 적용할 수 있습니다.

10) 고정 함수 파이프라인이나 ps_1.0-1.3 를 이용할 경우, 투영 플래그를 `D3DTTFF_COUNT4 | D3DTTFF_PROJECTED` 으로 설정합니다.

11) . 픽셀 쉐이더 1.4 이상을 이용할 경우, 그림자 맵 샘플러에서 투영된 텍스처 꺼내기를 수행합니다.

12) 하드웨어는 그림자 맵 텍스처 좌표의 투영된 x 와 y 좌표를 이용하여 텍스처를 검색할 것입니다.

13) 그림자 맵의 깊이 값을 텍스처 좌표의 투영된 z 값을 비교할 것입니다. 텍스처 좌표 깊이가 그림자 맵 깊이보다 클 경우, 꺼내기 작업에서 얻은 결과는 0 일 것입니다(그림자의 경우). 그렇지 않으면 그 결과는 1 일 것입니다.

14) 그림자 맵 샘플러를 위해 를 작동할 경우, 하드웨어는 높이 비교를 4 차례 수행하고 한 개 샘플과 **같은 비용으로** 겹선형으로 결과를 여과할 것입니다. 이렇게 하면 보기가 더 좋아집니다.

15) 조명으로 조절하기 위해 이 값을 이용합니다.

초기 NVIDIA 드라이버(버전 45.23 이하)는 그림자 맵이 투영될 예정이라는 것을 은연 중에 나타냈습니다. 이 행동은 NVDIA 드라이버 52.16 이상과 함께 바뀌었습니다. 프로그래머들은 이제 적절한 텍스처 단계 플래그를 분명하게 설정해야 합니다. 특히, ps.2.0 쉐이더 모델에 그림자 맵을 이용하려면, 투영 텍스처 검색(예를 들어, `tex2Dproj(ShadowMapSampler, IN.TexCoord0).rgb`)을 분명하게 시작해야 합니다. w-드라이버를 직접 모방해서 동일한 명령어를 모방하고 그런 다음 비투영 텍스처 검색을 모방하는 것은 효과적이지 않습니다! 예를 들어, `tex2D(ShadowMapSampler, IN.TexCoord0/IN.TexCoord0.w)`는 효과가 없습니다.

이와 마찬가지로, ps1.1-1.3 쉐이더 모델을 이용할 때에는 드라이버 버전 52.16 이상의 경우 그림자 맵을 시험하는 텍스처 단계(예를 들어, `SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_PROJECTED)`를 위해 투영 플래그를 분명하게 설정해야 합니다.

주: ForceWare 61.71 이상에서는 어떠한 텍스처 명령(`tex2D`, `tex2Dlod` 등)도 그림자 맵과 함께 정확하게 작동합니다.

SDK에는 DirectX와 OpenGL에 하드웨어 그림자 매핑을 설정하는 간단한 보기가 포함되어 있습니다. http://download.nvidia.com/developer/SDK/Individual_Samples/samples.html 에서 볼 수 있습니다.

Chapter 7.
NVIDIA SLI 및 멀티 GPU 성능 정보

이 장에서는 애플리케이션을 NVIDIA의 SLI 기술과 같은 멀티 GPU 구성으로 작동시킬 때 최대의 성능을 발휘할 수 있게 하는 몇 가지 유용한 정보를 소개하고 있습니다. 자세한 내용은ftp://download.nvidia.com/developer/presentations/2004/GPU_Jackpot/SLI_and_Stereo.pdf에서도 확인할 수 있습니다.

7.1. SLI 란 무엇인가?

"Scalable Link Interface"의 약자인 SLI 는 여러 개의 GPU 가 함께 작동하여 더 높은 성능을 실현할 수 있게 해주는 기술입니다. SLI 인증을 받은 마더보드는 x16 물리적 레인이 2 개인 PCI Express 마더보드입니다. 슬롯 각각에는 PCI Express 그래픽 보드가 들어갑니다. 그래픽 보드 두 개를 외부 SLI 브리지 커넥터를 통해 연결하면, 드라이버가 이 구성을 인식하고 여러분은 SLI 멀티-GPU 모드를 시작할 수 있습니다. SLI 멀티-GPU 모드에서 드라이버는 2 개 보드를 모두 단일 장치로 구성합니다. 2 개의 GPU 가 모든 그래픽 애플리케이션의 단일한 논리 장치 같이 보이기 때문입니다.

