NVIDIA DLSS(버전 3.1.13)

프로그래밍 가이드

문서 개정: 3.1.13

출시되었습니다: 출시: 2023년 5월

목차

콘텐츠

초록

DLSS 프로그래밍 가이드에서는 게임 또는 3D 애플리케이션에 DLSS를 통합하고 배포하는 방법에 대한 자세한 내용을 제공합니다. 또한 이 가이드는 임베디드 샘플 코드와 전체 샘플 구현에 대한 링크를 GitHub에 제공합니다.

언리얼 엔진 또는 Unity에서 DLSS를 사용하는 방법에 대한 자세한 내용은 각각 https://www.unrealengine.com/marketplace/en-US/product/nvidia-dlss 및 https://docs.unity3d.com/2021.2/Documentation/Manual/deep-learning-super-sampling.html 을 참조하세요.

개정 내역

개정	**변경 사항**	**날짜**
3.1.13	- 성능 품질 모드로 DLAA 추가 - NVSDK_NGX_DLSS_Hint_Render_Preset_G Enum 추가	5/17/2023
3.1	- GetFeatureReqs API 추가 - 업데이트 기능 API 추가 - 벌칸 확장 쿼리 API 추가	1/19/2023
2.5.1	-사전 설정 선택 섹션 추가	1/4/2023
2.4.1	-레거시 바닥글 정리하기	10/24/2022
2.4	- 사소한 버그 수정 및 성능 개선 - Streamline SDK 포인터 추가 - Linux 기능 섹션 추가	3/21/2022
2.3.2	- DLSS 샤프닝에 대한 참조 제거 - 사용자 혼동으로 인해 실행 모드에서 UltraQuality에 대한 참조를 제거했습니다.	03/08/2022
2.3.1	-노출 규모 문제를 진단하는 데 도움이 되는 디버그 시각화가 추가되었습니다.	09/30/2021

2.2.3	- 선명도와 복잡한 디테일을 보존하기 위해 텍스처 LOD 바이어스 권장 사항을 조정했습니다. - 고급 지터 디버깅 섹션에 몇 가지 세부 정보를 추가했습니다. - 얇은 피처의 모양을 개선하기 위해 보수적인 래스터를 사용하는 방법에 대한 섹션이 추가되었습니다.	09/02/2021

2.2.2	- DLSS SDK에 대한 더 쉬운 액세스를 기반으로 업데이트되었습니다. - Linux 지원 업데이트	7/12/2021
2.2.1	- NVSDK_NGX_VULKAN_RequiredExtensions() 사용법 업데이트 - 벌칸 애플리케이션이 벌칸 1.1 이상에서 실행되어야 함을 명확히 했습니다.	5/26/2021
2.2.0	- DLSS 자동 노출에 대한 섹션 추가 - 예상 노출값을 약간 명확히 했습니다.	4/9/2021
2.1.10	-DLS 신청 절차에 대한 참조 제거	3/3/2021
2.1.9	-벌칸의 리소스 상태 업데이트.	2/12/2021
2.1.8	-거부된 기능에 대한 재정의가 추가되었습니다.	1/29/2021
2.1.7	- NGX SDK에 새로운 진입점을 추가했습니다. - 새로운 항목 매개변수에 대한 정보를 추가했습니다. - 새로운 반환 오류 코드에 대한 정보를 추가했습니다. - 뎁스 버퍼가 필요한 포스트 프로세싱 셰이더에 대한 정보 추가	1/13/2021
2.1.6	-3.6.3 모션 벡터 스케일 섹션 추가	12/10/2020
2.1.5	-멀티 뷰에서 DLSS의 사용법을 약간 명확히 하여 VR뿐만 아니라 모든 멀티 뷰 사용 사례에서 사용할 수 있음을 명확히 했습니다.	12/2/2020
2.1.4	- SDK 초기화 시 SDK에 전달된 SDK API 버전 값의 올바른 사용에 대한 정보를 추가했습니다. - NGX 앱 로깅 후크 API에 대한 정보 추가	11/2/2020
2.1.3	- VRAM 사용에 대한 섹션 추가 - 현재 프레임 편향에 대한 섹션 추가 - 깃허브의 샘플 코드 링크 수정 - 모션 벡터 해상도 및 확장 요구 사항 명확화 - 지터 오프셋 디버그 오버레이에 섹션 추가	9/18/2020

	- 업데이트된 DLSS 실행 시간	
2.1.2	- 새로운 매개 변수를 포함하도록 섹션 5 수정 - 고주파 텍스처의 밉 매핑 섹션에 주의 사항 추가	7/2/2020
	-뎁스 버퍼 섹션에서 사용하지 않는 매개변수 제거	
2.1.1	- 밉맵 바이어스 요구 사항 명확화 - 동적 해상도에 대한 섹션 추가 - VR 애플리케이션의 DLSS에 대한 섹션 추가 - 노출 및 사전 노출 파라미터에 대한 섹션 추가 - DLSS 선명도 섹션 추가 - 부록에서 DLSS v1 전환 노트 제거	6/26/2020
2.0.2	-뎁스 버퍼 파라미터를 설명하는 섹션이 추가되었습니다.	4/10/2020
2.0.1	- 스왑지터 디버깅 핫키 명확화 - JitterConfig 디버깅 설정 목록 섹션 추가	3/26/2020
2.0.0	- 문서 전반의 일반적인 편집을 통해 DLSS v2에 맞게 업데이트합니다. - 렌더러 요구 사항 추가 - DLSS 실행 시간 추가 - 배포 체크리스트 추가 - 지터 문제 해결에 대한 섹션 추가 - 리소스 상태에 대한 섹션 추가 - DLSS 디버그 로깅에 대한 섹션 추가	3/23/2020
1.3.9	- 벌칸 리소스 메타데이터 래퍼 추가 - 디버그 오버레이로 버그 수정	11/08/2019

