NVPerfHUD 를 통한 GPU 성능 최적화

NVPerfHUD 3.0
성능 분석을 위한 빈틈 없는 디스플레이

DU-01231-001_v01
2005 년 4 월

DEVELOPMENT

차례

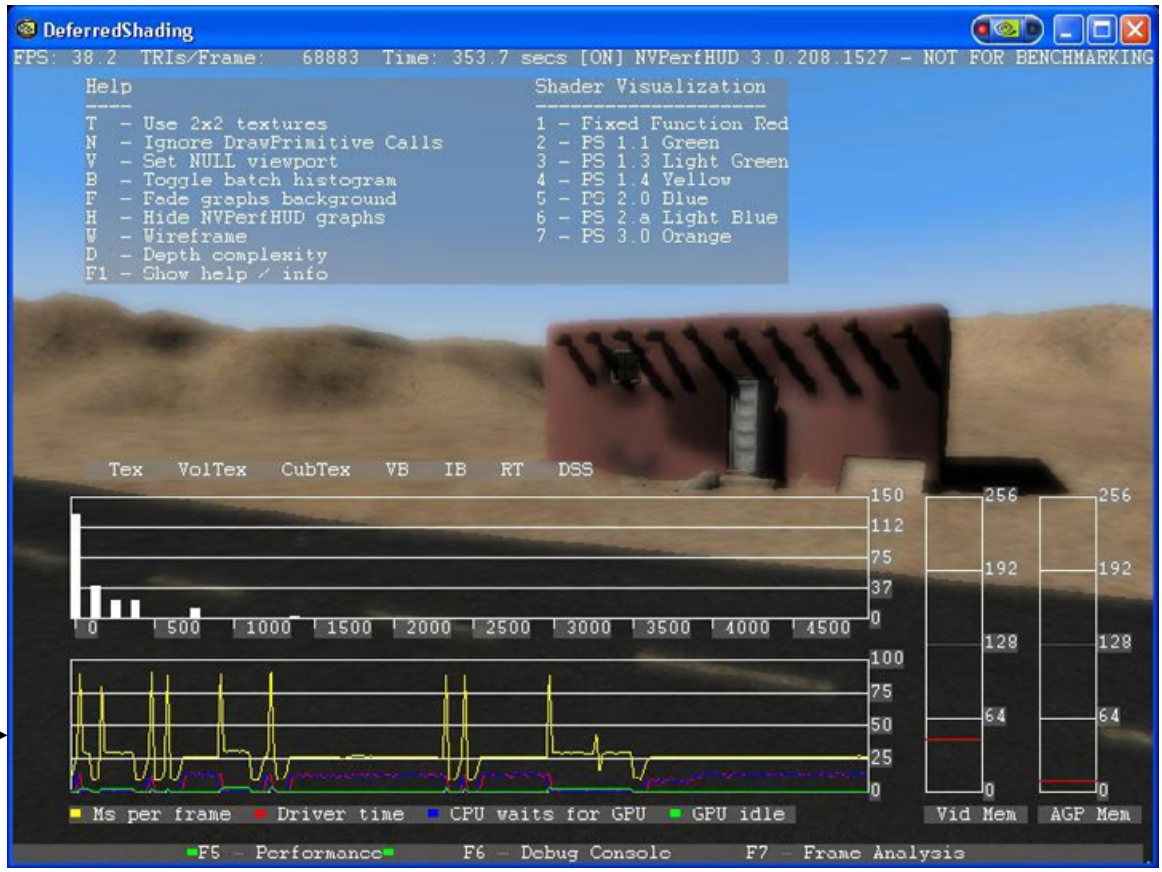
그림 목록

Chapter 1. NVPerfHUD 안내

오늘날의 그래픽 처리 장치(GPU)는 파이프라인 처리된 연산 시퀀스를 통해 이미지를 만들어냅니다. 파이프라인은 가장 느린 스테이지 속도에 맞춰 실행되므로, 최적의 성능을 위해 그래픽 애플리케이션을 조정하려면 파이프라인 기반 접근방식이 필요합니다. NVPerfHUD 는 그래픽 파이프라인의 가장 느린 스테이지를 인식하여 애플리케이션의 전체 성능을 향상시켜 드립니다.

NVPerfHUD 는 한 번에 한 스테이지에서 애플리케이션 성능을 분석할 수 있도록 지원하며 Direct3D 9 애플리케이션의 어떤 스테이지에서도 성능 병목현상 및 기능 문제를 진단할 수 있는 실시간 통계를 표시합니다.

NVPerfHUD 를 활성화하면, 아래 그림 1. NVPerfHUD 를 활성화한 Direct3D 애플리케이션과 같이 애플리케이션 상단에 그래픽 오버레이가 표시됩니다.



성능을 표시하는 그래픽 오버레이

그림 1. NVPerfHUD 를 활성화한 Direct3D 애플리케이션

1.1. 시스템 요구사항

- 모든 종류의 NVIDIA GPU(GeForce 3 이상)
  GeForce 6 시리즈 이상 권장
  기존 GPU 는 축소된 기능을 지원
- NVIDIA 디스플레이 드라이브 71.80 이상
- Microsoft DirectX 9.0c
- Windows 2000 또는 Windows XP

1.2. 권장 참고자료

- NVIDIA 개발자 웹 사이트
  http://developer.nvidia.com
  - *최적의 성능을 위한 그래픽 파이프라인 균형조정* – 백서
    [링크]

- *NVIDIA GPU 프로그래밍 안내서* – 최신 도움말 및 요령 [링크]
- NVIDIA FX Composer – 쉐이더 개발 환경 [링크]
- NVShaderPerf – 쉐이더 성능 분석 유틸리티 [링크]
- NVIDIA SDK – 수백 가지 샘플 및 효과 [링크]
- NVTriStrip – 버텍스 캐시가 인식할 수 있는 스트립을 만듭니다 [링크]

- GPU Gems: *실시간 그래픽을 위한 프로그래밍 기술, 도움말 및 요령* [링크]
  성능 관련 챕터는 특히 유용합니다
- GPU Gems 2: *고성능 그래픽 및 일반 컴퓨팅을 위한 프로그래밍 기술*[링크]
  Microsoft DirectX 웹 사이트 [링크]
- Microsoft Developer Network (MSDN) 웹 사이트 [링크]
  "성능" 및 "최적화" 검색
- Microsoft DirectX SDK 문서 – 설치 후 시작 메뉴에 있음

Chapter 2. 시작하기

NVPerfHUD 는 GPU 및 드라이버 자체에서 수치를 직접 수집하는 디스플레이 드라이버에서 특수 성능 모니터링 루틴을 사용합니다. 또한, NVPerfHUD 는 API 인터셉션으로 다양한 수치를 수집하며 애플리케이션과 상호작용을 합니다. NVPerfHUD 를 적절하게 사용하는 데 필수적인 이러한 계측 기술은 소량의 추가적인 오버헤드를 가져옵니다(7% 이하).

NVPerfHUD 를 사용하여 Direct3D 애플리케이션을 시작하면, NVPerfHUD 사용자 인터페이스가 DirectX 그래픽 상단에 표시되는 것을 확인할 수 있습니다. 작동 단축키를 사용하면 애플리케이션 및 NVPerfHUD 와의 상호작용을 전환할 수 있습니다.

NVPerfHUD 분석을 위해 애플리케이션을 활성화하고, 아래 빠른 시작 절차를 실행하여 NVPerfHUD 를 시작하십시오.

2.1. 애플리케이션 준비

NVPerfHUD 는 애플리케이션의 내부 기능을 이해할 수 있도록 돕는 강력한 성능 분석 도구입니다. 승인 받지 않은 제 3 자가 사용자의 애플리케이션을 허가 없이 분석하지 않도록 하려면, NVPerfHUD 를 약간 수정할 필요가 있습니다. NVPerfHUD 로 애플리케이션을 적절하게 사용하는 방법에 대한 자세한 내용은 이 문서 뒷부분에 수록된 문제해결을 참조하십시오.

> 출시 전에, 애플리케이션에서 NVPerfHUD 분석을 비활성화 상태로 설정했는지 확인하십시오. 그렇지 않으면, 다른 사람이 사용자의 애플리케이션에서 NVPerfHUD 를 사용할 수 있게 됩니다!

그래픽 파이프라인을 설정할 때 제일 먼저 해야 할 작업 중 하나는 Direct3D CreateDevice() 함수를 호출하여 디스플레이 장치를 만드는 것입니다. 애플리케이션에서 다음과 같이 보일 수 있습니다.

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

NVPerfHUD 로 애플리케이션을 시작하면, 특수한 **NVIDIA NVPerfHUD** 어댑터가 만들어집니다. 이 어댑터를 선택하면 NVPerfHUD 에게 애플리케이션을 분석할 수 있는 권한을 주는 것입니다. 그 뿐 아니라, 일부 애플리케이션에서는 **NVIDIA NVPerfHUD** 어댑터 ID 를 무심코 선택하여, 승인 받지 않은 분석을 허용하게 될 수 있으므로 장치 종류를 "D3DDEVTYPE_REF"로 선택해야 합니다. 사용자가 NVPerfHUD 어댑터를 선택하지 않는 한, 애플리케이션에서는 레퍼런스 래스터 장치를 실제로 사용하지 않습니다.

애플리케이션에서 NVPerfHUD 분석을 활성화하는 최소한의 코드 변경을 실시하면 다음과 같이 보일 수 있습니다.

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

(위와 같이 **GetAdapterCount()-1** 을 호출하여) 마지막 어댑터를 사용하면 NVPerfHUD 에서 만든 **NVIDIA NVPerfHUD** 어댑터 ID 가 목록의 맨 마지막에 오는 것으로 가정합니다.

NVIDIA NVPerfHUD 어댑터를 선택하고 **DeviceType** 플래그를 **D3DDEVTYPE_REF** 로 설정하는 두 가지 작업만이 애플리케이션을 위해 NVPerfHUD 분석을 활성화하는 데 필요한 변경사항입니다. 둘 중 한 가지 매개변수만을 변경하면, NVPerfHUD 분석을 사용할 수 없습니다.

알고 있는 것처럼, 이처럼 빠른 구현에는 한 가지 문제가 있습니다. **NVIDIA NVPerfHUD** 어댑터를 사용할 수 없는 경우 레퍼런스 래스터 장치를 사용하게 되어 버린다는 것이다. 이 문제를 예방하기 위해, **CreateDevice()**를 호출하는 부분을 다음과 같은 형식으로 변경할 것을 권장해 드립니다.

```
// 기본 설정
UINT AdapterToUse=D3DADAPTER_DEFAULT;
D3DDEVTYPE DeviceType=D3DDEVTYPE_HAL;

#if SHIPPING_VERSION
// 출시 버전을 구축할 때 NVPerfHUD를 비활성화한다(옵트아웃)
#else
// 'NVIDIA NVPerfHUD' 어댑터를 찾는다
// 어댑터가 있으면 기본 설정을 대체한다
for (UINT Adapter=0;Adapter<g_pD3D->GetAdapterCount();Adapter++)
{
      D3DADAPTER_IDENTIFIER9  Identifier;
      HRESULT                 Res;

      Res = g_pD3D->GetAdapterIdentifier(Adapter,0,&Identifier);
      if (strcmp(Identifier.Description,"NVIDIA NVPerfHUD") == 0)
```