이 단일 논리 장치는 단일 GPU 와 비교하여 최대 1.9 배 빠르게 작동합니다. 드라이버가 렌더링 부하를 2 개의 물리적 GPU 에 나누기 때문입니다. SLI 모드로 작동하면 이용 가능한 비디오 메모리가 2 배로 늘어나지 않는다는 사실에 주의하십시오. 예를 들어, 256MB 그래픽 보드 두 개를 꽂아도 여전히 최대 256MB 의 비디오 메모리를 가진 장치가 될 뿐입니다. 왜냐하면 일반적으로 드라이버는 GPU 두 개에 모든 비디오 메모리를 중복하기 때문입니다. 다시 말해, 언제라도 GPU 0 의 비디오 메모리에는 GPU 1 의 비디오 메모리와 동일한 데이터가 들어 있습니다.

SLI 시스템에서 애플리케이션을 작동하면, 드라이버가 호환성 모드, 교차 프레임 렌더링(AFR) 또는 SFR 모드 중에 어떤 모드에서 작동할지 결정합니다.

호환성 모드는 GPU 한 개만 이용하여 모든 것을 렌더링합니다(즉, 두 번째 GPU 는 항상 유휴 상태입니다). 이 모드에서는 성능 개선을 볼 수 없지만, 애플리케이션이 SLI 와 호환됩니다.

AFR 의 경우, 드라이버는 GPU 0 의 프레임 n 과 GPU 1 의 프레임 n+1 을 전부 렌더링합니다. 프레임 n+2 는 GPU 0 등에서 렌더링합니다. 각 프레임이 독립적(즉, 프레임이 데이터를 거의 또는 전혀 공유하지 않음)인 한, AFR 은 최대로 효율적입니다. 버텍스별 작업, 래스터화 작업, 픽셀별 작업과 같은 모든 렌더링 작업이 GPU 에 공평하게 나뉘어 있기 때문입니다. 일부 데이터를 프레임끼리 공유할 경우(예를 들어, 이전에 렌더링한 텍스처를 재사용함), GPU 들이 서로 데이터를 전송해야 있습니다. 이 데이터 전송은 완벽한 2 배속을 방해하는 커뮤니케이션 오버헤드를 성립합니다.

SFR 의 경우, 드라이버는 프레임의 최상위 부분을 GPU 0 에, 최하위 부분을 GPU 1 에 할당합니다. 프레임에서 GPU 1 의 이용이 적으면, 부하량을 기준으로 프레임의 최상위와 최하위 부분의 크기를 조정합니다. 최상위 부분이 최하위 부분보다 렌더링 작업이 적으면 두 GPU 2 개가 똑같이 바쁠 수 있도록 드라이버가 최상위 부분의 크기를 확대하기 때문입니다. GPU 0 과 GPU 1 각각의 최상위 부분과 최하위 부분에 화면을 고정하면 두 GPU 의 프레임에서 모든 버텍스를 처리하는 것을 피하려고 하는 것입니다.

SFR 모드는 가령 텍스처로 렌더링하는 작업에 대한 데이터 공유를 여전히 필요로 합니다. AFR 은 일반적으로 커뮤니케이션 오버헤드가 적고 버텍스 부하 균형이 좋기 때문에 사람들이 더 선호합니다. 그러나 가령 애플리케이션이 2 이하로 버퍼링한 프레임의 최대 수를 제한한다면, 가끔 AFR 은 적용에 실패합니다.

개발 중인 애플리케이션은 현행 드라이버(66.93 이상)의 경우 호환성 모드를 기본으로 설정합니다. 애플리케이션 개발자들은 드라이버에, 즉 디스플레이 드라이버 제어판에 애플리케이션 프로파일을 작성함으로써 SFR 모드를 강제 실행하고 GeForce 탭을 클릭하고 “성능 및 품질 설정”을 선택하며 “프로파일 추가”를 클릭한 다음 애플리케이션의 실행명을 입력합니다. 그런 다음 “고급 설정 보기”를 실행하여 “멀티-GPU”에 체크 표시합니다.

미래의 드라이버는 더욱 간편하게 이용할 수 있는 제어판과 API 옵션을 추가하여 다양한 SLI 모드를 강제 실행합니다.

다음은 SLI 시스템의 성능을 최대한 증대하기 위해 해야 될 것과 하지 말아야 할 것들을 모아 놓은 것입니다.

7.2. CPU 병목현상 피하기

애플리케이션이 CPU 위주이면, 더 강력한 그래픽 솔루션에서 애플리케이션을 작동해도 성능에 거의 또는 전혀 영향을 미치지 않습니다. 멀티-GPU 구성을 활용하려면 CPU 로 인해 장애가 되는 것을 피해합니다. 본 프로그래밍 설명서의 2 장은 CPU 장애를 발견하고 피하는 방법을 안내하고 있습니다.

이와 마찬가지로 애플리케이션이 프레임 속도를 임의의 고정 값으로 제한한다면, 프레임 속도는 그 값을 초과할 수 없습니다.