1.3.8	- 모션 벡터를 더 잘 설명하고 다이어그램을 추가했습니다. - 예상되는 지터 및 새로운 디버그 모드에 대한 세부 정보 추가 - LDR/HDR 처리 모드가 더욱 명확해졌습니다. - 최근 SDK에 추가된 두 가지 리소스 및 관련 데이터 구조인 카메라 벡터와 변환 행렬을 제거했습니다. - SDK 열거형 및 구조체 명명법을 정책에 맞게 수정했습니다. - 기능 DLL을 검색할 수 있는 SDK API 엔트리 포인트에 기능 경로 목록 매개 변수를 추가했습니다.	11/05/2019
1.3.7	-최소 드라이버 버전 확인을 위한 샘플 코드 추가	10/15/2019
1.3.6	- 화면 디버그 정보 관련 정보 추가 (3.7) - LDR/SDR 및 HDR의 정의 추가(3.1.3) - NVIDIA 연구진이 고려 중인 향후 DLSS 리소스를 설명하는 섹션이 추가되었습니다.	10/10/2019
	-코드 통합 섹션의 문구 명확화	
1.3.5	-지터 패턴 추천 관련 섹션 3.5 업데이트	9/13/2019
1.3.4	- 밉맵 LOD 바이어스 섹션 - 다양한 버퍼에 대한 필수/지원 형식 명확화 - 새로운 매개변수, 플래그 및 옵션에 대한 초기화 및 기능 생성 섹션(5.x)이 업데이트되었습니다.	9/11/2019
1.3.3	- 파이프라인 배치 섹션이 두 가지 배치 옵션(포스트 처리 전 및 포스트 처리 후)을 모두 포함하도록 업데이트되었습니다. - 스페큘러 앨리어싱에 대한 문제 해결 섹션 추가	9/06/2019
1.3.2	-3.1 섹션 3.1 파이프라인 배치 업데이트	9/06/2019
1.3.1	- 손쉬운 통합을 위한 다양한 섹션 업데이트 - DLSS 1.2.x에서 DLSS 1.3.x 섹션 9.1로의 전환이 추가되었습니다. - DLSS 1.3.x 섹션 9.2에 새로운 매개변수 추가	8/28/2019

1.2.0.0	- 승인 요건에 대한 공지 추가 - 지터에 대한 섹션 추가 - 모션 벡터 요구 사항 명확화 - 버퍼 포맷 전용 섹션 추가	8/15/2019
1.1.0.0	- 모션 벡터, 포맷 지원 및 파이프라인 통합을 위한 섹션이 추가되었습니다. - 문서 버전 번호를 DLSS 릴리스 버전에 맞추기 - 디버그 오버레이 지원 - 더 이상 사용되지 않는 스크래치 버퍼 설정 - 샘플 코드 링크 추가	7/08/2019
1.0.0.7	- 벌칸 타이틀 지원 - VS 2012 및 VS 2013 정적 라이브러리 포함	5/17/2019
1.0.0.6	- DLSS 샘플 코드 포함(섹션 5 및 6) - SDK 문서에 RTX 개발자 가이드라인 포함 - 일반 문서 정리	4/10/2019
1.0.0.5	-초기 릴리스	2019년 3월

1 소개

NVIDIA DLSS 기술은 고성능 라이브러리에서 기능 향상, 안티앨리어싱 및 업스케일링과 같은 스마트한 기능을 제공합니다. 이 라이브러리는 NVIDIA RTX GPU의 최신 기능을 활용하도록 조정되었습니다. 개발자는 DLSS를 사용하여 높은 프레임 속도를 유지하면서 시각적 경험을 향상시키는 고급 렌더링 기술과 효과에 더 많은 프레임 시간을 할애할 수 있습니다.

DLSS는 NVIDIA RTX(https://developer.nvidia.com/rtx)의 핵심 구성 요소 중 하나인 NVIDIA NGX의 기능으로 빌드 및 배포됩니다. NVIDIA NGX를 사용하면 사전 구축된 AI 기반 기능을 게임과 애플리케이션에 쉽게 통합할 수 있습니다. NGX 기능인 DLSS는 NGX 업데이트 기능을 활용합니다. NVIDIA가 DLSS를 개선하면 NGX 인프라를 사용하여 현재 게임이 설치된 모든 클라이언트에서 특정 타이틀에 대한 DLSS를 업데이트할 수 있습니다.

기본 NGX 시스템을 구성하는 세 가지 주요 구성 요소가 있습니다:

1. **NGX SDK**: 이 SDK는 애플리케이션이 NVIDIA에서 제공하는 AI 기능을 사용할 수 있도록 CUDA, Vulkan, DirectX 11 및 DirectX 12 API를 제공합니다. 이 문서에서는 DLSS SDK를 사용하여 게임이나 애플리케이션에 DLSS 기능을 통합하는 방법을 다룹니다(NGX SDK를 모델로 하지만 별도로 존재).
2. **NGX 코어 런타임**: NGX 코어 런타임은 각 기능 및 애플리케이션(또는 게임)에 대해 로드할 공유 라이브러리를 결정하는 시스템 구성 요소입니다. NGX 런타임 모듈은 지원되는 NVIDIA RTX 하드웨어가 감지되면 항상 NVIDIA 그래픽 드라이버의 일부로 최종 사용자의 시스템에 설치됩니다.
3. **NGX 업데이트 모듈**: NGX 기능(DLSS 포함)에 대한 업데이트는 NGX 코어 런타임 자체에서 관리합니다. 게임이나 애플리케이션이 NGX 기능을 인스턴스화하면 런타임은 NGX 업데이트 모듈을 호출하여 사용 중인 기능의 새 버전을 확인합니다. 최신 버전이 발견되면 NGX 업데이트 모듈은 호출하는 애플리케이션의 DLL을 다운로드하여 교체합니다.

2 시작하기

2.1 시스템 요구 사항

Windows 및 Linux용 DLSS를 로드하고 실행하려면 다음이 필요합니다:

- NVIDIA RTX GPU(지포스, 타이탄 또는 쿼드로)
- 최신 NVIDIA 그래픽 드라이버를 권장합니다. DLSS 2.4.11+를 실행하려면 최소한 2022년 3월 3일 이후에 발급된 NVIDIA 드라이버(예: 512.15)가 있어야 합니다. Linux 드라이버 호환성 정보는 섹션 9.6을 참조하세요.

Windows에서 DLSS를 로드하고 실행하려면 다음이 필요합니다:

- Windows 10 v1709(2017년 가을 크리에이터 업데이트 64비트) 이상이 설치된 Windows PC

- DLSS SDK를 게임에 통합하기 위한 개발 환경은 다음과 같습니다:
    - Microsoft Visual Studio 2015 이상.
    - Microsoft Visual Studio 2012 및 2013은 지원되지만 향후 더 이상 사용되지 않을 수 있습니다.