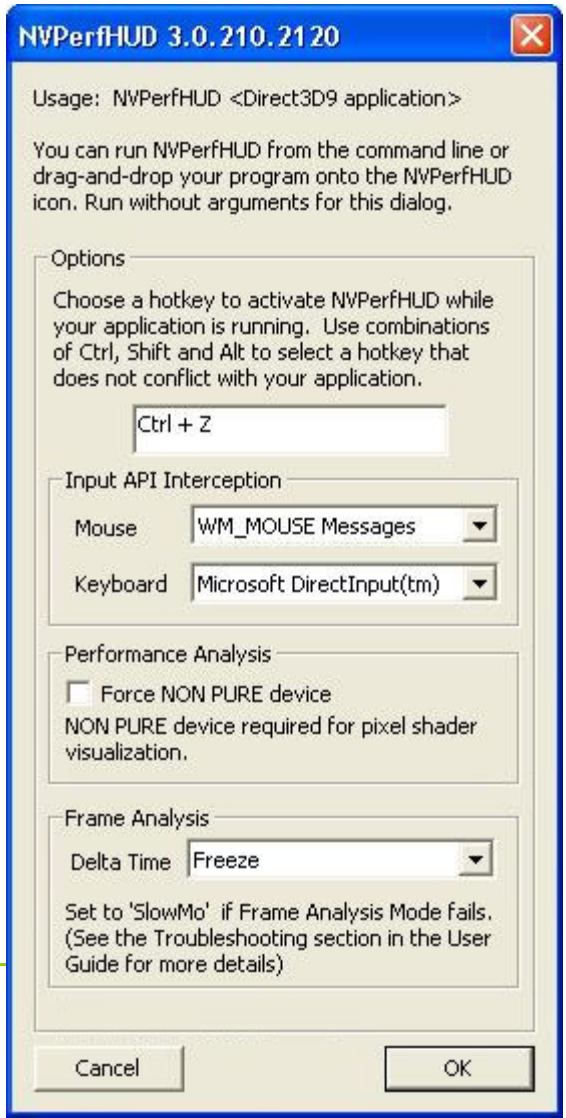
```
        {
                AdapterToUse=Adapter;
                DeviceType=D3DDEVTYPE_REF;
                break;
        }
}
#endif

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

이 코드는 사용을 원할 때 NVPerfHUD 를 활성화 시켜주며, NVPerfHUD 를 설치하지 않았을 때 소프트웨어 레퍼런스 래스터 장치를 사용하게 되는 문제를 예방합니다. NVPerfHUD 로 애플리케이션을 실행하는 방법에 대한 자세한 내용은 아래 빠른 시작 절차를 참조하십시오.

2.2. 빠른 시작

NVPerfHUD 와 함께 애플리케이션을 실행하면, 기본 세트의 그래프와 정보가 애플리케이션 상단에 표시됩니다. NVPerfHUD 의 고급 기능은 아래 설명되어 있는 작동 단축키를 통해 사용할 수 있습니다.



1. **NVPerfHUD 설치**
   설치 프로그램이 바탕화면에 새 아이콘을 만듭니다.
2. **NVPerfHUD 실행**
   NVPerfHUD 를 처음 실행하면 설정 대화 상자가 자동으로 표시됩니다. 분석할 애플리케이션을 지정하지 않고 시작 절차를 실행하면, 언제든지 구성 대화 상자를 볼 수 있습니다.
3. **작동 단축키 선택**
   애플리케이션에서 사용하는 키와 충돌하지 않도록 주의하십시오.
4. **API 가로채기 구성**
   NVPerfHUD 가 마우스 및 키보드 이벤트를 캡처하는 방법을 설정합니다. 애플리케이션에서 지원하지 않는 방법을 사용하면, 키보드 또는 마우스를 사용할 수 없습니다.

작동 단축키로 NVPerfHUD 를 활성화하면, 이어지는 모든 키보드 이벤트를 NVPerfHUD 가 가로채게 됩니다.

5. **선택사항: NVPerfHUD 의 Force NON PURE 장치 활성화.**
애플리케이션에서 PURE 장치를 사용하는 경우에는 **Force NON PURE 장치** 확인란을 반드시 선택해야 합니다. PURE 장치를 사용하는 애플리케이션에서 이 상자를 선택하지 않으면, 성능 모드에서 프레임 분석 모드 또는 픽셀 쉐이더 시각화를 사용할 수 없습니다.

6. **NVPerfHUD 바탕화면 아이콘으로 애플리케이션 끌어다 놓기.**
확인을 클릭하여 구성 옵션을 확인한 후 .EXE, .BAT .LNK(바로가기) 파일을 NVPerfHUD 시작 프로그램 아이콘으로 끌어다 놓습니다. 명령줄에서 NVPerHUD.exe 를 실행하여 명령줄 인수로 분석할 애플리케이션을 지정할 수도 있습니다. 일부 개발자들은 배치 파일 작성 또는 IDE 설정 수정을 선택하므로 이 현상이 자동으로 나타납니다.

7. **선택사항: 애플리케이션이 충돌하거나 프레임 분석 모드에 문제가 있는 경우에는 "Delta Time" 설정을 SlowMo 로 변경하십시오.** 이를 통해, 문제가 NVPerfHUD 또는 애플리케이션 중 어디에 있는지 확인할 수 있습니다. 자세한 내용은 문제해결 부분을 참조하십시오.

애플리케이션을 지정하지 않고 NVPerfHUD 를 실행하면 구성 대화 상자에 언제든지 액세스할 수 있습니다.

NVPerfHUD 기본 세트의 그래프와 정보가 애플리케이션 상단에 표시되면서 애플리케이션이 실행되는지 확인해야 합니다. 선택한 작동 단축키를 사용하여 NVPerfHUD 와 상호작용을 수행하며, **F1** 을 눌러 화면 도움말을 표시합니다. 애플리케이션으로 돌아가려면 단축키를 다시 누르십시오.

NVPerfHUD 는 PresentationInterval 을 D3DPRESENT_INTERVAL_IMMEDIATE 로 설정하여 수직 리프레시 동기화를 해제합니다. 이를 통해, 애플리케이션에서 병목을 정확하게 확인할 수 있습니다.

2.3. 기본 워크플로우

NVPerfHUD 를 구성하면 그래픽 파이프라인 실험, 성능 수치 그래픽 표시를 수행할 수 있으며, 여러 가지 성능 시각화 모드를 통해 잠재적인 문제를 발견할 수 있습니다. 기본 성능 분석 모드를 디버그 콘솔 모드 또는 프레임 분석 모드로 전환할 수도 있습니다. 디버그 콘솔은 DirectX Debug 런타임의 메시지와 NVPerfHUD 의 경고, 애플리케이션의 커스텀 메시지를 표시합니다. 프레임 분석 모드에서는 그래픽 파이프라인의 각 스테이지에 대한 세부사항을 보여주는 고급 상태 검사기에 액세스할 수 있습니다.

다음 키를 눌러 NVPerfHUD 를 활성화하면 각 모드로 전환할 수 있습니다.

F5 성능 분석
타이밍 그래프와 직접 실험으로 병목을 확인합니다.

F6 디버그 콘솔
DirectX Debug 런타임의 메시지, NVPerfHUD 경고, 애플리케이션의 커스텀 메시지를 검토합니다.

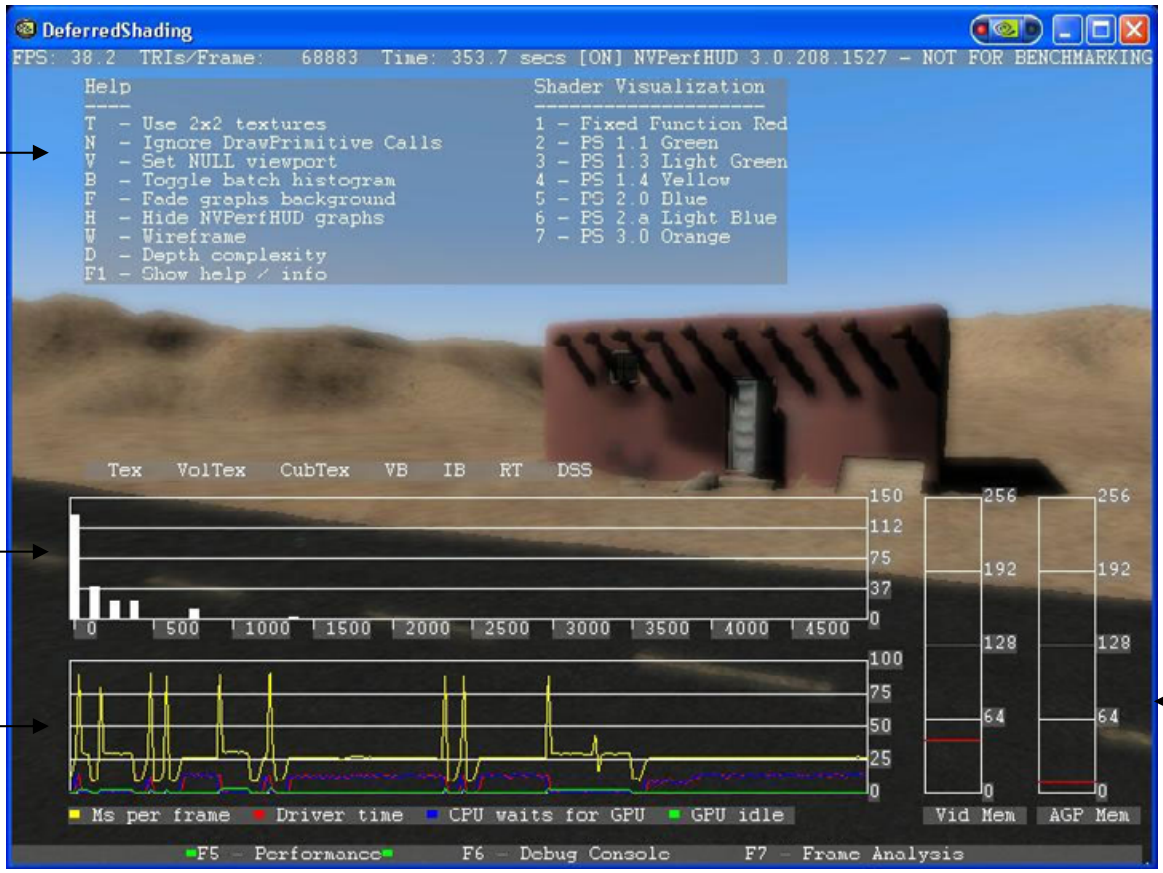
F7 프레임 분석
아래 설명한 상태 검사기로 그래픽 파이프라인의 각 스테이지를 조사하여 현재 프레임을 중단하고 한 번에 장면 하나에 대한 그리기 요청을 단계별로 처리합니다.

Chapter 3.
성능 분석 모드

여기에서는 성능 분석 모드에서 NVPerfHUD 가 표시하는 정보를 해석하는 방법을 설명하며, 대상 성능 실험을 어떻게 실시하여 애플리케이션의 성능 병목현상을 찾아 최적화하는지 설명합니다.

그림 2. NVPerfHUD 성능 분석 모드

는 성능 분석 모드에서 사용할 수 있는 그래프와 오버레이를 나타냅니다.



유용한 정보
(F1 – 전환)

배치 그래프

타이밍
그래프

메모리
그래프

그림 2. NVPerfHUD 성능 분석 모드

3.1. 성능 그래프

애플리케이션을 처음 시작하면 NVPerfHUD 는 여러 가지 그래프와 기본 성능 수치를 기본값으로 표시합니다.

작동 단축키 및 다음 옵션을 사용하면, 필요에 따라 그래프와 정보를 추가로 표시할 수 있습니다.

- **F1** 주기적인 유용한 정보 표시
- **B** 배치 크기 히스토그램 디스플레이 전환
- **F** 배경을 흐릿하게 처리하여 그래프 가독성을 높임
- **H** 그래프 숨김
- **W** 곡면
- **D** 깊이 복잡성

> DirectX Debug 런타임을 활성화하면 타이밍 그래프 및 직접 테스트가 비활성화됩니다. 디버그 런타임 환경의 추가 오버헤드 및 성능 특성은 성능 분석을 위한 부적절한 구성을 만들어 냅니다. 경고 메시지가 표시되지만 그리기 요청 그래프, 배치 히스토그램, 메모리 그래프가 여전히 작동합니다.

3.1.1. 그래프 읽기

이 모드에 표시되는 데이터는 다음과 같은 두 가지 영역으로 구분됩니다.

FPS 및 트라이앵글/프레임

기본 성능 수치는 화면 왼쪽 모서리에 표시됩니다(그림 2. NVPerfHUD 성능 분석 모드

). 이 두 숫자들은 애플리케이션이 작업부하를 얼마나 빨리 해결하는지를 보여 줍니다.

그림 3. NVPerfHUD 정보 표시줄(상단)

스크롤링 그래프

심장 박동수 모니터와 같은 이 그래프는 오른쪽에서 왼쪽으로 모든 프레임을 스크롤하므로 시간에 따른 변경사항을 확인할 수 있습니다.

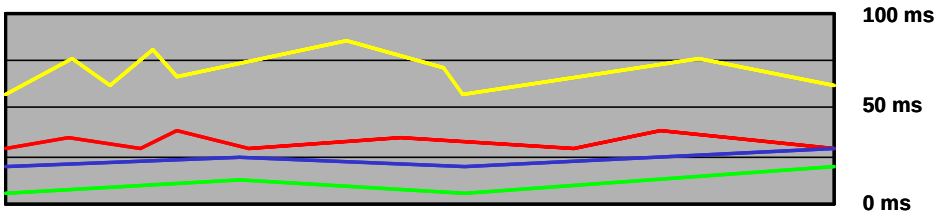


그림 4. 성능 그래프

- 녹색 = GPU_IDLE
  각 프레임에서 GPU 가 유휴 상태가 되는 시간 합계
- 파란색 = DRIVER_WAITS_FOR_GPU
  드라이버가 GPU 를 기다려야 했던 누적 시간
  (이 문제의 원인은 부록 A 를 참조하십시오)
- 빨간색= TIME_IN_DRIVER
  프레임별로 CPU 가 드라이버 코드를 실행하는 데 사용한 시간 합계.
  DRIVER_WAITS_FOR_GPU (파란색) 포함
- 노란색 = FRAME_TIME
  한 프레임의 끝부터 다음 프레임까지 소요된 시간 – 이 FRAME_TIME (노란색) 라인을 되도록 낮게 유지하려 할 것입니다.

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

FRAME_TIME (노란색) 라인과 TIME_IN_DRIVER (빨간색) 라인의 시간 차이는 애플리케이션 로직과 운영체제에 의해 소요된 시간을 의미합니다

백그라운드에서 실행 중인 운영 체제 프로세스에 의해 스파이크 형태가 나타나는 것을 볼 수 있을 것입니다 아래 그래프는 이러한 경우를 나타냅니다. 이는 하드 디스크 액세스, 텍스처 업로드, 운영 체제 컨텍스트 스위치 등에 의해 발생합니다.

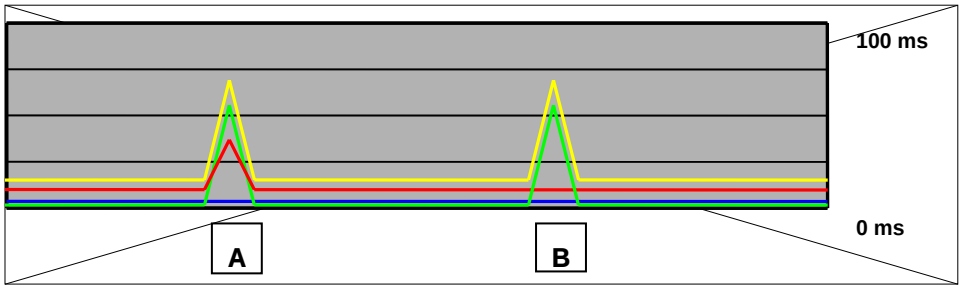


그림 5. 간헐적인 스파이크

이러한 스파이크는 정상적인 경우에도 가끔 발생하는데, 그 이유를 파악하고 있어야 합니다. 이러한 상황이 일정하게 지속된다면 애플리케이션에서 CPU 집약적인 처리를 비효율적으로 진행하고 있는 것일 수 있습니다.

- **A** 형:

TIME_IN_DRIVER (**빨간색**) 및 FRAME_TIME (노란색) 라인이 동시에 스파이크를 만든다면, 드라이버가 CPU 에서 GPU 로 텍스처를 업로딩하기 때문일 수 있습니다.

- **B** 형:

FRAME_TIME (노란색) 라인은 스파이크를 만들지만 TIME_IN_DRIVER (**빨간색**) 라인은 그렇지 않은 경우에는 애플리케이션에서 CPU 집약적인 작업(예, 오디오 디코딩)을 하고 있거나 하드 디스크에 액세스하고 있을 가능성이 높습니다. 이 상황은 운영 체제에서 다른 프로세스를 처리하고 있는 경우에도 발생할 수 있습니다.

녹색선은 GPU 로 데이터를 보내지 않을 때 스파이크를 만들 수 있다는 점을 기억하십시오.

DP(Draw Primitives) 그래프

이 그래프는 각 프레임에서 `DrawPrimitive`, `DrawPrimitiveUP` 및 `DrawIndexedPrimitives` (DP)가 호출된 회수를 나타냅니다. 이 정보를 사용하여 성능 병목현상을 찾아내는 방법은 다음 페이지 및 Chapter 6 의 파이프라인 실험에서 설명합니다.