7.3. VSync 해제(기본 설정)

VSync 를 가동하면 인위적으로 프레임 속도를 모니터 재생 속도로 제한하게 됩니다. 멀티-GPU 구성은 그래픽 성능이 뛰어나기 때문에 모니터 재생 속도보다 (훨씬) 빠른 프레임 속도를 실현할 것입니다. 따라서 VSync 를 해제할 경우에만 이 프레임 속도를 얻을 수 있습니다.

삼중 버퍼링은 프레임 속도를 높이기 위한 해결책이 아닙니다. 삼중 버퍼링은 그래픽스 어댑터가 렌더링할 수 있는 추가적인 백 버퍼를 할당할 뿐입니다.

삼중 버퍼링의 경우, 그래픽스 어댑터는 순환순서 방식으로 최대 3 개의 버퍼로 렌더링할 수 있습니다. 그러나 그래픽스 어댑터가 모니터 재생 속도보다 빠른 속도로 꾸준히 버퍼마다 렌더링을 완료한다면, 백 버퍼의 수는 아무 관련성이 없습니다. 그래서 모니터 재생 속도가 전체 프레임 속도를 계속 통제합니다.

거기다 삼중 버퍼링은 2 가지 중요한 단점이 있습니다.

1. 더 많은 비디오 메모리를 차지합니다(화면 해상도가 높고 안티앨리어싱을 작동한 경우, 추가적인 비디오 메모리의 양이 상당함).
2. 렌더링 명령을 보낸 다음 그 명령이 화면에 나타나기까지 랙(시간 지체)이 증가합니다.

따라서 기본 설정상 VSync 를 해제하는 것이 유일한 해결책입니다. DirectX 에서 그렇게 하려면 `IDirect3D9::CreateDevice()`를 호출할 때 `D3DPRESENT_PARAMETERS` 구조의 `PresentationInterval` 부분을 `D3DPRESENT_INTERVAL_IMMEDIATE` 로 설정해야 합니다.

7.4. 최소 프레임 2 개로 랙(Lag) 제한

DirectX 에서는 드라이버가 최대 3 개의 프레임을 버퍼링할 수 있습니다. CPU 와 GPU 가 서로 독립적으로 작동하여 성능을 최대한 실현할 수 있도록 프레임을 버퍼링하는 것이 바람직합니다. 한편, 프레임을 많이 버퍼링할수록 명령어를 보낸 다음 그 결과를 화면상으로 보기까지 시간이 더 길어집니다. 이러한 시간 지체는 보통 바람직하지 않습니다. (테스트 시나리오 따라 달라지지만) 인간이 감지하고 거부감을 느낄 수 있는 시간 지체가 불과 30ms 이기 때문입니다.

그래서 일부 게임에서는 버퍼링하는 프레임 수를 인위적으로 제한합니다. 예를 들어, 백 버퍼를 잠그면, CPU 와 GPU 사이에 심한 동기화가 강제 실행됩니다. 우선 백 버퍼를 잠그면 CPU 가 중지되고

모든 버퍼가 비워지고 그런 다음 GPU 가 중지됩니다. 끝내 시스템이 작동하지 않고 버퍼링하는 프레임 수는 제로가 됩니다.

이런 식으로 시스템을 중지하면 성능상 심한 불이익이 있으므로, 특히 멀티-GPU 구성에서는 피해야 합니다.

버퍼링하는 프레임의 수를 제한하는 것과 관련하여 그나마 괜찮은 해결책은 각 프레임 끝에 있는 명령어 스트림(예: DirectX 이벤트 조회)에 토큰을 삽입하는 것입니다. 추가적인 렌더링 명령어를 보내기 전에 이러한 이벤트가 소비되었다는 것을 확인하면 버퍼링하는 프레임의 수가 제한되고, 이 때문에 1~3 개 프레임에서 지체됩니다.

멀티-GPU 시스템은 특히 버퍼링하는 프레임의 수를 제한하는 데 민감합니다. 일반적으로, n 개의 GPU 를 갖고 있는 시스템은 최소한 n 개의 프레임이 최대한 효율적으로 버퍼링하는 것을 요구합니다.

놀랍게도 그렇게 하면 랙 이 증가하지 않습니다. 듀얼 GPU 시스템이 일반적으로 단일 GPU 시스템의 2 배 빠르기로 작동하기 때문입니다. 예를 들어, 듀얼 GPU 시스템에서 2 개 프레임을 버퍼링하면(각기 렌더링에 15ms 소요), 단일 GPU 시스템의 1 개 프레임에서 버퍼링(각기 렌더링에 30ms 소요)하는 것과 같은 30ms 의 지체가 발생합니다.