Linux에서 DLSS를 로드하고 실행하려면 다음이 필요합니다:

- Linux 커널, 2.6.32 이상
- DLSS SDK: glibc 2.11 이상 버전
- DLSS SDK 샘플 코드: gcc 및 g++ 8.4.0 이상 버전

2.2 렌더링 엔진 요구 사항

DLSS 알고리즘은 일련의 프레임에 걸쳐 수집된 정보로부터 고해상도 출력 버퍼를 구축합니다. 이 문서에는 DLSS를 올바르게 통합하는 데 필요한 사항이 자세히 설명되어 있으며 전체 내용을 읽어보셔야 합니다. 요약하자면, DLSS가 고화질의 이미지로 작동하려면 렌더링 엔진이 **반드시 필요합니다**:

- DirectX11, DirectX 12 또는 Vulkan 기반
    - **벌칸에 대한 추가 참고 사항**: DLSS의 벌칸 경로는 애플리케이션이 벌칸 버전 1.1 이상에서 실행될 것으로 예상합니다.
- 각 평가 호출(즉, 각 프레임)에서 다음을 제공합니다:
    - 프레임의 원시 색상 버퍼(HDR 또는 LDR/SDR 공간)입니다.
    - 화면 공간 모션 벡터: 정확하고 픽셀당 16비트 또는 32비트로 계산되며, 매 프레임마다 업데이트됩니다.
    - 프레임의 깊이 버퍼입니다.
    - 노출 값(HDR 공간에서 처리하는 경우)입니다.
- 서브 픽셀 뷰포트 지터를 허용하고 최소 16개 이상의 지터 페이즈(32개 이상 권장)로 픽셀 커버리지가 양호해야 합니다.

- 유효한 `ProjectID`를 사용하여 NGX 및 DLSS를 초기화합니다(5.2.1절 참조).
- 텍스처와 지오메트리에 대한 LOD를 네거티브 바이어스하는 기능입니다.

향후 호환성을 확보하고 NVIDIA의 지속적인 연구를 용이하게 하기 위해 엔진은 *선택적으로 다음을 수행할* 수도 있습니다.
추가 버퍼와 데이터를 제공합니다. 이에 대한 자세한 내용은 9.4항을 참조하세요.

2.3 최종 사용자 시스템에서 DLSS 지원 쿼리하기

통합 애플리케이션은 최종 사용자 시스템에 DLSS 지원에 적합한 비트가 있는지 쿼리할 수 있습니다. 또한 실제 디바이스가 생성되거나 NGX가 초기화되기 전에 '

GetFeatureRequirements ' API:

```
NVSDK_NGX_Result NVSDK_NGX_D3D11_GetFeatureRequirements(
IDXGIAdapter *Adapter,
const NVSDK_NGX_FeatureDiscoveryInfo *FeatureDiscoveryInfo,
NVSDK_NGX_FeatureRequirement *OutSupported);
```

참고: 이 도우미는 정적 시스템 조건에 대해서만 NGX 기능 지원을 감지하며, 다른 SDK 진입점을 사용하는 경우 런타임에 기능이 지원되지 않는 것으로 판단될 수 있습니다(즉, 사용 가능한 디바이스 메모리가 충분하지 않을 수 있습니다).

FeatureDiscoveryInfo에는 기능 검색, 초기화 및 로깅에 필요한 모든 NGX 기능에 공통된 정보가 포함되어 있습니다:

OutSupported는 *NVSDK_NGX_FeatureRequirement* 구조체에 대한 포인터입니다. 애플리케이션은 *NVSDK_NGX_Result_Success가* 반환되면 OutSupported에서 반환된 값을 확인해야 합니다.

```
유형 정의 구조체 NVSDK_NGX_FeatureDiscoveryInfo
{
    NVSDK_NGX_버전 SDKVersion; NVSDK_NGX_기능
    FeatureID; NVSDK_NGX_응용프로그램_식별자 식별
    자; const wchar_t* ApplicationDataPath;
    const NVSDK_NGX_FeatureCommonInfo* FeatureInfo;
} NVSDK_NGX_FeatureDiscoveryInfo;
```

SDK 버전은 *NVSDK_NGX_Version_API로* 설정해야 합니다.

DLSS의 경우 **FeatureID를** *NVSDK_NGX_Feature_SuperSampling으로 설정*합니다.

식별자는 NVSDK_NGX_Application_Identifier 유형입니다.

ApplicationDataPath는 로그 및 기타 임시 파일을 저장하는 로컬 시스템 경로입니다(쓰기 권한 필요).

FeatureInfo는 모든 NGX 피처에 공통으로 적용되는 정보입니다. 현재 NGX가 기능별 dll을 찾기 위해 스캔할 수 있는 경로 목록입니다. DLSS의 경우 - nvngx_dlss.dll

```
유형 정의 구조체 NVSDK_NGX_Application_Identifier
{
    NVSDK_NGX_응용 프로그램_아이덴티파이어_유형 식별자 유
    형; 유니온 v {유니온 {
        NVSDK_NGX_ProjectIdDescription ProjectDesc; 서명
        되지 않은 긴 긴 ApplicationId;
    } v;
} NVSDK_NGX_응용프로그램_식별자;
```

식별자 유형: NVSDK_NGX_ProjectIdDescription 또는 고유 애플리케이션 식별자 사용 여부

ProjectDesc: NVSDK_NGX_ProjectIdDescription을 사용하는 경우, 이 앱을 식별하는 데 필요한 프로젝트 ID와 엔진 정보가 포함됩니다.

ApplicationId: NVSDK_NGX_ProjectIdDescription을 사용하지 않는 경우, 여기에는 이 앱에 대해 NVIDIA에서 제공한 ID가 포함됩니다. NVIDIA 담당자가 이 용도로 ID를 제공하지 않은 경우, 엔진 유형으로 NVSDK_NGX_ENGINE_TYPE_CUSTOM과 함께 ProjectDesc를 사용합니다.

```
유형 정의 구조체 NVSDK_NGX_FeatureRequirement
{
    NVSDK_NGX_Feature_Support_Result FeatureSupported; 부
    호 없는 int MinHWArchitecture;
    char MinOSVersion[255];
} NVSDK_NGX_FeatureRequirement;
```

FeatureSupported: *NVSDK_NGX_Feature_Support_Result의* 값 중 하나입니다. DLSS의 경우, Is는 *NVSDK_NGX_FeatureSupportResult_ 지원되는* 경우 NVSDK_NGX_Feature_SuperSampling이 지원됩니다.