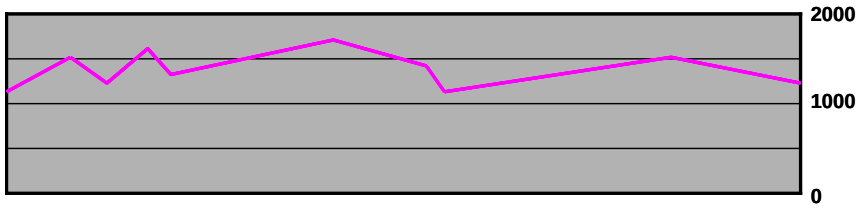


그림 6. DP 호출 그래프

배치 크기 히스토그램

배치 크기 그래프는 (작동 단축키를 사용하여) NVPerfHUD 를 활성화한 상태에서 B 키를 눌렀을 때만 나타납니다. 단위는 배치 수입니다. 왼쪽으로부터 첫 열은 0 에서 100 개 사이의 삼각형, 그 다음은 100 개에서 200 개 사이의 삼각형, 다음부터는 이와 같이 100 개씩 증가하는 삼각형 수의 범위를 나타냅니다. 따라서 작업 배치를 많이 사용하면 왼쪽이 높은 형태의 막대가 그려집니다.

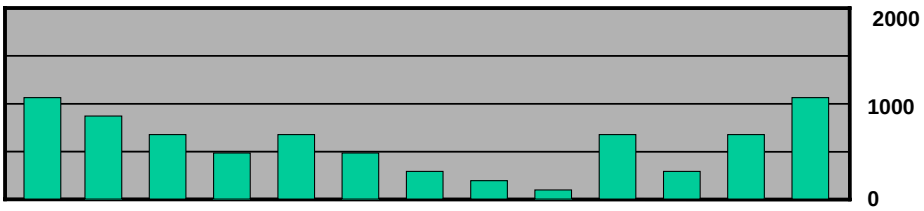


그림 7. DP 호출 히스토그램

메모리 그래프

이 그래프는 드라이버에서 할당한 AGP 및 비디오 메모리의 메가바이트를 표시합니다.

그림 8. 메모리 그래프

3.1.2. 리소스 작성 모니터

리소스 작성 모니터 표시등은 리소스를 만들 때마다 깜박입니다. Direct3D 에서 리소스를 역동적으로 작성하면 일반적으로 성능에 나쁜 영향을 미치므로 되도록 자제해야 합니다.

그림 9. 리소스 작성 모니터

모니터링되는 리소스 작성 이벤트 종류는 다음과 같습니다.

Tex	CreateTexture()로 작성된 2D 텍스처
VolTex	CreateVolumeTexture()로 작성된 볼륨 텍스처
CubTex	CreateCubeTexture()로 작성된 큐브맵 텍스처
VB	CreateVertexBuffer()로 작성된 버텍스 버퍼
IB	CreateIndexBuffer()로 작성된 인덱스 버퍼
RT	CreateRenderTarget()으로 작성된 렌더 대상
DSS	CreateDepthStencilSurface()로 작성된 깊이 스텐실 표면

Note: 리소스 작성 이벤트도 디버그 콘솔(F6)에 로그인되므로 표시등이 깜박이는 원인을 확인할 수 있습니다.

3.2. 파이프라인 실험

병목 지점을 찾으려면 한 번에 한 스테이지씩 그래픽 파이프라인의 특정 스테이지에 초점을 맞춰서 조사를 진행할 필요가 있습니다. NVPerfHUD 를 사용하면 다음과 같은 실험을 할 수 있습니다.

- **T—텍스처 유닛 격리**

  GPU 가 2×2 텍스처를 사용하도록 만듭니다. 프레임 속도가 극적으로 높아지면 텍스처 대역폭에 의해 애플리케이션 성능이 제약을 받고 있는 것입니다.

- **V—버텍스 유닛 격리**

  파이프라인 스테이지에서 버텍스 유닛 이후, 1×1 scissor rectangle 을 사용하여 모든 래스터화 및 쉐이딩 작업을 클립 처리합니다. 이 접근방식을 활용하면 버텍스 유닛 이후의 그래픽 파이프라인을 잘라내는 것과 유사한 효과를 낼 수 있으므로, 애플리케이션이 버텍스 쉐이더 또는 버스 트랜잭션에 의해 성능상 제약을 받는지를 판단할 수 있습니다.

- **N—GPU 제거**

  이 기능은 모든 DrawPrimitive() 및 DrawIndexedPrimitives() 요청을 무시함으로써

무한도로 빠른 GPU 를 측정할 수 있습니다. 이 프레임 속도는 전체 그래픽 파이프라인에 성능 비용이 없을 때 얻을 수 있는 수치를 대략적으로 나타냅니다. 상태 변경에 따라 나타나는 CPU 오버헤드도 생략된다는 점을 기억하십시오.

또한, 버전에 따라 픽셀 쉐이더를 선택적으로 비활성화하고, 애플리케이션에서 사용하는 방법을 시각화할 수도 있습니다. 특정 쉐이더 버전을 비활성화한 경우에는 이 그룹의 모든 쉐이더가 같은 색으로 표시됩니다. NVPerfHUD 에서 제공하는 쉐이더 시각화 옵션은 아래 목록과 같습니다.

1 — 고정 함수 (**빨간색**)
2 — 1.1 픽셀 쉐이더 (**녹색**)
3 — 1.3 픽셀 쉐이더 (**연한 녹색**)
4 — 1.4 픽셀 쉐이더 (**노란색**)
5 — 2.0 픽셀 쉐이더 (**파란색**)
6 — 2.a 픽셀 쉐이더 (**하늘색**)
7 — 3.0 픽셀 쉐이더 (**오렌지색**)

쉐이더 시각화는 Direct3D 장치를 NON PURE 장치로 만들 때만 작동합니다. NVPerfHUD 구성 설정에서 NON-PURE 장치가 강제로 생성되도록 설정할 수 있습니다.

Chapter 4.
디버그 콘솔 모드

여기에서는 디버그 콘솔 모드에서 사용할 수 있는 정보에 대해 설명합니다. 디버그 콘솔 화면은 그림 10. 디버그 콘솔 모드

에 나와있습니다.

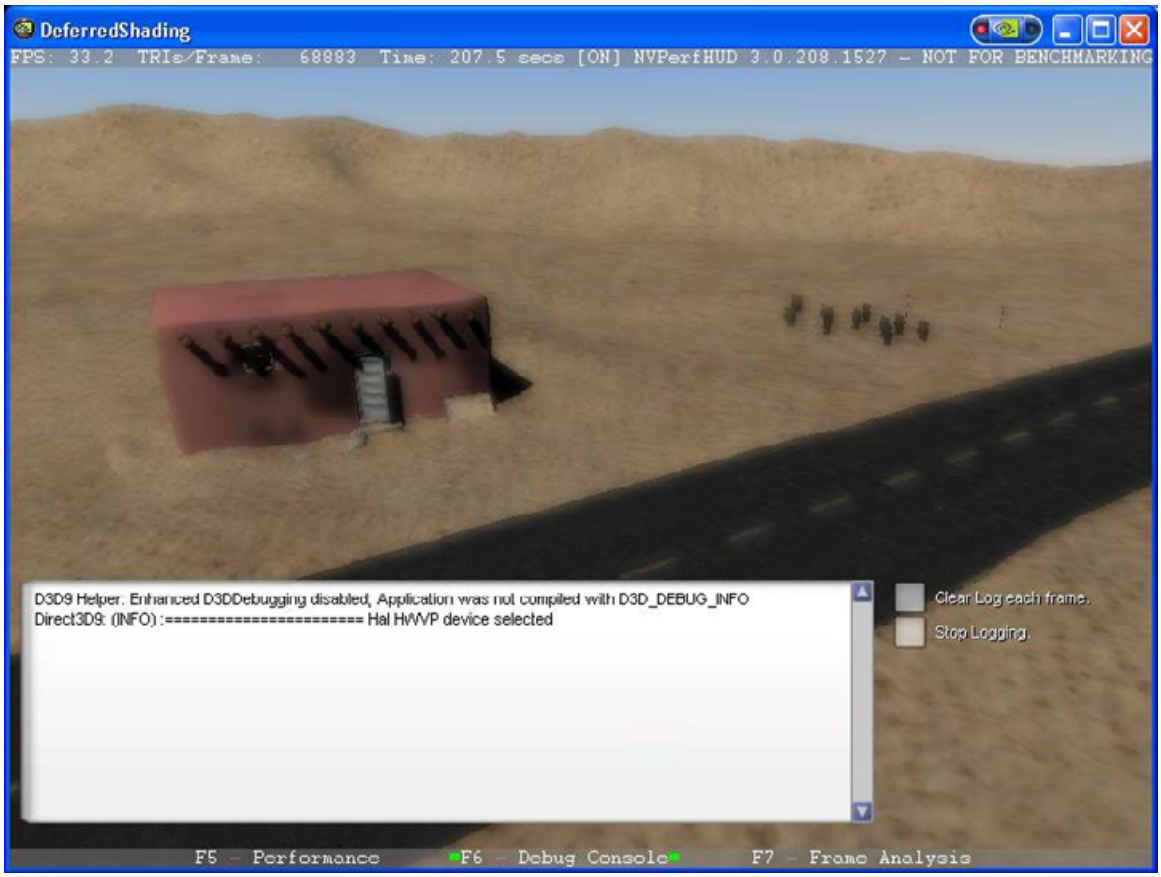


그림 10. 디버그 콘솔 모드

디버그 콘솔은 DirectX Debug 런타임을 통해 보고된 모든 메시지, OutputDebugString() 함수를 통해 애플리케이션에서 보고된 메시지, NVPerfHUD 에서 감지한 추가적인 경고 또는 오류를 표시합니다.

NVPerfHUD 에서 감지한 리소스 작성 이벤트와 경고는 디버그 콘솔에서도 로그인됩니다.

아래 옵션을 통해 디버그 콘솔 작업을 사용자 정의할 수 있습니다.

- **C** 각 프레임 로그 지우기
- **S** 로깅 중단
- **F** 페이드 콘솔

"각 프레임 로그 지우기" 확인란을 선택하면 프레임 시작 부분에서 콘솔 창의 내용이 지워지므로 현재 프레임에서 생성된 경고만을 볼 수 있게 됩니다. 이는 애플리케이션에서 콘솔 창에 적합한 것보다 더 많은 경고를 프레임별로 생성할 때 유용합니다.

"로깅 중단" 확인란을 선택하면, 콘솔에서 새 메시지 표시가 중단됩니다.

Chapter 5.
프레임 분석 모드

여기에서는 프레임 분석 모드와 고급 그래픽 파이프라인 상태 검사기를 최대한으로 활용하는 방법에 대해 설명합니다. 프레임 분석 모드 화면은 그림 11. 프레임 분석 모드과 같습니다.

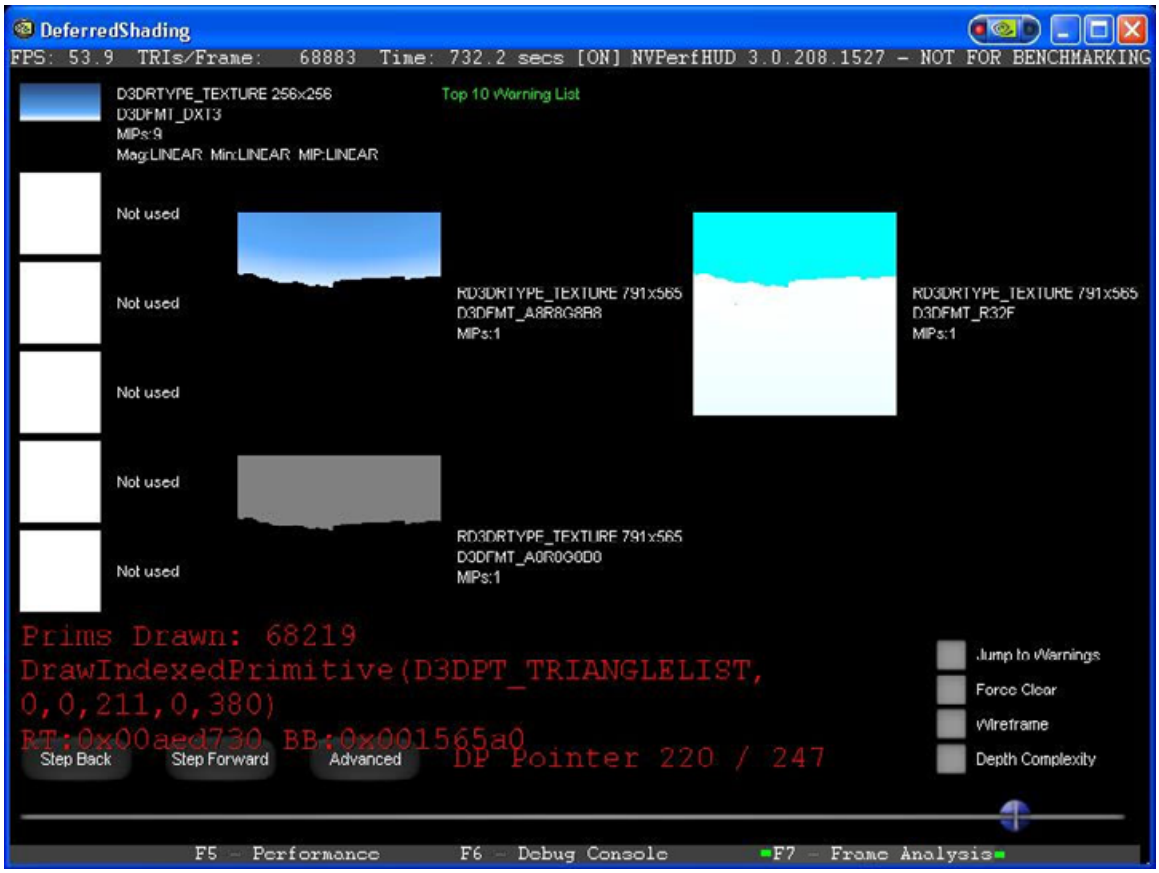


그림 11. 프레임 분석 모드

왼쪽/오른쪽 화살표 키로 이전/다음 그리기 요청을 표시하고, 프레임 분석 모드를 구성하는 데 필요한 옵션을 표시할 수 있습니다.

- **A** 고급 / 단순 디스플레이 전환

- **S** 경고 표시
- **W** 곡면
- **D** 깊이 복잡성

5.1. 렌더링 분해

렌더링 개체가 있는 프레임을 인식한 경우에는 장면을 그린 명령을 확인하고, 경고의 원인을 자세히 알아보며, 프레임 분석 모드를 사용하여 프레임 내에서 그리기 요청을 탐색할 수 있습니다. 프레임 분석 모드로 전환하면 NVPerfHUD 가 애플리케이션의 시계를 중단시키므로, 정지 상태에서 현재 프레임을 상세하게 분석할 수 있습니다. 프레임 기반 애니메이션을 사용하는 애플리케이션에서는 애니메이션 객체에 정지 시간이 영향을 미치지 않습니다.

> 프레임 분석 모드를 효과적으로 사용하려면, NVPerfHUD 가 이를 통제할 수 있도록 해야 합니다. 몇 가지 요구 조건이 아래 기재되어 있습니다. 더 자세한 내용은 이 설명서 끝 부분에 있는 문제해결을 참조하십시오.

프레임 분석 모드에서는 애플리케이션이 `QueryPerformanceCounter()` 또는 `timeGetTime()` win32 함수를 사용하고 이에 의존할 것을 요구합니다. 애플리케이션은 경과 시간(dt) 계산을 강력하게 처리해야 하며 `dt` 가 영(0)일 경우에는 더욱 그러합니다. 다시 말해, 프로그램을 `dt` 로 구분하지 말아야 한다는 뜻입니다.

애플리케이션을 멈춘 상태에서, **다음(오른쪽 화살표)** 및 **이전(왼쪽 화살표)** 키보드 버튼을 눌러 해당 프레임의 모든 그리기 요청을 단계별로 확인합니다. 화면 하단에 있는 슬라이더를 앞뒤로 끌거나 **PgUp / PgDn** 을 눌러 특정 그리기 요청을 빠르게 탐색할 수도 있습니다. 마우스 또는 키보드 이벤트 가로채기가 제대로 작동하지 않는 경우에는 NVPerfHUD 구성 대화 상자에서 애플리케이션을 종료하고 대체 API 가로채기 옵션을 선택하십시오.

각 그리기 요청에 표시되는 정보는 다음과 같습니다.

- 직전에 그린 그리기 요청 및 전체 개수
- 마지막 그리기 요청의 함수 이름 및 매개변수 그리기 요청에 경고가 생기면 이 경고 메시지가 함께 표시됩니다.

5.1.1. 경고 표시

경고 표시를 활성화하면, 화면 상단 목록 상자에 DP 요청의 가장 중요한 버텍스 버퍼(VB) 잠금을 포함한 경고 목록이 표시됩니다. 경고를 클릭하면 연관된 DP 요청으로 직접 이동합니다. **위 / 아래 화살표**를 사용하여 목록을 스크롤할 수도 있습니다. 경고 메시지가 표시된 상태에서

다음(오른쪽 화살표) 또는 **이전(왼쪽 화살표)**을 클릭하면, 화면 하단의 슬라이더가 이전 / 다음 그리기 요청으로 이동합니다.

잠금에 사용하는 시간을 이해하고 최소한으로 만들려면 각각의 VB 잠금을 검사해야 합니다. 이는 애플리케이션에서 버텍스 버퍼를 설정하는 데 사용하는 CPU 시간을 이해하는 데에도 유용합니다.

5.1.2. 텍스처 유닛 및 RTT 정보

각 텍스처 유닛에 대해 표시되는 정보와 텍스처 (RTT) 대상에 대한 오프스크린 렌더는 다음과 같이 구성됩니다.

- 각 텍스처 유닛에 저장된 텍스처와 각각의 속성:
  - 치수
  - 필터링 매개변수: 수정, 확대, MTP 레벨
  - 텍스처 포맷: `RGBA8, DXT1, DXT3, DXT5` 등
  - 텍스처 대상: 1D, 2D, 볼륨 텍스처, NP2 등
- 현재 그리기 요청이 오프스크린 텍스처(RTT)로 렌더링되면, 해당 텍스터의 내용과 속성이 화면 가운데 표시됩니다.

화면에 적합한 것보다 많인 텍스처가 사용되는 경우에는 스크롤 막대로 목록을 확인할 수 있습니다.

5.1.3. 시각화 옵션

프레임 분석 모드는 여러 가지 시각화 옵션을 지원합니다.

- **D 깊이 복잡성:** 이 옵션은 붉은 빛으로 추가 블렌딩을 켭니다. 화면이 밝아 질수록 더 많은 버퍼에 손을 대게 됩니다. 버퍼프레임 RMW(읽기-수정-쓰기)가 증가되면 컬러 값도 8 씩 늘어납니다. 최대값은 32 RMW 입니다. 화면 컬러가 포화되면, 애플리케이션에서 프레임 버퍼를 너무 자주 덮어써 제한된 성능 병목을 채우게 될 수 있습니다.