그래서 우리는 애플리케이션에서 GPU 를 몇 개 이용할 수 있는지 확인하고 필요할 경우 버퍼링하는 프레임의 수를 최소한 GPU 의 개수로 제한할 것을 권합니다. nvCPL.dll API 는 시스템이 SLI 모들에 있는지 조회가 가능하며, 현재 GPU 를 몇 개 사용하고 있는지 알 수 있습니다. 특히,
NvCplGetDataInt(NVCPL_API_NUMBER_OF_SLI_GPUS, &number)
함수는 시스템에서 SLI 를 장착한 GPU 의 수를 내보냅니다.
NVControlPanel_API.pdf 문서와 NVSDK 샘플 NvCpl 에 자세한 내용이 있습니다.

7.5. 모든 프레임에 있는 모든 렌더 타겟 텍스처 업데이트

멀티-CPU 시스템의 효율성은 GPU 의 데이터 공유 수준에 반비례합니다. 조건이 가장 좋을 경우, GPU 는 데이터를 공유하지 않으며, 따라서 동기화 오버헤드가 없으며, 따라서 효율성이 최대로 발휘됩니다. 공유 데이터의 양을 최소화하려면, 렌더링한 각 프레임을

이전의 모든 프레임과 독립적이어야 합니다. 특히, 텍스처로 렌더링하는 기법을 이용할 때 프레임에 쓰인 모든 렌더 타겟 텍스처를 동일한 프레임 내에서 생성하는 것이 바람직합니다. 거꾸로, 한 프레임씩 걸러서 렌더 타겟을 업데이트하되 모든 프레임의 텍스처와 똑같은 렌더 타겟을 사용하는 것은 피하십시오.

애플리케이션이 단일 GPU 시스템의 렌더링 속도를 높이기 위해 렌더 타겟 업데이트를 명백하게 거른다면, 멀티-GPU 구성의 알고리즘을 수정하면 도움이 될지도 모릅니다. 가령, 애플리케이션이 여러 GPU 와 함께 작동하고 있는지 감지하고, 그렇다면 프레임마다 렌더 타겟을 업데이트하거나(즉 시각적 충실도를 높임) 2 개 프레임 연이어 렌더 타겟을 업데이트한 다음 2 개 프레임 연이어 업데이트를 건너뜁니다.

또는, 일찍부터 렌더 타겟으로 렌더링하고 나중에 프레임에서 결과를 이용하는 것 또한 SLI 시스템에 이롭습니다. 그러면, 1 개 GPU 가 다른 GPU 의 텍스처 렌더링 작업 결과를 기다리는 일이 없어집니다.

7.6. 렌더 타겟과 프레임 버퍼 초기화

하드웨어는 애플리케이션이 렌더 타겟의 모든 픽셀을 업데이트할지, 렌더 타겟이 사용 전에 초기화되지 않았는지 알 수 없기 때문에, 렌더링한 결과를 멀티-GPU 시스템(0 항과 유사)의 GPU 끼리 공유할 필요가 있을 것입니다. 사용 전에 렌더 타겟을 초기화하면, 동기화가 필요하지 않다는 것을 드라이버와 하드웨어가 알 수 있습니다.

7.7. D3DPOOL_MANAGED 에 버텍스 버퍼 할당

0 항과 비슷하게, 멀티-GPU 시스템은 여러 개의 GPU 간에 버텍스 버퍼를 공유합니다. `D3DPOOL_MANAGED` 에서 DirectX 아래 버텍스 버퍼를 할당하면, `D3DPOOL_DEFAULT` 의 할당과 비교하여 관련 오버헤드가 줄어듭니다. 이 오버헤드 감소는 동적 버텍스 버퍼에 효과적이며, 부분적으로 업데이트된 동적 버텍스 버퍼의 경우에 특히 효과적입니다.

Chapter 8.
입체 게임 개발

이 장에서는 NVIDIA 의 스테레오 렌더링 구현의 과정과 이것을 애플리케이션에 100% 활용하는 방법을 설명하고 있습니다.

8.1. 왜 스테레오에 신경쓰는가?

게임 개발자들은 게임의 사실주의 제고에 대한 영원한 탐구에서 한 가지 요인을 간과하기 십상입니다. 사람들은 현실 세계를 두 눈으로 바라본다는 점입니다. 인공 입체 시청(화면과 실생활에서)은 거대한 시장이 아니지만, 많은 게이머들은 NVIDIA 의 스테레오 오버라이드 드라이버와 함께 저렴한 특수 고글을 쓰고 게임을 하면서 또 다른 존재감을 즐깁니다.

게다가, 개발 중에 스테레오로 게임을 보면 이점이 있습니다. 거짓처럼 보이는 것을 즉시 골라낼 수 있습니다. 운동 시차는 스테레오에 유사한 시각적 힌트를 주지만 움직이면서 깊이 운동 시차를 통해 정보를 얻을 경우 스테레오 뷰어는 사용자가 보고 있는 것을 바로 인식합니다.