MinHWArchitecture: 기능 지원에 필요한 NvAPI GPU 프레임워크에 정의된 NV_GPU_ARCHITECTURE_ID 값에 해당하는 HW 아키텍처 값을 반환합니다.

MinOSVersion: NGX 기능 지원에 필요한 최소 OS 버전에 해당하는 문자열 값입니다.

2.3.1 벌칸 확장 쿼리

Vulkan API를 사용하는 애플리케이션은 '*GetFeatureDeviceExtensionRequirements*' 및 '*GetFeatureInstanceExtensionRequirements*' API를 사용하여 NGX 기능 지원에 필요한 Vulkan 디바이스/스턴스 확장을 식별해야 합니다:

```
NVSDK_NGX_Result NVSDK_NGX_VULKAN_GetFeatureDeviceExtensionRequirements(
const VkInstance 인스턴스,
const VkPhysicalDevice PhysicalDevice,
const NVSDK_NGX_FeatureDiscoveryInfo *FeatureDiscoveryInfo,
uint32_t *InOutExtensionCount,
VkExtensionProperties **OutExtensionProperties);


NVSDK_NGX_Result NVSDK_NGX_VULKAN_GetFeatureInstanceExtensionRequirements(
```

```
const NVSDK_NGX_FeatureDiscoveryInfo *FeatureDiscoveryInfo,
uint32_t *InOutExtensionCount,
VkExtensionProperties **OutExtensionProperties);
```

OutExtensionProperties가 NULL인 경우 InOutExtensionCount는 확장자 수로 채워집니다.

그렇지 않으면 **아웃익스텐션** 프로퍼티는 *VkExtensionProperties* 구조의 배열을 가리킵니다.

2.4 최신 업데이트 받기

이제 애플리케이션에서 무선(OTA) 업데이트를 통해 사전 설정(섹션 3.12)에 대한 업데이트를 수신하도록 선택할 수 있습니다. 다음 호출은 가능한 경우 이 업데이트를 트리거합니다.

```
NVSDK_NGX_Result NVSDK_NGX_UpdateFeature(
const NVSDK_NGX_응용프로그램_식별자 &ApplicationId,
const NVSDK_NGX_기능 FeatureID);
```

ApplicationId: NVSDK_NGX_ProjectIdDescription을 사용하지 않는 경우, 여기에는 이 앱에 대해 NVIDIA에서 제공한 ID가 포함됩니다. NVIDIA 담당자가 이 용도로 ID를 제공하지 않은 경우, 엔진 유형으로 NVSDK_NGX_ENGINE_TYPE_CUSTOM과 함께 ProjectDesc를 사용합니다.

FeatureID: 가용성을 쿼리 중인 DLSS v3 기능에 해당하는 유효한 NVSDK_NGX_Feature 열거형입니다.

업데이트가 성공하면 다음번 NGX가 초기화될 때까지(즉, NVSDK_NGX_<API>_Init()가 다시 호출될 때까지 업데이트가 적용되지 않습니다. 섹션 5.2 참조). **이 API를 사용하면 NVIDIA가 애플리케이션에 가장 최신의 IQ 및 성능 향상을 제공할 수 있으므로 이 API를 사용하는 것이 좋습니다.**

이 API는 자체 전용 스레드에서 호출해야 합니다. 업데이트를 다운로드하는 데 걸리는 시간은 네트워크 연결/인터넷 속도 등 통제할 수 없는 요인에 따라 달라질 수 있기 때문입니다.

DLSS 실행 시간 및 GPU RAM 사용량

DLSS의 정확한 실행 시간은 엔진마다 다르며 통합에 따라 달라집니다. 엔진 메모리 관리자의 작동 방식과 추가 버퍼 복사본이 필요한지 여부와 같은 요소가 최종 성능에 영향을 미칠 수 있습니다. 대략적인 가이드를 제공하기 위해 NVIDIA는 3D 렌더러 없이 명령줄 유틸리티를 사용하여(즉, 3D 렌더러 없이) 전체 범위의 NVIDIA GeForce RTX GPU에서 DLSS 라이브러리를 실행하고 예상 실행 시간의 대략적인 추정치로 이 결과를 제공했습니다. 개발자는 이 수치를 사용하여 DLSS가 제공하는 잠재적인 절감 효과를 추정할 수

있습니다.

이 테스트 시나리오에서는

1. DLSS 라이브러리는 내부적으로 GPU에 일부 RAM을 할당합니다. 할당된 메모리의 양은 NVSDK_NGX_DLSS_GetStatsCallback()을 사용하여 쿼리할 수 있습니다. 아래 표는 출력 해상도에 따라 할당된 대략적인 RAM 양을 보여줍니다:

	1920x1080	2560x1440	3840x2160	7840x4320
할당된 메모리	60.83 MB	97.79 MB	199.65 MB	778.3 MB

이 수치는 대략적인 수치이며 실제 수치는 다소 다를 수 있습니다. 예를 들어, InEnableOutputSubrects 플래그를 사용하면 메모리 사용량이 높아질 수 있습니다. 반면에 출력 색상 버퍼가 RGBA16 형식인 경우 DLSS가 내부 임시 데이터를 저장하는 데 사용할 수 있으므로 할당된 메모리 양이 더 적을 수 있습니다.

2. DLSS 알고리즘은 입력이 1/4인 16비트 '성능 모드'에서 실행됩니다.
   픽셀 수를 출력으로 지정합니다:

   - 1920x1080 결과는 960x540픽셀의 입력 버퍼 크기에서 생성됩니다.
   - 1280x720 픽셀의 입력 버퍼 크기에서 2560x1440 결과가 생성됩니다.
   - 3840x2160 결과는 1920x1080 픽셀의 입력 버퍼 크기에서 생성됩니다.

GeForce GPU	1920x1080	2560x1440	3840x2160	7680x4320
RTX 2060 S	0.61ms	1.01ms	2.18ms	10.07ms
RTX 2080 TI	0.37ms	0.58ms	1.26ms	5.52ms
RTX 2080 (노트북)	0.56ms	0.91ms	1.98ms	9.09ms
RTX 3060 TI	0.45ms	0.73ms	1.52ms	7.01ms
RTX 3070	0.41ms	0.66ms	1.36ms	5.77ms
RTX 3080	0.32ms	0.47ms	0.94ms	4.25ms
RTX 3090	0.28ms	0.42ms	0.79ms	3.45ms
RTX 4080	0.2ms	0.37ms	0.73ms	2.98ms
RTX 4090	N/A	N/A	0.51ms	1.97ms

2.5 DLSS 배포 체크리스트

통합 및 테스트 중에는 이 문서를 전체적으로 읽고 따르고, DLSS가 포함된 게임이나 애플리케이션을 출시하기 전에 품질이 높은지, 최소한 다음 사항이 사실인지 확인하세요.