- **W 곡면**: 곡면 렌더링을 강제하는 옵션이므로 장면의 기하 복잡성을 시험할 수 있습니다.

5.1.4. 고급 상태 검사기

고급... (A) 버튼을 클릭하면 고급 상태 검사기가 활성화됩니다. 화면 하단에 있는 슬라이더를 사용하여 탐색을 계속할 수 있지만, 화면 상단에도 그래픽 파이프라인의 각 스테이지에 대한 여러 버튼이 생깁니다. 버튼을 클릭하거나 단축키를 눌러 각 검사기를 전환할 수 있습니다.

- **1** 인덱스 유닛 – 버텍스 데이터 페치
- **2** 버텍스 쉐이더 – 버텍스 쉐이더 실행

- 3 픽셀 쉐이더 – 픽셀 쉐이더 실행
- 4 래스터 연산 – 프레임 버퍼에서의 포스트 쉐이딩 연산

Note: 각 NVPerfHU 상태 검사기에 표시되는 상태 정보에 대한 자세한 내용은 DirectX SDK 최신 버전과 함께 설치된 문서를 참조하십시오.

각 스테이지를 클릭하면, 현재 그리기 요청에서 이 스테이지에 발생한 나타난 생황을 자세하게 알아볼 수 있습니다. 아래에서는 각 상태 검사기에 의해 표시되는 정보를 설명합니다.

5.2. 인덱스 유닛 상태 검사기

이 상태 검사기를 선택하면, 현재 그리기 요청시 인덱스 유닛에 대한 정보가 표시됩니다.

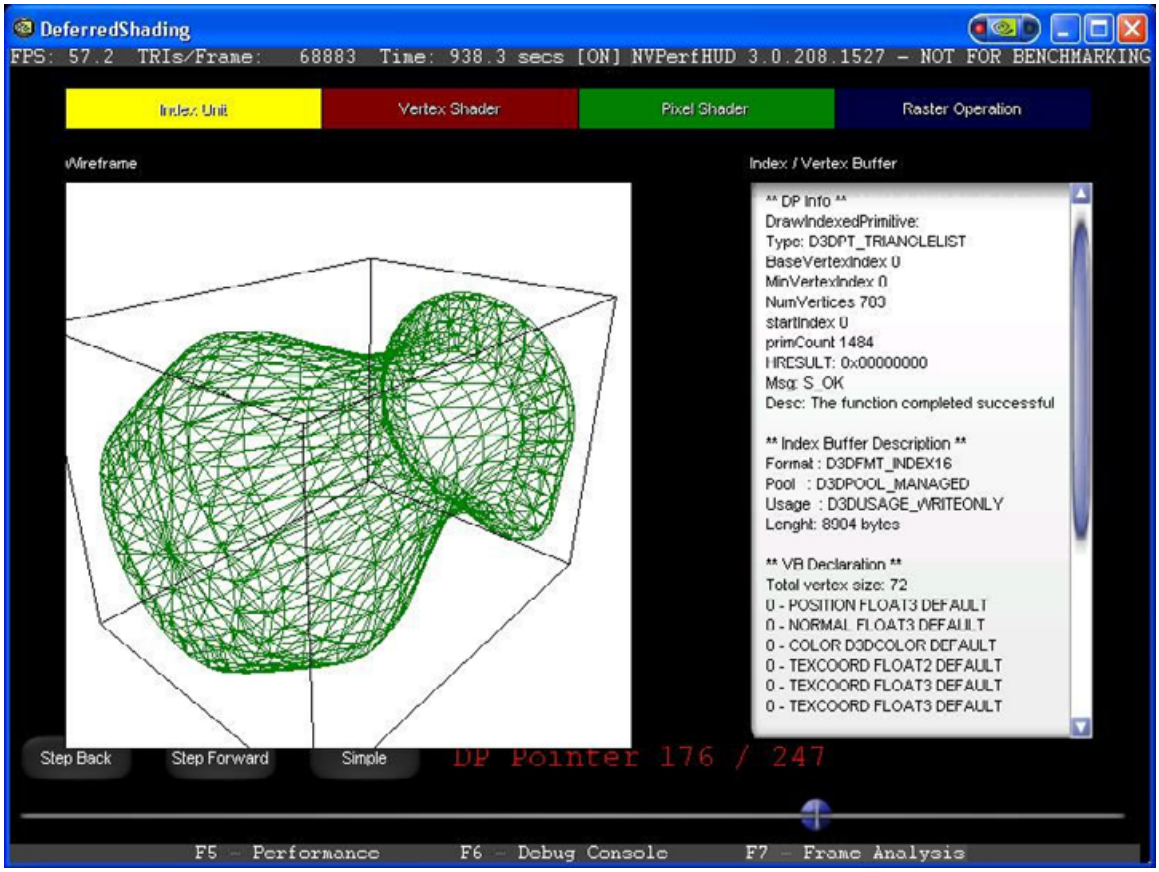


그림 12. 인덱스 유닛 상태 검사기

이 그리기 요청과 연관된 기하의 회전하는 곡면 렌더링이 화면 가운데 경계선 상자 안쪽에 표시됩니다.

Note: 이 버전에서는 점과 라인을 지원하지 않습니다.

다음으로, 이 그리기 요청에 대한 버텍스 데이터를 가져오는 데 사용한 모든 정보(아래 참조)를 보여주는 목록 상자가 나타납니다.

- 그리기 요청 매개변수 및 반환 플래그
- 인덱스 및 버텍스 버퍼 형식, 크기 등
- FVF

인덱스 유닛 상태 검사기를 사용하는 경우에는 애플리케이션에서 보낸 배치가 정확한지 곡면 렌더링을 살펴 보아야 합니다. 예를 들어, 매트릭스 팔레트 스키닝 및 렌더링이 손상된 경우에는 버텍스 버퍼/인덱스 버퍼의 레퍼런스 포스처가 정확한지 확인해야 합니다. 이 레퍼런스 포스처가 정확한 경우, 이 기하의 렌더링 손상은 버텍스 쉐이더나 잘못된 버텍스 무게가 원인일 수 있습니다.

인덱스 형식이 올바른지도 확인해야 합니다. 되도록 16 비트 인덱스를 사용하도록 하십시오.

5.3. 버텍스 쉐이더 상태 검사기

이 상태 검사기를 선택하면, 현재 그리기 요청시 버텍스 쉐이더에 대한 정보가 표시됩니다.

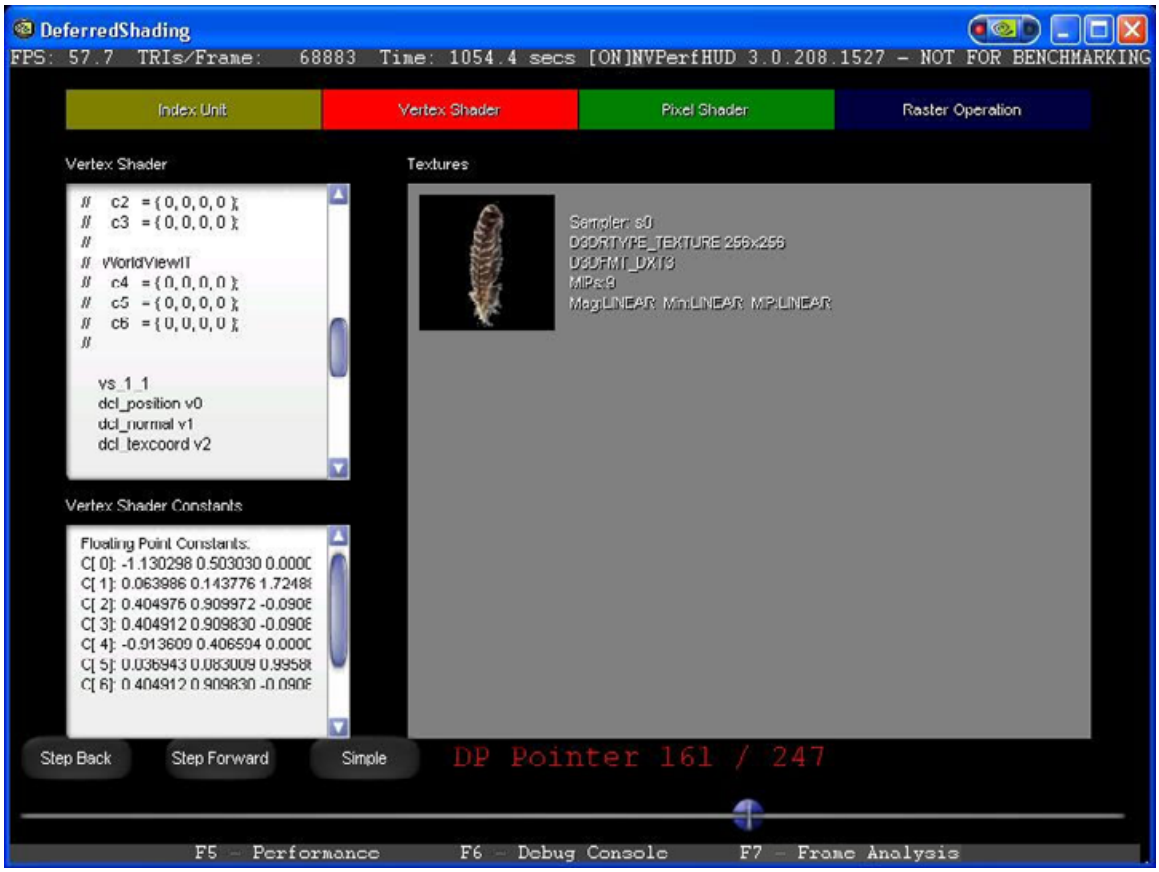


그림 13. 버텍스 유닛 상태 검사기

버텍스 쉐이더 프로그램과 관련 상수, 텍스처가 조사를 위해 표시됩니다. 버텍스 쉐이더가 주소 레지스터(예, 매트릭스 팔레트 스키닝)를 사용하는 경우에는 모든 상수가 표시됩니다. 각 텍스처 샘플러에 대한 정보도 레퍼런스로 표시됩니다. + / – 키를 사용하여 표시된 텍스처 배율을 조정하십시오.

버텍스 쉐이더 상태 검사기를 사용할 때에는

- 현재 그리기 요청에 예상한 버텍스 쉐이더가 적용되었는지 검사해야 합니다.
- 상수가 **#NAN** 또는 **#INF** 가 아닌지 확인해야 합니다.

5.4. 픽셀 쉐이더 상태 검사기

이 상태 검사기를 선택하면, 현재 그리기 요청시 픽셀 쉐이더에 대한 정보가 표시됩니다.

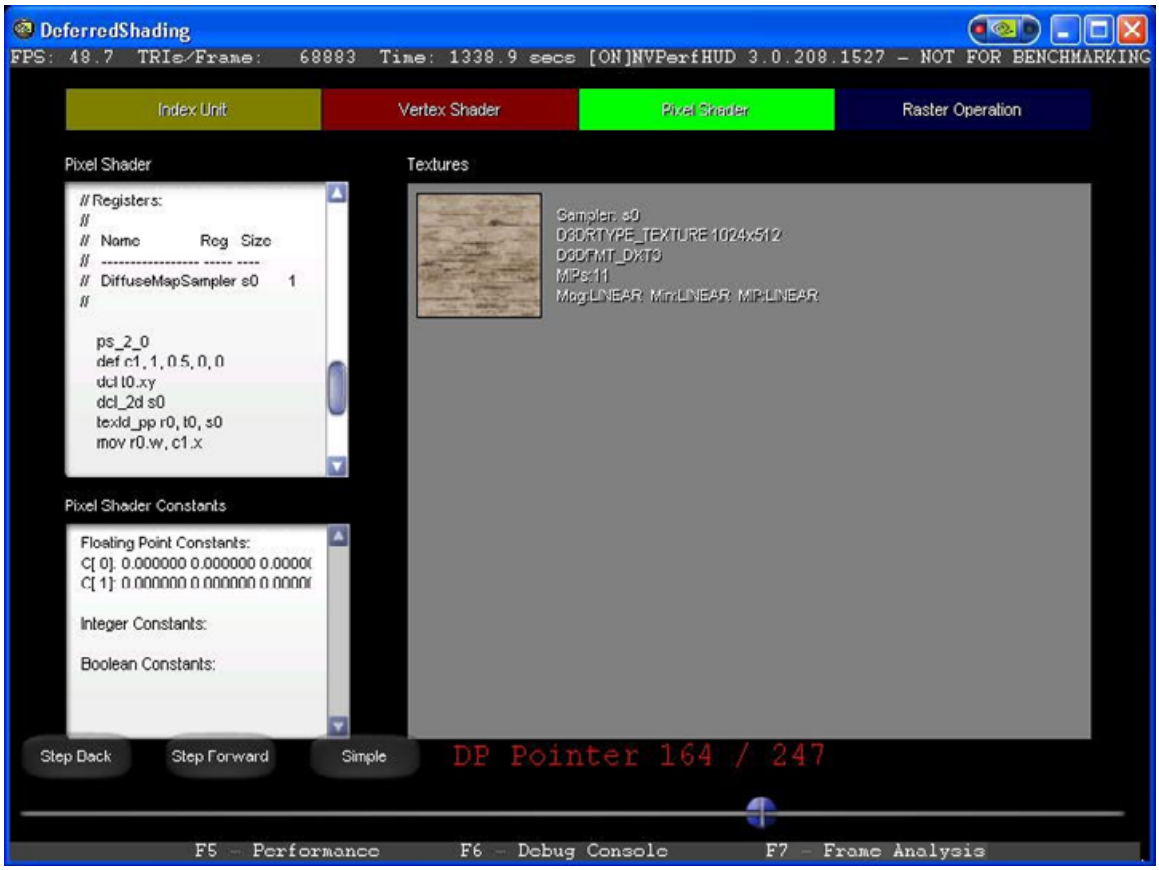


그림 14. 픽셀 쉐이더 상태 검사기

픽셀 쉐이더 프로그램과 관련 상수, 텍스처가 조사를 위해 표시됩니다. 각 텍스처 샘플러에 대한 정보도 레퍼런스로 표시됩니다. + / – 키를 사용하여 표시된 텍스처 배율을 조정하십시오.

픽셀 쉐이더 상태 검사기를 사용할 때에는

- 현재 그리기 요청에 예상한 픽셀 쉐이더가 적용되었는지 검사해야 합니다.
- 상수가 **#NAN** 또는 **#INF** 가 아닌지 확인해야 합니다.
- 텍스처 및 렌더-텍스처가 제대로 사용되었는지 확인해야 합니다.

5.5. 래스터 연산 상태 검사기

이 상태 검사기를 선택하면, 현재 그리기 요청시 래스터 연상(rop)에 대한 정보가 표시됩니다.

Note: 래스터 연산 상태 검사기에 표시되는 상태 정보에 대한 자세한 내용은 DirectX SDK 최신 버전과 함께 설치된 문서를 참조하십시오.

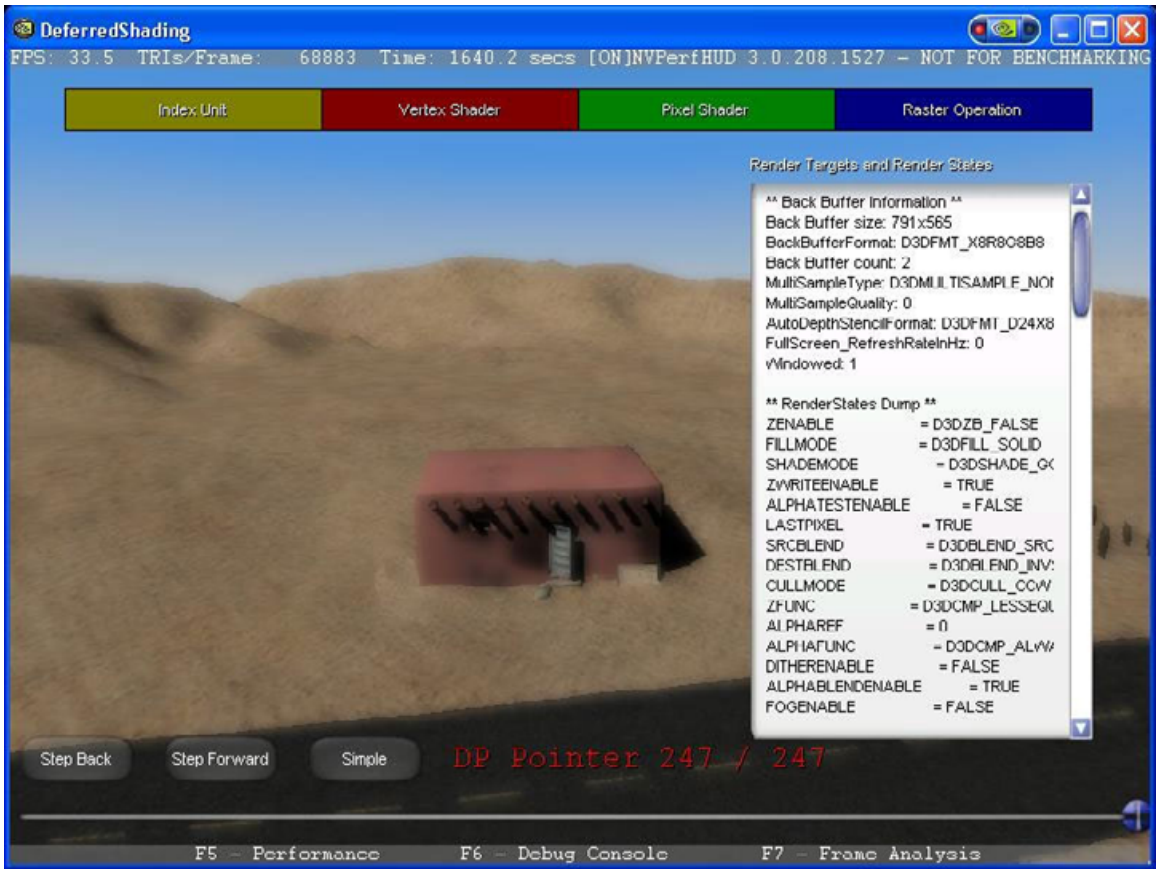


그림 15. 래스터 연산 상태 검사기

이 그리기 요청에 대한 포스트 쉐이딩 래스터 연산 정보가 검사를 위해 표시됩니다. 표시되는 정보는 다음과 같습니다.

- 렌더 대상 형식
- 백 버퍼 형식
- 값비싼 프레임 버퍼 처리의 원인이 될 수 있는 렌더 상태는 다음과 같습니다.
  - **Zenable** – **Z** 연산 비교
  - **Fillmode** – 래스터화 모드
  - **ZWriteEnable** – 깊이 버퍼에 **Z** 쓰기 여부
  - **AlphaTestEnable** – 알파 테스트 사용
  - **SRCBLEND** 및 **DSTBLEND** – 혼합 연산 정의
  - **AlphablendEnable** – 프레임 버퍼의 애플리케이션 혼합

- **Fogenable** – 안개 사용
- **Stencil enable** – 스텐실 버퍼에 쓰기 활성화
- **StencilTes**t….

래스터 연산 상태 검사기를 사용할 때에는

- 혼합이 제대로 작동하지 않으면, 백 버퍼 향식에 알파 상수가 진짜로 있는지 확인해야 합니다.
- 불투명 객체를 **blendEnable** 로 그리지 않도록 해야 합니다.

Chapter 6.
성능 병목 분석

6.1. 그래픽 파이프라인 성능

지난 몇 년 동안 하드웨어 가속 렌더링 파이프라인은 복잡도가 계속 증가했으며, 이와 더불어 성능 특성 역시 점점 복잡하고 혼란을 야기하는 방향으로 변하고 있습니다. 예전에는 단순히 렌더링 루프 내의 CPU 주기를 줄이면 성능을 높일 수 있었으나, 지금은 병목 지점을 찾은 후 이것을 체계적으로 최적화해 나가는 일련의 주기로 변화되었습니다. 이러한 *'발견과 최적화'라는* 반복적인 과정은 가장 느린 스테이지 만큼만 속도만 낼 수 있는 것이 파이프라인이라는 확실한 사실에 기초하여, 이종 멀티 프로세서 시스템을 조율하는 기본적인 개념입니다. 논리적으로 결론을 내려볼 때, 단일 프로세서 시스템에서는 섣부르고 초점 없는 최적화로도 어느 정도의 성능 향상을 가져올 수 있지만, 멀티 프로세서 시스템에서는 이 정도의 최적화로 어떠한 성능 향상도 가져올 수 없습니다.

그래픽 최적화에 엄청난 노력을 투자하고도 성능을 향상시키지 못한다는 것은 재미 없는 일이 것입니다. 여기에서는 NVPerfHUD 를 어떻게 사용하여 병목을 찾아내는지 설명하여, 사용자가 시간을 낭비하지 않도록 하는 것이 목표입니다.

6.1.1. 파이프라인 개요

최상위 레벨에서 파이프라인은 CPU 와 GPU, 두 부분으로 구분됩니다.

그림 16. 파이프라인 개요

은 GPU 안에 병렬 처리가 가능한 여러 개의 기능 단위와 병목이 발생할 수 있는 여러 개의 지점이 있다는 것을 보여줍니다. 이 때, 기능 단위는 본래 특수 목적을 위해 분리된 프로세서로 볼 수 있습니다. 여기에는 버텍스 및 인덱스 인출, 버텍스 쉐이딩(변형 및 조명), 픽셀 쉐이딩, 텍스처 로딩, 래스터 연산(ROP) 등이 있습니다.

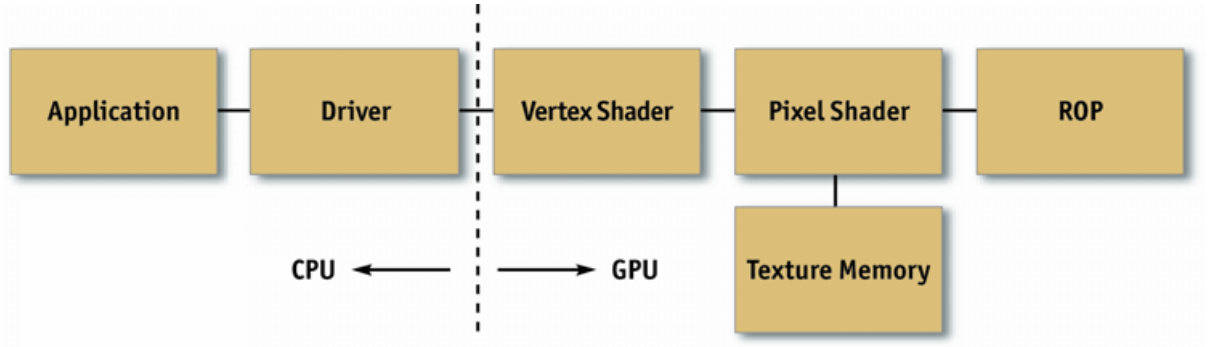


그림 16. 파이프라인 개요

6.1.2. 방법론

병목 지점을 찾는 과정이 없이 최적화를 한다는 것은 많은 개발 노력을 헛되게 만드는 결과를 초래할 수 있습니다. 그런 의미에서 최적화 과정을 다음과 같은 기본적인 '발견과 최적화' 루프로 정형화해 볼 수 있습니다.

- **인식**

  파이프라인의 각 단계에서 NVPerfHUD 를 사용하여 그 단계만 분리해내어 실험을 합니다. 성능이 변한다면 병목 지점을 찾은 것입니다. 이와 비슷하게 애플리케이션에서 그 단계의 작업량을 변화시키는 실험을 구현하여 병목점을 찾을 수도 있습니다.

- **최적화**

  위 과정을 통해 병목점을 발견된 스테이지에서, 성능이 더 이상 증가하지 않거나 원하는 수준에 도달할 때까지 작업량을 줄입니다.

- **반복**

  원하는 수준에 도달할 때까지 1 단계와 2 단계를 반복합니다.

6.2. 병목점 찾기

병목점을 찾는 것은 최적화 싸움의 절반을 차지하는 과정이라고 말할 수 있습니다. 이를 바탕으로, 최적화 노력의 초점을 어디에 맞출 것인지에 대한 현명한 판단을 내릴 수 있기 때문입니다. 그림 17. 병목점 찾기

은 애플리케이션에서 정확한 병목점을 찾아가는 일련의 과정을 나타낸 순서도입니다. 여기에서는 프레임 버퍼 연산(래스터 연산이라고도 함)과 함께 파이프라인의 맨 끝에서 출발하여 CPU 에서 끝을 맺는다는 점에 주목해야 합니다. 또한, 단일 기본도형(주로 삼각형)은 명백하게 하나의 병목점을 갖는데, 그 병목점 차체가 프레임 흐름 속에서 변할 가능성이 높다는 것입니다. 그렇기 때문에 파인프라인 중 둘 이상의 스테이지에서 작업부하를 수정하면 성능에 변화가 생기는 경우가 자주 발생합니다. 예를 들어 로우 폴리곤으로 만들어진 스카이 박스의 경우 픽셀 쉐이딩 또는 프레임 버퍼 접속에 의해 처리 속도가 제한되는 반면, 적은 수의 픽셀에 매핑되는 스킨 메시의 경우 CPU 또는 버텍스 처리에 의해 속도가 제한됩니다. 그렇기 때문에, 객체별 또는 재질별로 작업량을 변화시켜 보는 것이 도움이 되는 경우도 있습니다.

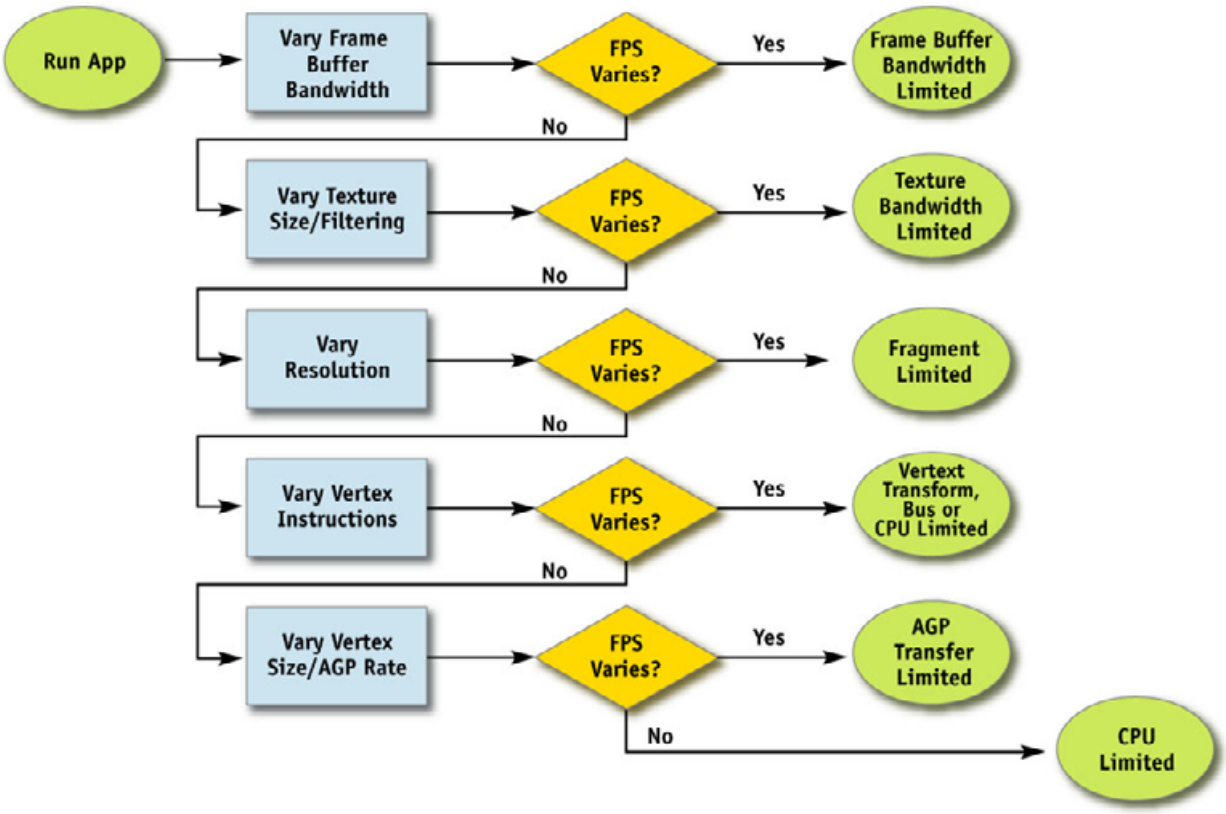


그림 17. 병목점 찾기

애플리케이션이 CPU 제한적이라고 의심되는 경우에는 언제든지 N 을 눌러보십시오. 애플리케이션의 프레임 속도가 크게 늘어나지 않으면, CPU 제한적인 애플리케이션입니다.

6.2.1. 래스터 연산 병목현상

파이프라인의 끝에 위치한 래스터 연산(ROP)은 깊이와 스텐실을 읽고 쓰는 작업을 담당합니다. 여기에는 깊이와 스텐실 값 비교 연산, 컬러값 읽고 쓰기, 알파 블렌딩 및 테스팅과 같은 작업들이 있습니다. 보다시피, ROP 관련 작업들은 프레임 버퍼 대역폭을 소비합니다.