게임으로 나오기 전에 보면서 바로 시각적 결함을 수정할 수 있다는 점에서, 게임을 개발하면서 입체 시청을 이용하는 것은 경쟁 우위라 할 수 있습니다. 그러면 물론 스테레오로 게임을 하는 사람들의 경험이 한층 나아질 것입니다.

8.2. 스테레오의 작동 방식

NVIDIA 3D Stereo Driver 는 DirectX 와 OpenGL 기반 게임의 풀스크린 입체 시청을 허용합니다. 3D Stereo Driver 는 특수 고글에 적합한 페이지를 넘기면서 보는 시청뿐만 아니라 빨간 색과 파란 색으로 된 2 색 입체 사진을 지원합니다. 호환 가능한 하드웨어를 장착하면, 깊이를 감지하면서 이미지를 볼 수 있을 것입니다. Stereo Driver 버전은 작능하려면 Display Driver 버전과 맞아야 한다는 데 주의하십시오.

입체드라이버를 가동한 상태에서 3D 게임을 하면, 화면을 2 개 시선에서 렌더링할 수 있을 것입니다. 시선이 좌우 양쪽 눈에서 나오는 것처럼, 각 시선은 실제 시선에서 약간 측면으로 비켜가 있습니다. 이것은 고정 함수 렌더링과 버텍스 쉐이더와 함께 작동합니다.

정확한 스테레오 효과를 위해서, 개발자들은 아래 나열된 여러 가지 문제를 고려해야 합니다.

8.3. 스테레오를 방해하는 것들

여기에 스테레오에 부정적인 영향을 미치는 일반적인 문제를 나열하고 이 문제를 해결하는 방법을 제시했습니다.

8.3.1. 정확하지 않은 깊이에서 렌더링

이 문제는 제일 먼저 처리해야 하는 사항입니다. 3D Stereo Driver 는 깊이를 이용하여 스테레오 효과를 창출하므로, 정확한 깊이에서 하지 않은 렌더링을 하면 스테레오 상태에서 시청할 때 보기 싫게 두드러져 보입니다.

- 배경 이미지, 개인관람석, 스카이 돔을 깊이상 가장 먼 곳에 둡니다. 그렇지 않으면, 세상이 스테레오의 작은 상자에 있는 것처럼 보일 것입니다.
- HUD 항목을 적절한 3D 깊이에 둡니다. 객체를 맴도는 이름표가 있어서 객체의 3D 깊이에 이름표를 두면, 뷰 프러스트럼(view frustrum)의 가까운 평면에 두는 것보다 스테레오 효과가 좋습니다.

- HUD 를 되도록 화면의 먼 곳에 렌더링하는 것 또한 유용합니다. 이 방법을 쓰면, 나머지 화면에서 감지되는 깊이가 더 커지며 HUD 를 볼 때 눈에 피로가 생기지 않습니다.
- 지시하고 있는 객체 깊이의 3D 세계에 있지 않는 한, 레이저 포인터, 십자선, 커서는 정확하게 보이지 않습니다. 3D 세계에 없으면, 사용자의 눈은 하나의 깊이에 모이지만 커서는 다른 깊이에 있기 때문에 이 물건들을 사용하는 것이 거의 불가능합니다. 사용자는 2 개의 커서를 보지만, 2 개의 커서 중 어느 것도 정확한 위치를 가리키지 않습니다.
- 객체를 강조하는 것은 화면 공간이 아니라 객체 자체의 깊이에서 발생해야 합니다.

8.3.2. 게시판 효과

게시판 효과는 표준 3D 에서 평평하고 안 좋아 보입니다. 스테레오 상태에서는 더 안 좋아 보입니다. 표준 3D 에서는 움직이다 면 게시판이 보이지만, 3D Stereo 에서는 정지된 화면에서도 문제가 바로 나옵니다. 대부분의 게시판 효과는 스테레오 상태에서 활기가 없어 보입니다. 그래서 대신 될 수 있는 한 온갖 곳에다 기하학을 이용합니다. 해상도가 낮은 기하학이라도 좋아보입니다.

입자 효과 게시판(스파크, 스모크, 먼지 등)의 경우, 괜찮아 보일 수도 그렇지 않을 수도 있습니다. 가장 좋은 것은 스테레오 상태로 애플리케이션을 시험해서 어떤지 보고 품질이 양호한지 판단한 다음 게시판이 3D 의 의미있는 깊이를 갖고 있는지 확인하는 것입니다.

8.3.3. 후처리 및 화면 공간 효과

2D 화면 공간 효과는 스테레오 효과에 크게 지장을 줄 수 있습니다. 흐릿한 광채, 블룸 필터, 이미지 중심의 모션 블러와 같은 것들은 이 항목에 속합니다. 이러한 효과는 보통 3D 기하학을 텍스처로 렌더링한 다음, 화면에 쿼드로 배열된 2D 화면을 렌더링하면서 생깁니다. 텍스처의 기하학은 실제와 달리 깊이가 더 이상 정확하지 않습니다. 그래서 3D Stereo 에서 제대로 작동하지 않는 것입니다.