항목	확인됨 ☑

워터마크가 없는 정식 프로덕션 DLSS 라이브러리(nvngx_dlss.dll)는 릴리스 빌드에 패키지로 제공됩니다(4.2 섹션 참조).	☐
게임별 애플리케이션 ID는 초기화 중에 사용됩니다(5.2.1 섹션 참조).	☐
DLSS가 활성화된 경우 밉맵 바이어스 설정(섹션 3.5 참조)	☐
모든 씬, 머티리얼 및 오브젝트의 모션 벡터가 정확합니다(섹션 3.6 참조).	☐
정적 장면 해결 및 호환 지터 확인(섹션 3.7 참조)	☐
노출 값이 매 프레임마다 올바르게 전송(또는 자동 노출이 활성화됨)됩니다(3.9절 참조).	☐
DLSS 모드가 UI에서 쿼리되고 사용자가 선택할 수 있거나(섹션 3.2.1 및 RTX UI 개발자 가이드라인 참조) 동적 해상도 지원이 활성화 및 테스트 중입니다.	☐

3 DLSS 통합

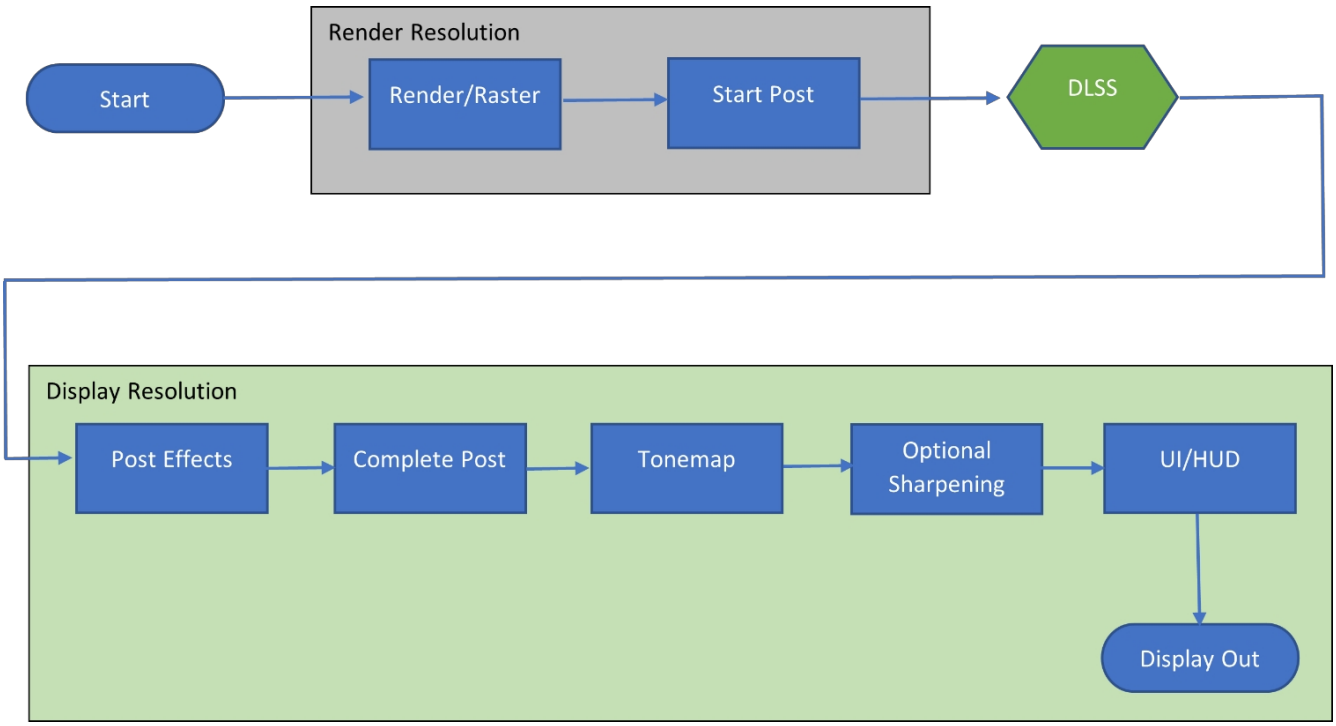
3.1 파이프라인 배치

DLSS 평가 호출은 톤 매핑 전 렌더링 파이프라인의 포스트 프로세싱 단계에서 이루어져야 합니다. 최상의 이미지 품질을 위해 가능한 한 포스트 프로세싱 시작에 가깝게(프레임에 효과가 적용되기 전 또는 가능한 한 적은 효과만 적용되기 전) DLSS를 배치하세요.

3.1.1 초기 단계 포스트 프로세싱 중 DLSS

DLSS는 포스트 프로세싱 파이프라인을 시작할 때 또는 아주 일찍 실행하는 것이 가장 좋습니다. DLSS를 호출하는 동안 프레임은 기능이 향상되고 앤티 앨리어싱이 적용된 상태로 처리되며 프레임의 해상도가 목표 해상도로 증가합니다.

명확하게 말하면, DLSS가 포스트 프로세싱의 시작 부분에 배치되면 모든 포스트 이펙트가 증가된 해상도를 처리할 수 있어야 하며 렌더링 해상도에서만 처리가 가능하다는 가정이 있어서는 안 됩니다.



3.1.2 LDR 및 HDR의 색상 범위

DLSS는 LDR(일명 SDR) 또는 HDR 값으로 저장된 컬러 데이터를 처리할 수 있습니다. DLSS의 성능은 LDR 모드에서 향상되지만 몇 가지 주의 사항이 있습니다.

1. LDR 모드의 색상 값 **범위는** 0.0에서 1.0 사이여야 합니다.

2. LDR 모드에서 DLSS는 낮은 정밀도로 작동하며 데이터를 8비트로 정량화합니다. 이 정밀도로 색 재현이 작동하려면 입력 색상 **버퍼가 지각적으로 선형이어야 합니다.**

인코딩(예: sRGB). 이는 톤 매핑 후에도 종종 발생하지만 보장되는 것은 아닙니다(예: HDR 디스플레이 패널이 감지되더라도 톤 매퍼가 여전히 선형 공간으로 출력하는 경우).