애플리케이션이 프레임 버퍼 대역폭에 의해 제한되는지 알아보는 가장 좋은 방법은 컬러와 깊이 버퍼의 비트 값을 바꿔보는 것입니다. 32 비트에서 16 비트로 비트 값을 낮췄을 때 성능이 크게 개선되면, 프레임 버퍼 대역폭에 의해 제약을 받고 있는 것이 분명합니다.

프레임 버퍼 대역폭은 GPU 의 메모리 클럭 속도에 의해 좌우됩니다.

6.2.2. 텍스처 대역폭 병목현상

텍스처 대역폭은 메모리로부터 텍스처를 가져올 때마다 소모됩니다. 메모리에 대한 방대한 요청을 최소화하기 위해 최신 GPU 에는 텍스처 캐시가 있지만, 텍스처를 메모리로부터 가져오는 작업은 여전히 필요하며 그 때마다 상당히 큰 메모리 대역폭이 소모됩니다.

NVPerfHUD 를 실행 중일 때 **T**를 누르면 애플리케이션의 모든 텍스처가 2x2 텍스처로 바뀝니다. 이렇게 하면 텍스처를 가져오는 속도가 훨씬 빨라지며, 텍스처 캐시 관련 응집성을 그대로 가져올 수 있습니다. 이렇게 해서 성능이 크게 증가하면, 텍스처 대역폭에 의해 제약을 받고 있는 것입니다.

텍스처 대역폭도 GPU 의 메모리 클럭 속도에 따라 좌우됩니다.

6.2.3. 픽셀 쉐이딩 병목현상

픽셀 쉐이딩은 컬러값과 깊이값을 포함한 픽셀을 생성하는 데 사용되는 실제 비용을 가리킵니다. 즉, “픽셀 쉐이더”를 실행하는 데 드는 비용을 말합니다. 픽셀 쉐이딩과 프레임 버퍼 대역폭은 “채움 속도”라는 단어로 모호하게 설명되기도 합니다. 둘 다 화면의 해상도와 관련이 있기 때문입니다. 하지만 이 둘은 분명 차이가 있으며, 그 차이를 정확하게 구분해내는 것은 효과적으로 병목현상을 찾고 최적화하는 데 매우 중요합니다.

픽셀 쉐이딩이 병목현상을 알아내는 첫 번째 단계는 NVPerfHUD 를 사용하여 모든 쉐이더를 아주 단순한 쉐이더로 바꿔버리는 것입니다. 이렇게 하려면, ‘성능 분석 상태’에서 픽셀 쉐이더 프로필을 한번에 하나씩 작동 해제하고 프레임 속도의 변화를 지켜보십시오. 이로 인해 성능이 향상된다면, 문제의 원인은 픽셀 쉐이딩일 가능성이 높습니다.

다음 단계는 NVShaderPerf 또는 FX Composer 의 Shader Perf 패널을 통해 가장 비용이 많이 드는 쉐이더가 무엇인지 알아보는 것입니다. 픽셀 쉐이더 비용은 픽셀 하나를 기준을 한 것이므로, 값이 비싸다 하더라도 적은 수의 픽셀에 영향을 미치는 쉐이더가 많은 수의 픽셀에 영향을 미치는 쉐이더에 비해 성능 관련 문제를 일으킬 가능성이 적다는 점을 기억하십시오. 기본적으로 ‘쉐이더 비용 = 픽셀당 비용 * 영향을 받는 픽셀의 수’입니다. 가장 비용이 많이 드는 쉐이더에 성능 최적화 노력을 집중하십시오.

FX Composer 에는 쉐이더의 성능 비용을 줄이는 데 유용한 여러 가지 쉐이더 최적화 관련 설명서가 들어 있습니다.

최신 버전의 FX Composer와 NVShaderPerf는 NVDIA GPU의 전 제품군에 걸쳐 쉐이더 성능을 분석할 수 있도록 지원합니다. 둘 다 http://developer.nvidia.com에서 구할 수 있습니다.

6.2.4. 버텍스 연산 병목현상

렌더링 파이프라인 중에서 버텍스 변형 단계는 버텍스 속성들(모델 공간 좌표, 버텍스 노멀, 텍스처 좌표 등)을 받아서 클리핑 연산과 래스터화에 적합한 속성들(이종 클립 공간 좌표, 버텍스 라이팅 계산 결과, 텍스처 좌표 등)을 만드는 것을 담당합니다. 물론 이 단계의 성능은 버텍스당 필요한 연산의 수와 처리해야 할 버텍스의 수에 달려 있습니다.

버텍스 처리가 병목점인지 판단하는 법은 애플리케이션을 NVPerfHUD 와 함께 실행한 다음 **v** 키를 눌러스 버텍스 유닛를 분리해 버리면 간단합니다. 그렇게 했을 때 프레임 속도가 원래의 프레임 속도와 거의 같다면, 애플리케이션이 버텍스/인덱스 버퍼 AGP 에 의한 전송, 버텍스 쉐이더 유닛 또는 비효율적인 lock 과 그로 인한 GPU 스톨(stall) 같은 요인들 중 하나에 의해 제약을 받고 있는 것입니다.

비효율적인 lock 을 배제하려면, 애플리케이션을 Direct3D 디버그 런타임으로 구동한 후 오류 메시지나 경고 메시지가 없다는 것을 확인합니다.

6.2.5. 버텍스 및 인덱스 전송 병목현상

파이프라인 중 GPU 부분의 첫 단계는 GPU에 의해 버텍스와 인덱스를 가져오는 것입니다. 버텍스 및 인덱스 페치 성능은 실제 버텍스와 인덱스가 어느 부분에 위치해 있는지에 따라 달라질 수 있습니다. 일반적으로 버텍스와 인덱스는 시스템 메모리에 존재하거나 로컬 프레임 버퍼 메모리에 존재합니다. 시스템 메모리에 존재한다는 것은 곧, 버텍스와 인덱스가 AGP 또는 PCI-Express와 같은 버스를 통해 GPU로 전달될 것임을 의미합니다. 많은 경우, 특히 PC 플랫폼에서 이러한 결정은 애플리케이션이 아니라 장치 드라이버에 맡기는데, 그 대신 애플리케이션은 최신 그래픽 AGI를 통해 드라이버가 적당한 메모리 유형을 고르도록 사용 힌트를 제공할 수 있습니다. 인덱스 버퍼와 버텍스 버퍼를 애플리케이션에서 최적의 방식으로 사용하는 것에 관해서는 NVIDIA GPU 프로그래밍 가이드[링크]를 참조하십시오.

버텍스와 인덱스 페치 중에 어느 것이 병목점인지는 버텍스 포맷 크기를 바꿔보면 알 수 있습니다.

버텍스와 인덱스 페치 성능은 그 데이터들이 시스템 메모리에 있는 경우에는 AGP/PCI-Express 의 전송 속도에 달려 있으며, 로컬 프레임 버퍼 메모리에 있는 경우에는 메모리 클럭에 달려 있습니다.

6.2.6. CPU 병목현상

애플리케이션이 CPU 에 의해 제약을 받는지 알아보는 데에는 두 가지 방법이 있습니다.

애플리케이션 성능이 CPU 에 의해 제약을 받는지 알아보는 한 가지 간단한 방법은 NVPerfHUD 의 타이밍 그래프에 있는 녹색(GPU Idle) 라인을 살펴보는 것입니다. 녹색 라인이 그래프 바닥에서 수평 상태를 유지한다면 GPU 가 결코 유휴 상태가 아닌 것입니다. 하지만 그 녹색 라인이 그래프 바닥에서 상승한다면, 이는 CPU 가 GPU 에 충분한 작업을 전송하고 있지 못하고 있음을 의미합니다.

N 키를 눌러 GPU 를 분리해내는 방법으로도 애플리케이션의 제약을 알 수 있습니다. 이것을 할 때에는, NVPerfHUD 가 Direct3D 런타임에 강제하여 모든 DP 호출을 무시하게 합니다. 그렇게 해서 나온 프레임 속도는 무한정으로 빠른 GPU 와 디스플레이 드라이버를 사용해서 얻을 수 있을 프레임 속도와 거의 같습니다.

애플리케이션 성능이 CPU 에 의해 제약을 받고 있다면(GPU 를 공급할 수 없을 정도로 바쁘다면) 다음과 같은 원인에 의한 것일 수 있습니다.

너무 많은 DP 호출 – 호출마다 드라이버 오버헤드가 있습니다. 관련 설명은 아래 그림 18. 너무 많은 드라이버 호출

- 을 확인하십시오.
- 애플리케이션의 로직, 물리 로직 등에 대한 요구 - **노란색**(FRAME_TIME) 라인과 **빨간색**(TIME_IN_DRIVER) 라인의 격차는 애플리케이션에 할당된 CPU 의 양을 나타냅니다.

리소스의 로딩 또는 할당 – 예를 들어, 드라이버는 텍스처별로 처리를 하므로, 드라이버가 많은(다량의) 텍스처를 로딩하느라 바쁜 동안에는 GPU 가 모든 보류 작업을 마무리할 수도 있습니다. 0. 그림 8. 메모리 그래프

- 를 참조하십시오.
-

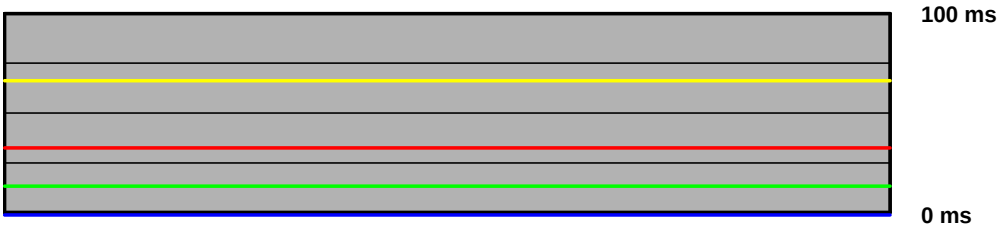


그림 18. 너무 많은 드라이버 호출

그림 18. 너무 많은 드라이버 호출

은 애플리케이션이 드라이버에 너무 많은 호출을 하고 있을 때의 일반적인 경우를 나타냅니다. 배치 수를 보고하는 그래프를 함께 보면서 이 시나리오를 다시 한번 점검합니다.

6.3. 최적화

이제 우리는 병목점이 어디인지 찾았으므로, 애플리케이션의 성능을 높이기 위해 그 특정 단계를 최적화해야 합니다. 다음에 나온 최적화 방법은 애플리케이션의 전체 성능을 향상시킬 수 있도록 단계별로 정리한 것입니다.

6.3.1. CPU 최적화

애플리케이션은 복잡한 물리 로직 또는 AI 로 인해 CPU 에 의한 제약을 받을 수도 있습니다. 부족한 배치 크기와 리소스 관리 때문에 성능에 저하가 올 수도 있습니다. 애플리케이션이 CPU 에 의해 제약을 받고 있다는 것을 알게 되었다면, 렌더링 파이프라인에서 CPU 작업량을 줄일 수 있도록 다음과 같은 방법을 시도해 보십시오.

6.3.2. 리소스 잠금 회수 줄이기

리소스는 텍스처 또는 버텍스 버퍼일 수 있습니다. GPU 리소스에 대한 접근을 요구하는 동기화 연산을 수행할 때마다, GPU 파이프라인을 통째로 작동 중단시킬 가능성이 있는데, 그렇게 되면 CPU 와 GPU 주기에 모두 비용이 발생합니다. CPU 는 GPU 파이프라인이 요청한 리소스를 빼서 넘겨줄 때까지 기다리면서 루프를 돌아야 하기 때문에 CPU 주기를 낭비하게 됩니다. 그런 다음 파이프라인이 유휴 상태로 있으면서 다시 채워넣어야 하기 때문에, GPU 주기도 낭비하게 됩니다.

이러한 현상은 다음과 같은 작업을 할 때 발생할 수 있습니다.

- 이전에 렌더링하고 있었던 표면을 잠그거나 그 표면에서 데이터를 읽어들일 때
- 텍스처 또는 버텍스 버퍼와 같이, GPU 가 데이터를 읽어들이는 표면에 쓰기를 할 때

바쁜 리소스를 잠그면 파란색(DRIVER_WAITS_FOR_GPU) 라인을 높이는 데 기여합니다. 드라이브가 GPU 를 기다리게 만드는 것들에 대한 자세한 내용은 부록 A 를 참조하십시오.

비효율적인 잠금 작업을 배제하려면, 애플리케이션을 Direct3D 디버그 런타임으로 구동한 후 오류 메시지나 경고 메시지가 없다는 것을 확인하십시오. 리소스에 대한 잠금 작업을 효율적으로 관리하는 법에 관해서는 다음 백서를 참고하십시오.
http://developer.nvidia.com/object/dynamic_vb_ib.html