여러분은 스테레오 상태에서 게임을 즐기는 사람들을 위해 이 효과를 해제할 수 있는 선택권을 제공하고 기하학을 백 버퍼에 렌더링해야 합니다.

8.3.4. 3D 화면에 2D 렌더링 사용

2D 로 렌더링한 객체는 사실 실제 3D 깊이를 갖고 있지 않습니다. 그래서 모니터 깊이에 위치하고 있습니다. 그러면 3D Stereo 에서 매우 평평하게 보일 것입니다. 2D 와 3D 를 HUD 에 혼합한다면, 깊이가 들쭉날쭉해서 눈이 피로해질 것입니다. 다시 말하지만 모든 것을 적절한 깊이에서 3D 로 렌더링해서 스테레오를 작동한 상태에서 테스트하십시오.

8.3.5. 서브뷰 렌더링

전시회 그림, 자동차 거울, 또는 상단 구석의 작은 지도와 같이 서브뷰를 화면에 렌더링할 때, 렌더링에 앞서 화상 표시 영역을 설정해야 합니다. 이런 방법을 쓰면, 본래의 화면 분할을 벗어난 이상한 스테레오 효과를 방지할 수 있습니다.

8.3.6. 더티 사각형(Dirty Rectangle)으로 화면 업데이트

화면의 일부분만을 결정해 수정하고 나머지 화면을 수정하지 않으면, 3D 스테레오에서 이상한 렌더링을 초래할 수 있습니다. 프레임마다 눈에 보이는 객체는 모두 렌더링하십시오.

8.3.7. 멀리 떨어뜨려서 충돌 문제를 해결

충돌을 해결하려고 객체들을 서로 떨어뜨려 놓을 때에는 너무 멀리 떨어뜨려 놓지 마십시오. 움직일 때 정상 3D 에서 안 좋게 보이지만 스테레오 상태일 때 두드러져 보여 사물들이 지상을 맴도는 것처럼 보이기 때문입니다.

8.3.8. 화면에서 객체마다 깊이 범위 변경

화면을 여러 개의 깊이 범위로 분할하면 스테레오 효과에 왜국이 생겨 일부 객체가 본래보다 짧거나 길어 보일 수 있습니다. 최상의 스테레오 효과를 위해서는 모든 객체를 일정한 깊이 범위로 렌더링해야 합니다.

8.3.9. 깊이 데이터에 버텍스를 제공하지 않음

소프트웨어 변형과 조명을 위해 렌더링할 버텍스를 D3D 로 보낼 때, 스테레오가 제대로 작동할 수 있도록 RHW 깊이 정보를 넣으십시오.

8.3.10. 윈도 모드에서 렌더링

NVIDIA 의 3D Stereo 는 애플리케이션이 풀스크린 전용 모드에 있을 때에만 작동합니다. 풀스크린 모드를 지원하지 않는다면, 게임 플레이어는 3D Stereo 를 이용할 수 없습니다.

8.3.11. 그림자

풀스크린 음영색 쿼드를 이용하여 스텐실 그림자를 렌더링하면 스테레오에서 제대로 작동하지 않습니다. 그러나 적절한 깊이에서 화면의 음영 객체를 음영색으로 다시 렌더링하면 스테레오 상태에서 제대로 작동할 것입니다. 그림자 맵은 제대로 잘 작동하고 있으며, 투영 그림자는 여러분이 그림자의 적절한 깊이에 투영하고 있는 한 작동합니다.

8.3.12. 소프트웨어 렌더링

NVIDIA 3D Stereo 드라이버는 DirectX 와 OpenGL 을 자동으로 지원합니다. 다른 API 를 이용하여 3D 기하학을 렌더링할 경우, 스테레오로 렌더링되지 않을 것입니다.

8.3.13. 렌더 타겟에 직접 작성

렌더 타겟에 잠금을 설정해서 직접 작성하지 마십시오. 그렇게 하면 스테레오 드라이버를 지나치게 됩니다.

8.3.14. 매우 어둡거나 콘트라스트가 심한 화면

매우 어두운 화면은 3D Stereo 특수 고글을 착용하면 훨씬 더 어두워질 수 있습니다. 밝기와 감마 조정을 제공하면 이 문제를 해결하는 데 도움이 될 것입니다. 매우 밝은 객체와 매우 어두운 객체에서 매우 밝은 객체를 이용하면 잔상이 발생하고, 이 잔상은 스테레오를 방해합니다. 스테레오 상태에서 게임을 테스트하면 이것이 문제인지 여부를 금방 알 수 있습니다.