3. LDR 모드에서는 색상 데이터가 선형 공간으로 제공되어서는 **안 됩니다.** DLSS가 LDR 모드에서 선형 색상을 처리하는 경우 DLSS의 출력에 색상 밴딩, 색상 이동 또는 기타 시각적 아티팩트가 나타납니다.

DLSS로 전송된 입력 컬러 버퍼가 이러한 요구 사항을 **충족하는** 경우 "NVSDK_NGX_DLSS_Feature_Flags_IsHDR"을 "0"으로 설정하여 DLSS가 LDR을 사용하여 처리하도록 설정합니다.

DLSS로 전송된 입력 컬러 버퍼가 선형 공간에 저장되어 있거나 어떤 이유로든 위의 요구 사항을 **충족하지 않는** 경우 "NVSDK_NGX_DLSS_Feature_Flags_IsHDR"을 "1"로 설정합니다. HDR 모드는 내부적으로 높은 범위의 고정밀 색상으로 작동하며 모든 휘도 값을 처리할 수 있습니다(즉, HDR 값의 허용 가능한 휘도에 대한 상한이 없음).

"NVSDK_NGX_DLSS_Feature_Flags_IsHDR" 기능 플래그에 대한 자세한 내용은 섹션 5.3을 참조하세요.

3.2 통합 개요

DLSS 통합은 다음 단계로 구성됩니다:

1. NGX를 초기화하고 오류가 반환되지 않는지 확인합니다.
2. 시스템에서 DLSS를 사용할 수 있는지 확인합니다.
3. 각 디스플레이 해상도 및 DLSS 실행 모드에 대한 최적의 설정을 얻습니다(3.2.1절 참조).
4. 헬퍼 메서드를 사용하여 DLSS 기능을 생성합니다.
5. 최종 해상도로 업스케일링할 때 DLSS를 평가합니다.
6. 디스플레이 해상도, RTX 토글, 입력/출력 버퍼 형식 변경 등 DLSS에 영향을 미치는 설정이 변경되면 현재 기능을 해제하고 3단계로 돌아갑니다.
7. DLSS가 더 이상 필요하지 않은 경우 정리 및 종료 절차를 수행합니다.

중요: DLSS는 메인 렌더 타깃의 기본 업스케일 패스만 대체해야 하며 섀도, 반사 등과 같은 보조 버퍼에는

사용해서는 안 됩니다.

3.2.1 DLSS 실행 모드

DLSS는 임의의 입력 버퍼 크기를 처리하고 그 결과를 최종 디스플레이 해상도의 출력 버퍼 크기로 출력합니다. 게임에서 렌더링하고 처리를 위해 DLSS로 전송할 입력 해상도는 각 "PerfQualityValue" 옵션에 대한 "DLSS 최적 설정"을 쿼리하여 결정됩니다. 현재 6개의 "PerfQualityValue"가 정의되어 있습니다:

1. 성능 모드
2. 밸런스 모드
3. 품질 모드

4. 초고성능 모드
5. DLAA(딥러닝 앤티 앨리어싱) 모드

또한 "DLAA" 또는 딥러닝 앤티 앨리어싱은 최적 설정 호출과 관계없이 입력 및 출력 렌더링 크기가 동일한 값(예: 스케일링 비율 1.0)으로 설정된 경우를 말합니다. 이는 게임에서 다른 UI 옵션으로 노출되어야 하므로 별도로 유지됩니다. DLAA는 성능 품질 모드이기도 합니다.

사용 중인 DLSS 알고리즘과 게임 성능 수준에 따라 위에 나열된 모드 중 일부 또는 전부가 활성화될 수 있습니다. 모두 선택해야 하지만 특정 구성에서 모두 활성화되지 않는 경우도 있습니다.

RTX UI 개발자 가이드라인에는 DLSS 자동 모드에 대한 설명이 나와 있습니다. 이는 특정 실행 모드가 아니라 기본 모드를 현재 출력 해상도에 매핑하기 위한 사용자 친화적인 추가 모드입니다. (*DLSS 자동 모드 및 DLSS 모드 기본값 섹션을 참조하세요*).

게임 설정에서 **활성화된 모드만 표시합니다. 다른 모든 모드는 완전히 숨깁니다.** 활성화된 모드의 경우 최종 사용자가 각 활성화된 모드 간에 전환하여 렌더링 대상 해상도를 변경할 수 있도록 합니다.

DLSS 최적 설정을 쿼리한 후 두 개 이상의 모드가 활성화된 경우 사용자가 특정 모드를 선택하지 않은 경우 기본 선택은 "자동", "품질", "균형", "성능", "초성능"이 됩니다.

사용자 대면 DLSS 모드 선택을 표시하는 방법에 대한 자세한 내용은 "NVIDIA RTX UI 개발자 가이드라인"(최신 버전은 "docs" 디렉터리의 GitHub 저장소에 있음)을 참조하시기 바랍니다.

DLSS 최적 설정에 대한 자세한 내용은 5.2.8항을 참조하세요.

3.2.2 동적 해상도 지원

DLSS는 동적 해상도를 지원하므로 출력 크기는 고정된 상태로 유지하면서 입력 버퍼의 크기를 프레임마다 변경할 수 있습니다. 따라서 렌더링 엔진이 동적 해상도를 지원하는 경우 DLSS를 사용하여 디스플레이 해상도에 필요한 업스케일을 수행할 수 있습니다.