6.3.3. 그리기 호출 최소화

도형을 그리는 모든 API 함수 호출은 그와 관련한 CPU 비용을 갖고 있습니다. 따라서 API 호출의 수를 최소화하고 특히 그래픽 상태 변화의 수를 최소화하면 렌더링해야 하는 특정한 수의 삼각형에 사용할 CPU 의 양을 최소화할 수 있습니다.

*배치(batch)*는 'DirectX9 에서 DrawPrimitive()와 DrawIndexedPrimitive() 같은 하나의 API 렌더링 호출로 렌더링한 도형(primitive) 그룹'으로 정의합니다. 그리고 배치의 "크기"는 그 배치에 포함된 도형의 수를 가리킵니다.

NVPerfHUD 를 사용하면 배치 작업을 제대로 수행하고 있는지 알 수 있습니다. **B** 키를 누르면 한 프레임에서 한 번의 그리기 호출당 삼각형의 수의 배치를 나타내는 막대 그래프를 볼 수 있습니다. 그림 19. 많은 사소한 DP 호출

는 적은 수의 도형으로 너무 많은 DP 호출을 하기 때문에 수행 능력이 떨어질 것으로 예상되는 애플리케이션을 나타냅니다.

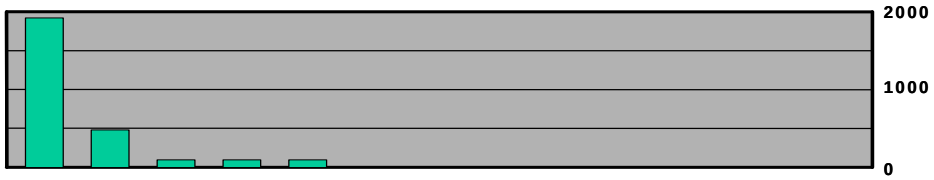


그림 19. 많은 사소한 DP 호출

DP 호출 수를 줄이려면 다음과 같은 방법을 시도해 보십시오.

- **'triangle strip'을 사용한다면, 분해된 스트립(strip)을 붙이기 위해 '퇴화된' 삼각형을 사용하십시오.** 이 방법을 쓰면 여러 개의 스트립을 보낼 수 있습니다. 단, 한 번의 그리기 호출에 재질을 공유해야 합니다. `NVTristrip` 라이브러리는 이에 대한 소스 코드를 제공하는 http://developer.nvidia.com 에서 얻을 수 있습니다.
- **텍스처 페이지를 사용하십시오.** 서로 다른 객체가 서로 다른 텍스처를 사용하면 배치가 자주 나뉘어집니다. 여러 개의 텍스처를 하나의 2D 텍스처에 배열하고 텍스처 좌표를 적절하게 정리하면, 한 번의 그리기 호출로 여러 개의 텍스처를 사용하는 도형을 보낼 수 있습니다. 그러나 이 기법은 밉매핑 및 앤티앨리어싱에 관한 문제가 있습니다. 이러한 문제를 비켜가는 한 가지 기법은 개별 2D 텍스처를 큐브맵의 면에 넣는 것입니다. NVIDIA SDK 최신 버전에는 텍스처 페이지를 만들고 미리 보는 텍스처 아틀라스 도구가 포함되어 있습니다.
- **버텍스 쉐이더의 상수 메모리를 매트릭스에 대한 룩업 테이블로 사용합니다.** 다수의 작은 객체가 같은 재질을 사용하더라도 매트릭스 상태만 다르면, 배치가 깨지는 경우가 자주 발생합니다(예를 들어 숲 속에 잇는 비슷한 나무들). 이러한 경우, 여러 개의 매트릭스를 버텍스 쉐이더 상수에 로딩하고 객체별로 인덱스를 버텍스 포맷으로 상수 메모리에 저장합니다. 그러면, 이 인덱스를 사용하여 버텍스 쉐이더에 있는 상수 메모리를 조회하고 정확한 변형 매트릭스를 사용하여 한번에 N 개의 객체를 렌더링할 수 있습니다.
- **화면 내에 동일한 메시가 여러 개 존재한다면 geometry instancing 을 사용합니다.** 이 기법을 사용하면, 하나의 그리기 호출과 두 개의 버텍스

스트림으로 동일한 메시 객체를 여러 개 그릴 수 있습니다. 메시의 각 복사본 또는 "인스턴스"는 서로 다른 위치에서, 서로 다른 모습으로 그릴 수 있습니다. 두 개의 스트림 중 한 스트림은 인스턴스가 될 메시의 한 개 복사본을 포함하고 있고, 다른 스트림은 인스턴스별 데이터(예를 들어 세계 변환, 컬러 등)를 포함하고 있습니다. 그런 다음, 한 번 그리기 호출을 전송하여 몇 개의 인스턴스를 그리고 싶은지 알려줄 수 있습니다. 일반적으로, geometry instancing 은 많은 그리기 호출의 CPU 오버헤드를 줄여주기 때문에 낮은 폴리곤(100 개 이하)의 객체를 여러 개 가질 가장 유용합니다. geometry instancing 의 한 가지 예(전체 소스 코드 포함)도 최신 버전의 NVIDIA SDK 에 들어 있습니다.

- **배치 크기를 늘리기 위해 GPU 쉐이더 분기를 사용합니다.** 최신 GPU 는 쉐이더 내에 분기(branching)를 허용하는 유연한 버텍스/픽셀 처리 파이프라인을 가지고 있습니다. 예를 들어 두 개의 배치가 하나는 4-bone 스키닝 버텍스 쉐이더를 필요로 하고 다른 하나는 2-bone 스키닝 버텍스 쉐이더를 필요로 하기 때문에 분리되어 있다고 한다면, 그 대신 블렌딩 가중치(weight)를 누적하면서 필요한 본(bone) 수만큼 반복하고 그 가중치가 결국 1 이 될 때 루프를 빠져나오는 버텍스 쉐이더를 작성할 수 있을 것입니다. 이렇게 하면, 두 개의 배치를 하나로 합칠 수 있을 것입니다. 쉐이더 분기를 지원하지 않는 아키텍처에서 모든 것에 4 bone 버텍스 쉐이더를 사용하고 4 개 미만의 bone 으로 된 영향력을 가진 버텍스에 bone 가중치를 그냥 없앰으로써 쉐이더 주기를 희생하고 이와 비슷한 기능을 구현할 수 있습니다.
- **결정은 파이프라인에서 되도록 나중으로 미룹니다.** 광택을 위해 배치를 쪼개서 픽셀 쉐이더 상수값을 설정하는 것보다, 텍스처의 알파 채널을 사용하는 것이 더 빠릅니다. 마찬가지로, 텍스처에 쉐이딩 데이터를 넣으려면 더 큰 배치 전송을 고려할 수 있습니다.

6.3.4. 버텍스 전송 비용 줄이기

최신 애플리케이션에서는 버텍스 전송이 병목이 되는 경우가 드물지만, 이것이 문제가 되는 것도 불가능하지만은 않습니다. 버텍스, 또는 더 가능성이 희박하게는 인덱스의 전송 때문에 애플리케이션에 병목이 생긴다면, 다음과 같은 방법을 시도해 봅니다.

- 버텍스 포맷에서 가능한 한 작은 바이트를 사용합니다. 바이트(byte) 단위로 충분하다면 모든 것에 float 를 사용하지 않습니다(예를 들어 컬러).
- 버텍스 프로그램 내에서 유도가 가능한 버텍스 속성을 입력 버텍스 포맷 속에 저장하지 않고 생성합니다. 예를 들어 tangent, binormal, normal 의 경우는 두 가지만 있으면 나머지 하나는 버텍스 프로그램 내 교차곱을 사용하여 유도할 수 있기 때문에 세 개를 전부 버텍스 포맷으로 저장할 필요가 없습니다. 이 기법은 버텍스 전송률 대신 버텍스 처리 속도를 우선한 것입니다.
- **32 비트 인덱스 대신 16 비트 인덱스를 사용합니다.** 16 비트 인덱스는 가져오고 옮기는 비용이 싸고 메모리를 덜 필요로 합니다.
- **비교적 순차적으로 버텍스 데이터에 접근합니다.** 최신 GPU 캐시 메모리는 버텍스를 가져올 때 접근합니다. 여느 메모리 체계와 같이, 참조의 공간적 지역성(spatial locality of reference)은 캐시 적중률을 극대화하여, 대역폭 요구량을 줄입니다.

6.3.5. 버텍스 처리 최적화

버텍스 처리 역시 최신 GPU 에서는 병목이 되는 경우가 드물지만, 사용 패턴과 대상 하드웨어에 따라 문제가 될 가능성이 있습니다. 버텍스 처리가 애플리케이션의 병목점이라는 것을 알았다면, 다음과 같은 방법을 시도해 봅니다.

- **객체별 연산을 CPU 쪽으로 뺍니다.** 객체별 또는 프레임별로 바뀌는 연산은 편의를 위해 버텍스 쉐이더에서 이루어질 때가 많습니다. 예를 들어, 직선 광원(directional light) 벡터를 시선 공간(eye space)으로 변환하는 연산은 간혹 버텍스 쉐이더에서 이루어집니다. 대신, 이 연산의 결과는 프레임별로 바뀔 뿐입니다.
- post-TnL 버텍스 캐시를 위해 최적화합니다. 최신 GPU는 최근에 변환된 버텍스의 결과를 저장하는 작은 FIFO 캐시를 가지고 있습니다. 이 캐시가 적중하면 그 이전에 파이프라인에서 이루어진 모든 작업과 더불어, 모든 변환 및 조명 작업을 구할 수 있습니다. 이 캐시를 활용하려면, 반드시 indexed primitive를 사용해야 하고, 버텍스의 순서를 재배치하여 메시에서 참조의 지역성이 극대화되도록 해야 합니다. 이러한 작업에 도움이 되는 툴로는 D3DX 함수와 NVTriStrip이 있으며, 무료로 구할 수 있는 툴입니다. NVTriStrip은 http://developer.nvidia.com/object/nvtristrip_library.html에서 제공합니다.
- **처리할 버텍스 수를 줄입니다.** 이것은 근본적인 문제는 거의 못 되지만, 정적 LOD 세트와 같이, 단순한 LOD(level of detail) 계획을 사용하면 확실히 버텍스 처리 부하를 줄일 수 있습니다.
- **버텍스를 처리할 때 LOD 를 사용합니다.** 처리할 버텍스의 수를 줄이는 데 LOD 를 사용하는 것과 더불어, 실제 버텍스 연산 자체에 LOD 를 시도해 봅니다. 예를 들어, 멀리 있는 인물의 경우 완전한 4-bone 스키닝을 할 필요가 없습니다. 그리고 조명의 경우 더 비용이 저렴한 근사치로 할 수도 있을 것입니다. 재질이 멀티 패스로 이루어진다면, 멀리 있는 낮은 LOD 의 경우 패스의 수를 줄이면 버텍스 처리 비용을 줄일 수도 있을 것입니다.
- 정확한 좌표 공간을 사용합니다. 좌표 공간의 선택이 버텍스 프로그램에서 값을 연산하는 데 필요한 명령의 수에 영향을 미칠 때가 많습니다. 예를 들어, 버텍스 라이팅을 할 때, 버텍스 노멀이 객체 공간에 저장되어 있고 라이트 벡터가 시선 공간에 저장되어 있으면, 버텍스 쉐이더 내에서 두 벡터 중 어느 하나를 변환시켜야 할 것입니다. 그러나 CPU 에서 객체별로 라이트 벡터가 객체 공간으로 한번 변형되었다면, 버텍스별 변환은 필요 없을 것이며, 그에 따라 GPU 버텍스 명령도 줄어들 것입니다.
- **버텍스 분기를 사용하여 연산을 빨리 끝냅니다.** 버텍스 쉐이더에서 여러 개의 광원에 대해 루프가 돌아가고 있고 정상적인 low dynamic range[0..1] 라이팅을 사용하고 있다면, 1 에 대한 포화상태를 점검할 수 있지만, 광원에서 먼 곳으로 향하고 있다면 더 이상의 연산을 수행하지 않을 수 있습니다. 이와 비슷한 최적화가 스키닝의 경우에도 일어날 수 있는데, 이 경우에는 가중치 합계가 1 에 도달했을 때(따라서 모든 이어지는 가중치가 0 이 될 것이다) 연산을 멈출 수 있습니다. 이것은 GPU 가 버텍스 분기를 구현하는 방식에 따라 달라지며, 모든 아키텍처에 대해 성능 향상을 보장하지도 않는다는 점을 주의합니다.