8.3.15. 버텍스간 격차가 작은 객체

메시에 격차가 작아도 스테레오로 렌더링하면 더욱 뚜렷하게 보일 수 있습니다. 그러므로 메시를 촘촘하게 하고 스테레오 상태에서 테스트를 해서 이런 문제가 있는지 확인하는 것이 좋습니다.

8.4. 스테레오 효과 개선

사람들이 더욱 몰입하게 하는 경험을 형성하기 위해 이용할 수 있는 몇 가지 아이디어를 여기에 모아 놓았습니다.

8.4.1. 스테레오로 게임 테스트

훌륭한 3D Stereo 경험을 얻을 수 있는 최선의 방법은 스테레오를 작동한 상태에서 게임을 테스트하는 것입니다. 대부분의 문제는 바로 분명히 드러나며, 간단하게 수정할 수 있습니다. IODisplay(http://www.i-glasses.com)에서 저렴한 스테레오 키트를 구할 수 있습니다. 그리고 NVIDIA 3D Stereo Driver 는 빨간 색과 파란 색이 있는 2 색 입체 사진 스테레오 모드도 지원합니다. 그래서 3D 종이 안경이 주변에 있으면, 테스트하기가 훨씬 더 쉽습니다.

8.4.2. “Out of the Monitor” 효과 얻기

게임에서 근처 평면에 매우 가까운 사물들을 설계할 수 있지만 뷰 프러스트의 모서리들을 교차하지 마십시오. 이런 설계를 통해 “모니터에서 튀어나오는” 엄청난 스테레오 효과를 얻을 수 있습니다. 궤도, 우주선, 날아다니는 캐릭터 등은 몇 가지 가능성을 바탕으로 합니다. 이 모든 것은 스테레오 상태에서 정말 환상적으로 보입니다.

8.4.3. 정밀한 기하학 구조 이용

3D 에서 사실성을 높이려면 다각형을 더 많이 이용하십시오. 이 전략은 물론 항상 들어맞지만, 스테레오의 경우에는 더욱 그렇습니다. 건물, 식물, 나무, 캐릭터 등 장소를 가라지 않고 가능한 한 모든 곳에 다각형을 이용하십시오!

8.4.4. 대안적인 뷰 제공

사용자에게 게임의 시선에 대한 선택권, 즉 최소한 카메라에 대한 통제권를 제공합니다. 스테레오의 경우, 1 인칭, 3 인칭, 탑다운 등 일부는 다른 것들보다 나아 보입니다.

8.4.5. 게임으로 현 문제 검색

일단 3D Stereo Driver 를 설치하면 제어판에서 "스테레오 게임 구성" 하위 메뉴를 볼 수 있습니다. 이 페이지에서 여러분은 게임이 올라와 있는지 여부와 해당 게임과 관련하여 우리가 어떤 문제를 미리 발견했는지 알 수 있습니다. NVDIA Developer Relations 에 연락하면 문제 해결에 도움이 될 수 있습니다.

8.5. 스테레오 API

현재 우리는 지원 스테레오를 개선하기 위해 2 개의 별도 API 를 개발하고 있습니다.

- **StereoBLT API – 사전에 렌더링한 스테레오 이미지를 3D 로 표시** 표시가 실행되어 있는 동안, 미리 만든 이미지의 표시를 허용합니다.
- **IStereoAPI – 입체 렌더링에 대한 실시간 통제** 수렴, 스테레오 분리, 스테레오 드라이버의 자세한 내용을 허용합니다. 이 API 는 게임에서 이 설정을 실시간으로 통제하기 위해 이용할 수 있는 헤드와 라이브러리로 구성되어 있습니다. 즉시 수정이 발생하고 프레임마다 변경될 수 있습니다.
- **OpenGL 쿼드 버퍼링 스테레오.** 이것은 NVIDIA Quadro GPU 계열에서 이용할 수 있으며, 특수한 스테레오 드라이버를 필요로 하지 않으며 윈도 모드에서도 작동합니다.

8.6. 추가 정보

자세한 내용은 NVIDIA Developer Relations 판매점에 문의하거나 3DStereoDev@nvidia.com로 이메일을 보내 자세한 정보나 발매 전 Stereo API를 요청하십시오.

http://developer.nvidia.com/object/3D_Stereoscopic_Dev.html으로 이동하면 업데이트된 스테레오 정보를 온라인에서 찾을 수 있습니다.

Chapter 9.
성능 툴 개요

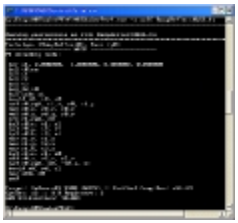
이 장에서는 성능 장애를 파악하고 치유하는 데 도움이 될 만한 몇 가지 툴에 대해 소개하고 있습니다.