> **참고**: 출력 해상도(일명 디스플레이 해상도)가 변경되면 DLSS를 다시 초기화해야 합니다.

```
static inline NVSDK_NGX_Result NGX_DLSS_GET_OPTIMAL_SETTINGS(
    NVSDK_NGX_Parameter *pInParams,
    부호 없는 int InUserSelectedWidth, 부호
    없는 int InUserSelectedHeight,
    NVSDK_NGX_PerfQuality_Value InPerfQualityValue, 부호
    없는 int *pOutRenderOptimalWidth,
    부호 없는 int *pOutRenderOptimalHeight,
    부호 없는 int *pOutRenderMaxWidth, 부호
    없는 int *pOutRenderMaxHigh, 부호 없는
    int *pOutRenderMinWidth, 부호 없는 int
    *pOutRenderMinHigh, float
    *pOutSharpness)를 생성합니다.
```

동적 해상도와 함께 DLSS를 사용하려면 섹션 5.3에 설명된 대로 NGX 및 DLSS를 초기화합니다. DLSS 중

각 DLSS 모드 및 디스플레이 해상도에 대한 최적 설정 호출을 수행하면 DLSS 라이브러리는 "최적" 렌더링 해상도를 `pOutRenderOptimalWidth` 및 `pOutRenderOptimalHeight`로 반환합니다. 그런 다음 이 값을 다음 `NGX_API_CREATE_DLSS_EXT()` 호출에 주어진 대로 정확하게 전달해야 합니다.

DLSS 최적 설정은 DLSS 평가 호출 중에 사용할 수 있는 허용 가능한 렌더링 해상도 범위를 지정하는 네 가지 추가 파라미터도 반환합니다. 반환되는 `pOutRenderMaxWidth, pOutRenderMaxHeight` 및 `pOutRenderMinWidth, pOutRenderMinHeight` 값은 포괄적이며, 정확히 최소 또는 최대 치수 사이에 값을 전달할 수 있습니다.

```
유형 정의 구조체 NVSDK_NGX_Dimensions
{
    부호 없는 int 폭; 부
    호 없는 int 높이;
} NVSDK_NGX_Dimensions;

유형 정의 구조체 NVSDK_NGX_D3D11_DLSS_Eval_Params
{
---
    NVSDK_NGX_Dimensions                    InRenderSubrectDimensions;
---
} NVSDK_NGX_D3D11_DLSS_Eval_Params;


static inline NVSDK_NGX_Result NGX_D3D11_EVALUATE_DLSS_EXT(
    ID3D11DeviceContext *pInCtx,
    NVSDK_NGX_핸들 *pInHandle, NVSDK_NGX_파라미터
    *pInParams,
    NVSDK_NGX_D3D11_DLSS_Eval_Params *pInDlssEvalParams)
```

지원되는 그래픽 API에 대한 DLSS 평가 호출은 추가 평가 매개변수로 `InRenderSubrectDimensions`를 받을 수 있습니다. 이는 현재 프레임에 사용할 입력 버퍼의 영역을 지정하며 프레임마다 달라질 수 있습니다(따라서 동적 해상도 지원 가능).

하위 직사각형은 버퍼에서 `InColorSubrectBase` (또는 다른 `*SubrectBase` 매개변수)를 사용하여 서브렉트의 왼쪽 상단 모서리를 지정할 수 있습니다.

DLSS 평가에 전달된 InRenderSubrectDimensions가 지원되는 범위(DLSS 최적 설정 호출에 의해 반환됨)에 있지 않으면 DLSS 평가 호출이 실패하고 `NVSDK_NGX_Result_FAIL_InvalidParameter` 오류 코드가 발생합니다.

참고: PerfQuality 모드, 해상도 및 레이 트레이싱의 모든 조합이 동적 해상도를 지원하는 것은 아닙니다. 또한 초기 범위에 대해 선택한 프리셋은 기능의 수명 기간 동안 유지됩니다(스케일링 비율이 변경되더라도 프리셋은 변경되지 않음). 동적 해상도가 지원되지 않는 경우

`NGX_DLSS_GET_OPTIMAL_SETTINGS`를 호출하면 최소 및 최대 렌더 해상도 모두 동일한 값을 반환합니다. 이는 동적 해상도를 지원하지 않는 이전 DLSS 라이브러리에서도 마찬가지입니다.

3.2.2.1 동적 해상도 주의 사항

DLSS와 함께 동적 해상도를 사용할 때는 다음 사항을 준수하는 것이 중요합니다:

1. 섹션 3.5에 설명된 대로 텍스처 밉맵 바이어스를 일관되게 유지합니다. 올바른 디스플레이 해상도 샘플링을 유지하려면 렌더 해상도가 변경될 때마다 밉 레벨을 업데이트해야 합니다(동적 해상도 시스템의 업데이트 속도에 따라 매 프레임마다 업데이트할 수도 있음).

   밉 바이어스를 이렇게 자주 변경할 수 없는 경우 개발자는 추정 바이어스를 사용하거나 제한된 밉맵 바이어스 세트를 사용할 수 있습니다. 화면상의 텍스처 선명도를 유지하려면 약간의 실험이 필요할 수 있습니다.

2. DLSS 이미지 재구성의 정확성을 보장하려면 렌더링 크기의 종횡비가 최종 디스플레이 크기와 일정하게 유지되어야 합니다. 이를 염두에 두고 투영 행렬을 계산할 것을 NVIDIA는 제안합니다.

```
// 디스플레이 해상도와 다른 해상도로 렌더링할 때는 다음을 확인하세요.
// 지오메트리가 전체 창/화면으로 확대되면 비슷하게 보입니다. float
aspectRatio = displaySize.x / displaySize.y;

// 이 종횡비(렌더링 비율이 아닌)를 사용하여 투영 행렬을 계산합니다. float4x4
projection =
    perspProjD3DStyle(dm::radians(m_CameraVerticalFov), aspectRatio, zNear, zFar);
```

3.3 지원되는 형식

DLSS는 형식화된 읽기를 사용하므로 대부분의 입력 버퍼 형식을 처리해야 합니다. 즉, DLSS는 다음과 같은 입력을 기대합니다:

1. 색상 입력 버퍼(메인 프레임): API에 지원되는 모든 버퍼 형식입니다.
2. 모션 벡터: RG32_FLOAT 또는 RG16_FLOAT(자세한 내용은 3.6.1 섹션 참조)
3. 뎁스 버퍼: 하나의 채널이 포함된 모든 포맷(예: R32_FLOAT 또는 D32_FLOAT)과 뎁스 스텐실 포맷(예: D24S8)을 사용할 수 있습니다.
4. 출력 버퍼(처리된 전체 해상도 프레임의 대상): API에 지원되는 모든 버퍼 형식입니다.
5. 이전 출력 버퍼(프레임 히스토리 및 누적에 사용): 선택 사항이지만, 제공될 경우 RGBA16F여야 합니다.