6.3.6. 픽셀 쉐이딩 속도 향상

길고 복잡한 픽셀 쉐이더를 사용하고 있다면, 픽셀 쉐이더에 의한 제약을 받고 있을 가능성이 높을 때가 많습니다. 이 경우에는 다음과 같은 방법들을 시도해 보십시오.

- **먼저 깊이를 렌더링합니다.** 주된 쉐이딩 패스를 렌더링하기 전에 깊이(컬러 제외)만 렌더링하면, 픽셀 쉐이딩과 프레임 버퍼 메모리의 양을 줄임으로써 성능을 크게 증대할 수 있습니다. 특히, 깊이 복잡도가 높은 화면의 경우 더욱 그렇습니다. 깊이만으로 된 패스를 100% 활용하려면, 단순히 프레임 버퍼에 컬러 쓰기를 해제하는 것만으로는 부족합니다. 컬러는 물론 깊이에 영향을 미치는 쉐이딩 작업을 비롯하여, 픽셀에 대한 모든 쉐이딩 작업을 비활성화해야 할 것입니다.
- **이른 z 값 최적화로 픽셀 처리를 줄입니다.** 최신 GPU 는 눈에 안 보이는 픽셀을 쉐이딩하지 않는 데 전념하는 실리콘이 있지만, 이것은 현재 시점까지의 장면에 대한 정보에 좌우되므로, 대강 앞에서 뒤의 순서로 렌더링함으로써 판단에 큰 도움을 줄 수 있습니다. 또한, 앞에 나와 있듯이 별도의 패스에서 먼저 깊이를 규정하면 쉐이딩 깊이의 복잡도를 1 까지 효과적으로 줄일 수 있으므로 그 뒤에 이어지는 패스(여기에서, 비용이 많이 드는 모든 쉐이딩 작업이 이루어짐)를 크게 가속화할 수 있습니다.
- **복잡한 함수를 텍스처에 저장합니다.** 텍스처는 룩업 테이블로 광범위하게 사용할 수 있으며, 아울러 그 결과를 공짜로 필터링하는 장점을 덤으로 얻을 수 있습니다. 여기서 표준적인 예는 정규화 큐브맵(normalization cubemap) 이 있으며, 이것을 사용하면, 텍스처 룩업 한 번에 드는 비용을 소요하여 높은 정밀도에서 자의적인 벡터를 정규화할 수 있습니다.
- **픽셀 단위 작업을 버텍스 쉐이더로 옮깁니다.** 버텍스 쉐이더에 있는 객체별 작업을 CPU 로 옮겨야 하는 것과 같이, 버텍스별 연산(화면 공간 안에서 정확하게 선형 보간할 수 있는 연산도 포함)은 버텍스 쉐이더로 옮겨야 합니다. 일반적인 예로, 벡터 연산하기, 벡터를 좌표계 상에서 변환시키기 등이 있습니다.
- **가능한 한 낮은 정밀도를 사용합니다.** DirectX 9 와 같은 API 는 양을 위한 픽셀 쉐이더 코드나 낮은 정밀도로 작업할 수 있는 연산의 경우에 정밀도 힌트를 지정할 수 있습니다. 많은 GPU 는 이 힌트를 활용하여 내부 정밀도를 낮추고 성능을 향상시킬 수 있습니다.
- **불필요한 정규화를 피합니다.** 흔히 저지르는 실수 중 하나는 지나칠 정도로 정규화를 선호하여, 연산을 수행할 때 매 단계에서 모든 단일 벡터를 정규화하는 것입니다. 어떤 변환이 정규직교기저(orthonormal basis)에 의한 변환처럼 길이를 보존하는지, 그리고 어떤 연산이 큐브맵 룩업과 같이 벡터의 길이에 따라 달라지는지 파악합니다.
- **가능하면 50% 정밀도의 정규화를 사용합니다.** 50% 정밀도로 정규화하는 것은 NV4x 클래스의 GPU 에서는 거의 비용이 안 드는 연산입니다. HLSL 에서 정규화를 수행할 벡터에 'half' 유형을 사용합니다. DirexX 9 에서 ps_2_0 이상 버전의 어셈블리 쉐이더를 사용한다면, **nrm_pp** 명령어를

사용하십시오. (또는 그와 같은 모든 연산에 '_pp' 수정자를 사용하십시오.) HLSL 쉐이더를 테스트할 때에는 생성된 어셈블리를 점검하여, 'half' 데이터 유형을 적절하게 사용했는지 확인하기 위해 정규화를 하는 것에 해당되는 연산에서 '_pp' 수정자를 사용하고 있는지 확인하는 것이 좋습니다. HLSL 쉐이더에서 생성된 어셈블리는 fxc.exe 또는 FX Composer 의 Shader Perf 패널을 실행하여 확인할 수 있습니다. 또는 NVShaderPerf 의 명령줄 유틸리티를 사용할 수도 있습니다.

- **픽셀 쉐이더에 LOD 를 사용하는 것을 고려합니다.** 버텍스 LOD 만큼 영향력이 큰 것은 아니지만(왜냐하면 멀리 있는 객체는 원근법 때문에 픽셀 처리과정에 자연적으로 LOD 가 적용되기 때문), 표면에 있는 패스의 수를 줄여나가는 것과 동시에 원거리에서 쉐이더의 복잡도를 줄여 나가면 픽셀 처리 부하를 줄일 수 있습니다.

6.3.7. 텍스처 대역폭에 의한 제약을 받지 않도록 합니다.

자세한 내용은 아래

텍스처 대역폭 줄이기

- 를 참조하십시오.

6.3.8. 텍스처 대역폭 줄이기

애플리케이션이 메모리 대역폭에 의해 제약을 받는다는 것을 알았지만 텍스처 데이터를 가져올 때 대부분 그러하다면 다음과 같은 최적화 기법을 고려해 보십시오.

- **텍스처의 크기를 줄입니다.** 대상 해상도와 텍스처 좌표를 고려합니다. 가장 높은 밉-레벨을 사용자들이 사용하기를 기대한 적이 있습니까? 아니라면, 텍스처의 크기를 줄이는 것을 고려해 보십시오. 이것은 과부하된 프레임 버퍼 메모리가 강제하여 텍스처링이 비로컬(non-local) 메모리에서 발생할 경우(AGP 또는 PCI-Express 버스를 통한, 시스템 메모리와 같음) 특히 유용할 수 있습니다. NVPerfHUD 메모리 그래프는 여러 힙(heap)에 드라이버에 의해 할당된 메모리의 양을 보여주기 때문에, 이러한 문제를 진단하는 데 도움이 될 수 있습니다.
- **최소로 할 수 있는 모든 표면에 항상 MIP 매핑을 사용합니다.** MIP 매핑은 텍스처 알리아싱(aliasing)을 줄임으로써 더 나은 이미지 품질을 실현합니다. 흐릿해 보이지 않는 고품질 밉맵을 생성하고자 할 때 다양한 필터를 사용할 수 있습니다. NVIDIA는 포토샵 플러그-인, 명령줄 유틸리티, 라이브러리와 같은 최적의 밉맵을 생성하기 위한 툴들을 제공합니다. 자세한 내용은 http://developer.nvidia.com/object/nv_texture_tools.html을 참조합니다. MIP 매핑이 없으면, 포인트 샘플링에 한정될 수밖에 없습니다. 그러면, 바람직하지 않은 가물거리는 효과를 초래할 수도 있습니다.

  어떤 표면에 밉매핑을 적용하여 그 표면이 흐릿하게 보인다면, 밉맵을 비활성화하거나 큰 음수의 LOD 바이어스를 추가하고 싶은 유혹을 피하십시오. 대신 비등방성 필터링을 사용합니다.

- **모든 컬러 텍스처를 압축합니다.** 데칼 또는 디테일 텍스처에 사용하는 모든 텍스처는 압축해야 하며, 텍스처의 알파 필요에 따라 DXT1, DXT3, DXT5 중 하나를 사용합니다. 그러면 메모리 사용량이 줄어들 뿐만 아니라, 텍스처 대역폭 필요도 줄어들고 텍스처 캐시의 효율성도 좋아집니다.
- **꼭 필요한 것이 아니라면 비용이 많이 드는 텍스처 포맷은 피합니다.** 64 비트나 128 비트 부동소수점 포맷과 같이 대형 텍스처 포맷은 페칭할 때 확실히 훨씬 더 큰 대역폭을 필요로 합니다. 따라서 필요할 때에만 이 포맷을 사용합니다.
- 적당한 이방성 텍스처 필터링 수준을 사용합니다. 낮은 빈도의 텍스처와 이방성이 높은 필터링 수준을 사용하면 GPU 가 추가 작업을 하는데, 시각 품질을 개선하는 데 전혀 도움이 안 됩니다. 텍스처 대역에 의해 제한을 받는다면, 이방성을 가장 낮은 수준으로 유지하십시오. 그러면 충분히 훌륭한 이미지 품질을 구현할 수 있습니다. 이상적인 애플리케이션에서는 텍스처에 맞게 이방성을 설정해야 합니다.
- **꼭 필요한 것이 아니라면 삼중 선형 필터링은 해제합니다.** 삼중 선형 필터링(trilinear filtering)은 별도의 텍스처 대역폭을 소모하지 않을 때라고 해도, 대부분의 최신 GPU 아키텍처의 경우 픽셀 쉐이더에서 연산할 때 주기를 추가로

필요로 합니다. 밉레벨 변이를 금방 확인할 수 없는 텍스처에서는 삼중 선형 필터링 기능을 해제하여 fillrate 를 절약합니다.

6.3.9. 프레임 버퍼 대역폭 최적화

파이프라인의 최종 단계인 ROP 는 프레임 버퍼 메모리와 직접 연결되어 있으며, 단일 단계로서는 프레임 버퍼 대역폭을 가장 많이 소모합니다. 이 때문에, 대역폭이 애플리케이션의 문제인 경우, ROP 로 거슬러 올라갈 때가 많습니다. 다음은 프레임 버퍼 대역폭을 최적화하는 방법입니다.

- **먼저 깊이를 렌더링합니다.** 이것은 앞에 나와 있듯이 픽셀 쉐이딩 비용을 줄여줄 뿐만 아니라 프레임 버퍼 대역폭 비용까지 줄여줍니다.
- **알파 블렌딩을 줄입니다.** 알파 블렌딩은 프레임 버퍼에 대해 읽기와 쓰기를 모두 필요로 합니다. 따라서 대역폭을 두 배 소모할 가능성이 있습니다. 알파 블렌딩은 그 사용을 꼭 필요한 상황으로 한정하고, 높은 수준으로 알파 블렌딩을 수행한 깊이 복잡도의 경우를 주의하십시오.
- **가능하면 깊이 쓰기를 사용하지 않습니다.** 깊이 쓰기는 대역폭을 소모하는 또 다른 요소입니다. 따라서 파티클(particle)과 같이 알파 블렌딩이 적용된 효과를 렌더링할 때, 객체를 셰도우 맵에 렌더링할 때(사실, 컬러를 기반으로 한 셰도우 맵에 렌더링하는 경우, read 역시 비활성화할 수 있다), 그리고 다중 패스 렌더링(최종 깊이가 이미 깊이 버퍼에 써 있음)에서는 깊이 쓰기를 반드시 비활성화해야 합니다.
- **불필요한 컬러 버퍼 지우기를 피합니다.** 모든 픽셀이 프레임 버퍼에서 애플리케이션에 의해 덮어 쓰기가 보장된다면, 컬러를 지우는 것을 피하도록 합니다. 왜냐하면 귀중한 대역폭을 소모하기 때문입니다. 그러나, 많은 early-Z 최적화가 지워진 깊이 버퍼의 결정적 컨텐츠에 의존하기 때문에, 될 수 있으면 매번 깊이와 스텐실 버퍼를 지워야 합니다.
- **앞에서 뒤로 렌더링합니다.** 앞에서 언급했듯이, 앞에서 뒤로 렌더링하는 것에 대한 픽셀 쉐이딩 장점 이외에도, 프레임 버퍼 대역폭의 영역에는 이와 비슷한 장점이 있습니다. 왜냐하면, early-Z 하드웨어 최적화가 불필요한 프레임 버퍼 읽기 및 쓰기를 버릴 수 있기 때문입니다. 사실, 이러한 최적화가 없는 구식 하드웨어라도 이 방식에서 장점을 얻을 수 있을 것입니다. 왜냐하면, 더 많은 픽셀이 깊이 테스트를 통과하지 못하여 프레임 버퍼에 컬러와 깊이를 쓰는 회수가 줄어들기 때문입니다.