9.1. NVPerfHUD

NVPerfHUD 는 DirectX 애플리케이션 상단에 전개된 4 개의 유용한 그래픽을 표시합니다. 이 그래프는 애플리케이션에 관한 중요한 통계 자료를 보여주며, 이 자료는 잠재적인 장애를 파악할 수 있도록 도움을 줍니다. 그래프는 심장 박동 모니터 형식으로 샘플 데이터를 표시합니다. 오른쪽에서 왼쪽으로 스크롤을 움직이면 마지막 256 개 프레임의 값을 볼 수 있습니다.

NVPerfHUD는 NVIDIA Developer 웹 사이트인 http://developer.nvidia.com/object/nvperfhud_home.html에서 얻을 수 있습니다.

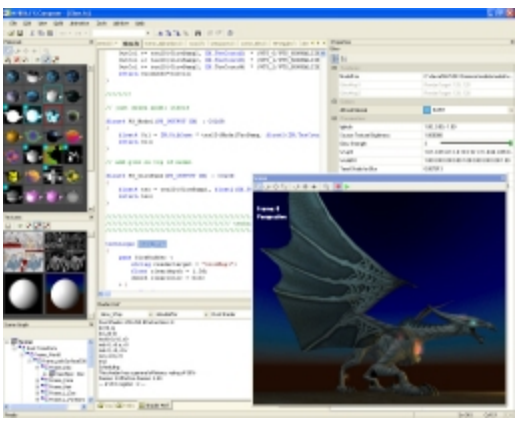
9.2. NVShaderPerf



NVShaderPerf 명령 행 유틸리티는 쉐이더 성능 기준을 장점을 보고하기 위해 FX Composer 의 Shader Perf 패널에 있는 것과 동일한 기술을 이용합니다. 이것은 HLSL, GLSL, Cg, `!!FP1.0`, `!!ARBfp1.0`, `ps_1_x`, `ps_2_x`.으로 작성된 DirectX 와 OpenGL 쉐이더를 지원합니다. 주기 계수, 레지스터 사용, GPU 활용 등급을 비롯하여 GeForce 6 시리즈와 GeForce FX GPU 전체 계열의 쉐이더에 대한 성능 보고서를 받아볼 수 있습니다.

NVShaderPerf는http://developer.nvidia.com/object/nvshaderperf_home.html 에서 얻을 수 있습니다.

9.3. FX Composer



FX Composer 는 고유의 실시간 프리뷰와 최적화 기능을 갖춘 통합된 개발 환경에서 고성능 쉐이더를 만들 수 있는 권한을 개발자들에게 줍니다. FX Composer 는 프로그래머들을 위해 쉐이더 개발과 최적화를 좀더 쉽게 하는 동시에, 특정 화면에 맞춰 쉐이더를 변경하는 아티스트들에게 편리한 GUI 를 제공하자는 목표로 설계한 것입니다.

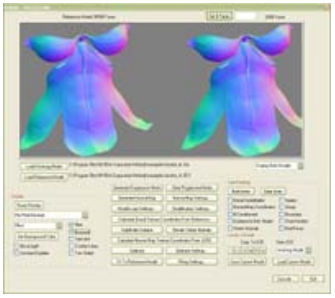
FX Composer 를 이용하면 고급 분석 및 최적화와 더불어 쉐이더 성능을 조율할 수 있습니다.

- 버텍스 쉐이더와 픽셀 쉐이더에 대한 성능 조율 작업을 실행할 수 있습니다.

- GeForce 6 시리즈와 GeForce FX GPU 전체 계열에 대해 성능을 모의 수행합니다.
- 미리 계산한 함수를 텍스처 검사표에 입력합니다.
- GPU 주기 계수, 레지스터 사용, 활용 등급, FPS 와 같은 경험적인 성능 기준을 제공합니다.
- 최적화에서 보이는 징후는 사용자에게 성능 장애를 알려줍니다.

FX Composer의 최신 버전은http://developer.nvidia.com/fxcomposer에서 내려받을 수 있습니다.

9.4. NVIDIA Melody



낮은디테일 모델을 높은디테일 모델처럼 보이게 만드는 저고품질의 노말맵을 만들려면, NVIDIA Melody를 이용하십시오. 낮은디테일 모델을 로딩한 다음 높은디테일 레퍼런스 모델을 로딩하고 "노말 맵 만들기" 버튼을 누른 다음 Melody가 신속하게 작업을 처리하는것을 지켜보기만 하면 됩니다. Melody는 http://developer.nvidia.com/object/melody_home.html에 가면 있습니다

9.5. 개발자 툴에 관한 문의와 의견

툴에 대한 여러분의 의견을 기다리고 있습니다. 의견이나 문제가 있으시면 sdkfeedback@nvidia.com로 메일 주십시오.