- **스카이박스 렌더링을 최적화합니다.** 스카이박스는 프레임 버퍼 대역폭에 의해 제약을 받을 때가 많습니다. 그러나 어떻게 최적화할지 결정을 해야 합니다. 스카이박스를 제일 나중에 렌더링하고 깊이(쓰기가 아님)를 읽은 후, 일반적인 깊이 버퍼링과 더불어 early-Z 최적화를 활성화하여 대역폭을 줄이는 한 가지 방법이 있습니다. 그리고 스카이박스를 먼저 렌더링하고 읽기와 쓰기를 모두 비활성화하는 방법이 있습니다. 이 중 어떤 기법이 대역폭을 더 많이 절약할 수 있느냐 하는 것은 스카이박스 중에 얼마나 최종 프레임상에서 보이느냐 하는 것과 대상 하드웨어에 달려 있습니다. 스카이박스의 상당 부분이 보이지

않는다면 전자의 기법이 더 나을 것이고, 그렇지 않다면 후자가 더 많은 대역폭을 절약할 수 있을 것입니다.

- **필요할 때에만 부동소수점 프레임 버퍼를 사용합니다.** 이것은 더 작은 정수 포맷보다 확실히 훨씬 더 큰 대역폭을 소모합니다. 이는 여러 개의 렌더 대상의 경우에도 마찬가지로 적용됩니다.
- **가능하면 16 비트 깊이 버퍼를 사용합니다.** 깊이 트랜잭션은 많은 양의 대역폭을 소모합니다. 그래서 32 비트 대신에 16 비트를 사용하는 것이 훨씬 이득이 클 수 있으며, 스케일이 작고 스텐실이 필요 없는 실내 장면에서는 16 비트로도 충분한 효과를 낼 때가 많습니다. 또한, 동적 큐브맵(dynamic cubemap)과 같이 깊이를 필요로 하는 텍스처 렌더링의 경우 16 비트로도 충분할 때가 많습니다.
- **가능하면 16 비트 컬러를 사용합니다.** 이것은 특히 텍스처 렌더링 효과에 해당됩니다. 동적 큐브맵과 투사된 컬러 셰도우 맵과 같이 많은 경우에 16 비트 컬러로 충분한 효과를 보여주기 때문입니다.

Chapter 7. 문제 해결

여러분의 질문과 의견을 항상 기다리고 있습니다. 개발자 포럼에 참여하시거나 NVPerfHUD@nvidia.com으로 이메일을 보내 주십시오.

7.1. 알려진 문제점

- NVPerfHUD 는 여러 대의 장치를 지원하지 않습니다. **`Direct3DCreate9()`**로 작성된 첫 번째 장치만을 지원합니다.
- 소프트웨어 버텍스 처리를 사용할 때 NVPerfHUD 가 충돌할 수 있습니다.
- **`rdtsc`** 를 원시적으로 사용하는 애플리케이션은 NVPerfHUD 의 프레임 분석 모드에서 제대로 작동할 수 없습니다. 해결 방법은 아래 문제해결 부분을 참조하십시오.
- 프레임 분석 모드에서는 애플리케이션이 **`QueryPerformanceCounter()`** 또는 **`timeGetTime()`** win32 함수를 사용하고 이에 의존할 것을 요구합니다. 애플리케이션은 경과 시간(dt) 계산을 강력하게 처리해야 하며 **`dt`** 가 영(0)일 경우에는 더욱 그러합니다. 다시 말해, 프로그램을 **`dt`** 로 구분하지 말아야 한다는 뜻입니다. 이러한 문제가 나타나는 경우에는 NVPerfHUD 구성 대화 상자에서 Delta Time 옵션을 “SlowMo”로 설정해 보십시오. 문제가 해결되면 **`dt`** 가 영(0)일 때 상황을 처리할 수 있도록 애플리케이션을 수정하는 것도 좋습니다.
- Index Unit State Inspector 는 점 및 선 디스플레이를 지원하지 않습니다.
- 애플리케이션이 고정된 T&L 을 사용할 때, State Inspector 는 세부 정보를 표시하지 않습니다.
- 프레임 분석 모드를 사용할 때, (작동 단축키를 사용하여) NVPerfHUD 를 비활성화로 설정하고 애플리케이션을 종료하면 "종료가 0 이 아닐 때 레퍼런스 수"를 나타내는 경고 메시지가 표시됩니다.
- NVPerfHUD 가 활성화되어 있는 경우에는 닫기 버튼을 클릭해도 창 모드에서 실행되는 애플리케이션이 종료되지 않을 수 있습니다.

7.2. 질문 및 대답(FAQ)

내 애플리케이션에서 NVPerfHUD 분석을 사용할 수 없다는 메시지가 NVPerfHUD 에 표시됩니다.

승인 받지 않은 제 3 자가 사용자의 애플리케이션을 허가 없이 분석하지 않도록 하려면, 대칭 수정을 통해 NVPerfHUD 분석을 활성화해야 합니다. 자세한 내용은 이 사용 설명서의 시작하기 부분을 참조하십시오.

NVPerfHUD 가 활성화되어 있을 때 내 애플리케이션이 응답하지 않습니다.

단축키로 NVPerfHUD 를 활성화해둔 경우에는 모든 키보드 입력이 무시되며 어떠한 키 스트로크 이벤트도 애플리케이션에 전달되지 않습니다. 선택한 작동 단축키로 이 모드를 켜거나 끌 수 있습니다.

NVPerfHUD 헤더가 화면 상단에 표시되어 있지만 작동 단축키에 응답하지 않습니다.

NVPerfHUD 는 키 스트로크 이벤트를 가로채는 몇 가지 방법을 사용합니다. 현재 지원하지 않는 방법을 사용하고 계신 경우에는 NVPerfHUD 를 업데이트할 수 있도록 저희에게 알려 주십시오.

Win2K NVPerfHUD 에서는 작동 단축키에 귀를 기울이고, 활성화 상태에서 키보드 명령을 가로채도록 DirectInput 을 사용한다는 점을 잊지 마십시오. DirectInput 은 '버퍼링된 데이터와' 및 '즉각 데이터' 등 두 가지 데이터 종류를 제공합니다. 버퍼링된 데이터는 애플리케이션에서 검색할 때까지 저장되어 있는 이벤트 기록입니다. 즉각 데이터는 장치의 현재 상태에 대한 스냅샷입니다.

이는 병목 확인 실험, 쉐이더 시각화 등과 같은 NVPerfHUD 의 고급 기능을 사용할 때, IDirectInputDevice8::GetDeviceState 인터페이스가 아닌 IDirectInputDevice8::GetDeviceData 인터페이스를 사용할 필요가 있다는 것을 나타냅니다.

NVPerfHUD 가 알파 및 일부 렌더링 상태에 영향을 미칩니다.

NVPerfHUD 는 프레임 마지막에 HUD 를 렌더링합니다. 그 뿐 아니라, 그리기 자체에서 렌더링 상태를 변경하지만 원래 상태로 이를 복원하지는 않습니다. 다시 말해, 성능상의 이유로 렌더링 상태를 압박하거나 급하게 처리하지 않는다는 뜻입니다. 그렇기 때문에, 각 프레임 시작 부분에서 애플리케이션이 렌더링 상태를 리셋하는 것처럼 가정합니다.

GPU_IDLE (녹색) 라인이 어떤 데이터도 보고하지 않습니다.

GeForce4 (NV25) 및 기존 GPU 에서는 이 정보를 사용할 수 없습니다. NVPerfHUD 가 디스플레이 드라이버와 제대로 통신할 수 없는 경우에는 새 GPU 에서도 이 문제가 나타날 수 있습니다. 최신 NVPerfHUD 버전 및 최신 NVIDIA 디스플레이 드라이버를 사용하고 있는지 확인하십시오.

성능 그래프의 컬러 라인이 표시되지 않거나 모두 영(0)이지만 NVPerfHUD 헤더 및 배치 히스토그램은 작동합니다.

DirectX 9 DEBUG 런타임을 사용할 때에는 이 기능을 활성화할 수 없습니다. 정확한 성능 분석을 위해, DirectX 제어판에서 DEBUG 런타임을 RETAIL 런타임으로 전환하십시오. 애플리케이션의 함수 문제를 확인하기 위해, NVPerfHUD 디버그 콘솔 모드만을 사용하는 경우에는 DEBUG 런타임을 사용해도 상관 없습니다.

내 NVPerfHUD 그래프에 불필요한 라인이 몇 개 보입니다.

(특히 Win2K 에서) 기존 드라이버를 사용하는 경우에는 NVPerfHUD 그래프 위쪽에 기존 NVPerfHUD 1.0 그래프가 겹쳐 나타날 수 있습니다. 문제를 해결하려면, 71.8x 이상으로 드라이버를 업그레이드하십시오.

프레임 분석 모드의 내 장면에서 일부 객체가 계속 애니메이션으로 나타납니다.

프레임 분석 모드로 전환하면 NVPerfHUD 가 애플리케이션의 시계를 중단시키므로, 정지 상태에서 현재 프레임을 상세하게 분석할 수 있습니다. 애플리케이션은 QueryPerformanceCounter() 또는 timeGetTime() win32 함수를 사용하며 이에 의존해야 합니다. 애플리케이션에서 **rdtsc** 명령을 사용하는 경우에는 프레임 분석 모드가 제대로 작동하지 않습니다.

프레임 기반 애니메이션을 사용하는 애플리케이션에서는 애니메이션 객체에 정지 시간이 영향을 미치지 않습니다.

여기 수록되지 않은 문제를 발견했습니다.

저희는 NVPerfHUD 가 애플리케이션 분석을 위한 유용한 도구로 개발자들에게 꾸준한 도움을 드리기를 원하고 있습니다. NVPerfHUD 를 사용하면서 문제를 발견하시거나 더 나은 기능을 발견하신 경우에는 저희에게 알려 주십시오.

NVPerfHUD@nvidia.com

부록 A.
드라이버는 왜 GPU 를 원하는가

GPU 를 완벽하게 활용한다는 것은 프로세서 두 개를 FIFO 로 연결한 일반적인 상황에서, 처리 능력 이상의 데이터를 다룰 수 있도록 한쪽 칩이 다른 쪽을 지원하는 것입니다.

아래 나타난 것과 같이, CPU 는 처리할 수 있는 것보다 많은 명령으로 GPU 를 지원하고 있습니다. 이러한 현상이 나타나면, 모든 명령이 "push buffer"라고도 불리우는 FIFO 대기열에서 구축되기 시작합니다. 이 FIFO 가 오버플로우되지 않도록 하기 위해, 드라이버는 FIFO 에 새 명령을 실행할 공간이 생길 때까지 대기 상태를 유지합니다.

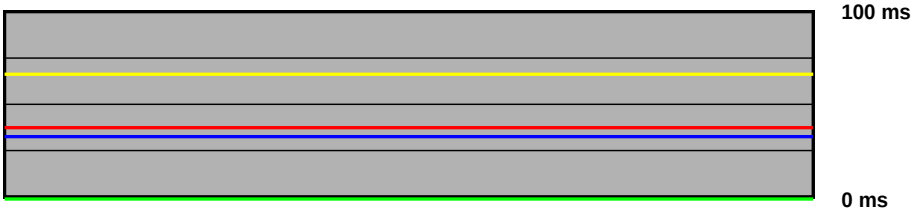


그림 20. GPU 를 기다리는 드라이버

프레임 속도에 따른 설명

- 프레임 속도가 **높음**이면 CPU 에서 더 많은 작업을 할 수 있으며, 프레임 속도(객체 컬링, 물리, 로직, 인공지능 등...)에 영향을 미치지 않습니다.
- 프레임 속도가 **적절하지 않음**이면, 장면 복잡성을 줄여 GPU 로드를 덜어 주어야 합니다.

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