3.4 리소스 상태

DLSS를 호출하는 게임 또는 애플리케이션은 DLSS에 전달된 버퍼가 올바른 사용 플래그로 설정되어 있고

평가 호출 시 올바른 상태인지 확인해야 합니다. DLSS를 사용하려면

1. 입력 버퍼(예: 컬러, 모션 벡터, 뎁스 및 선택적으로 히스토리, 노출 등)는 **읽기 상태**(**셰이더 리소스 뷰**, **HLSL "텍스처"** 또는 Vulkan에서는 **"샘플 이미지"**라고도 함)여야 합니다. 이는 또한 Vulkan의 경우 "**VK_IMAGE_USAGE_SAMPLED_BIT**" 사용 플래그를 사용하여 생성해야 함을 의미합니다. D3D12의 경우 - 컴퓨팅 셰이더가 해당 리소스를 읽게 되므로 상태는 **D3D12_RESOURCE_STATE_NON_PIXEL_SHADER_RESOURCE여야 합니다.**

2. 출력 버퍼는 **UAV 상태여야 합니다(HLSL RWTexture** 또는 Vulkan에서는 "**스토리지 이미지**"라고도 함). 즉, D3D12의 경우 "**D3D12_RESOURCE_FLAG_ALLOW_UNORDERED_ACCESS**" 플래그로, Vulkan의 경우 "**VK_IMAGE_USAGE_STORAGE_BIT**" 사용 플래그로 생성해야 한다는 의미이기도 합니다.

평가 호출 후 DLSS는 버퍼에 액세스하여 처리하고 상태를 변경할 수 있지만 항상 알려진 상태로 버퍼를 다시 전환합니다.

3.5 밉맵 바이어스

DLSS가 활성화되면 렌더링 엔진은 밉맵 바이어스(텍스처 LOD 바이어스라고도 함)를 0보다 낮은 값으로 설정해야 합니다. 이렇게 하면 DLSS에서 사용하는 낮은 렌더링 해상도가 아닌 *디스플레이* 해상도에서 텍스처가 샘플링되므로 전반적인 이미지 품질이 향상됩니다. NVIDIA는 사용을 권장합니다:

```
DlssMipLevelBias = NativeBias + log2(렌더 X해상도 / 디스플레이 X해상도) - 1.0 + 엡실론
```

예를 들어 DLSS 성능 모드에 최적 설정을 사용하는 경우 렌더링 X 해상도/디스플레이 X 해상도 = 0.5 비율입니다. 결과적으로 권장되는 멀티플 레벨 바이어스는 - 입니다.
2.0에 대한 성능 모드를 지원합니다.

- DLSS의 시간적 특성으로 인해 권장되는 음의 LOD 바이어스를 입력에 적용하면 깜박임 및/또는 모아레의 형태로 시간적 안정성이 향상될 수 있습니다. 이는 텍스처 디테일과 같은 씬 콘텐츠에 따라 크게 달라집니다. 깜박임이 산만해질 수 있는 장면과 오브젝트의 경우 LOD 바이어스를 덜 공격적으로 조정하는 것이 좋습니다. 일반적으로 문제가 있는 콘텐츠에서 로그2 (렌더링 해상도/디스플레이 해상도)를 넘지 않는 선에서 조정해 볼 수 있습니다. 권장 값보다 낮으면 더 부드러워지거나 흐릿해지는 경향이 있으므로 디테일 보존과 에일리어싱 사이의 절충점을 찾아야 합니다.

그림 1. 위의 예시에서 텍스처 LOD 바이어스가 -2.0 인 DLSS 가 디테일 보존 측면에서 네이티브 렌더링에 가장 가깝다는 것을 알 수 있습니다.

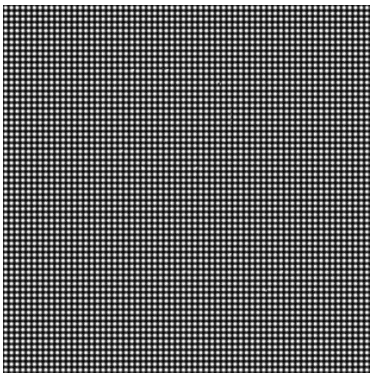
참고: DLSS가 활성화된 경우 텍스처 선명도를 주의 깊게 확인하고 기본 AA 방식으로 기본 해상도에서 렌더링할 때의 텍스처 선명도와 일치하는지 확인하세요. 텍스트나 기타 미세한 디테일(예: 벽의 포스터, 번호판, 신문 등)이 있는 텍스처에 주의를 기울이세요.

기본 해상도 렌더링 중에 네거티브 바이어스가 적용된 경우 일부 아트 에셋이 기본 바이어스에 맞게 조정되었을 수 있습니다. DLSS를 활성화하면 바이어스가 기본값에 비해 너무 크거나 너무 작아 이미지 품질이 저하될 수 있습니다. 이러한 경우 DLSS 밉 레벨 바이어스 계산의 "엡실론"을 조정하세요.

참고: 일부 렌더링 엔진에는 밉맵 바이어스에 대한 전역 클램프가 있습니다. 이러한 클램프가 있는 경우 DLSS를 활성화할 때 비활성화하세요.

3.5.1 밉맵 바이어스 주의: 고빈도 텍스처

밉 레벨이 고주파 패턴이 있는 텍스처에 편향된 경우 DLSS가 전체 해상도 프레임을 재구성하려고 할 때 아티팩트가 발생할 수 있습니다. 특히 아래 그림과 같은 텍스처를 사용하는 'LED 화면', '주식 시세표' 또는 이와 유사한 것을 시뮬레이션하려는 경우 해당 머티리얼의 밉맵 바이어스를 오버라이드하고 기본값으로 남겨두세요.



고주파 "LED 스크린" 텍스처 예시

3.6 모션 벡터

DLSS는 핵심 알고리즘의 핵심 구성 요소로 픽셀별 모션 벡터를 사용합니다. 모션 벡터는 현재 프레임의 픽셀을 이전 프레임의 해당 위치에 매핑합니다. 즉, 픽셀의 모션 벡터를 픽셀의 현재 위치에 더하면 그 결과는 픽셀이 이전 프레임에서 차지했던 위치가 됩니다.

중요: 부정확하거나 정밀도가 낮은 모션 벡터는 DLSS가 활성화된 경우 시각적 부작용의 가장 일반적인 원

인입니다. DLSS에서 시각적 아티팩트가 발견되면 언제든지 시각화 도구(예: 디버그 오버레이 - 섹션 8.2 참조)를 사용하여 모션 벡터를 확인하시기 바랍니다.

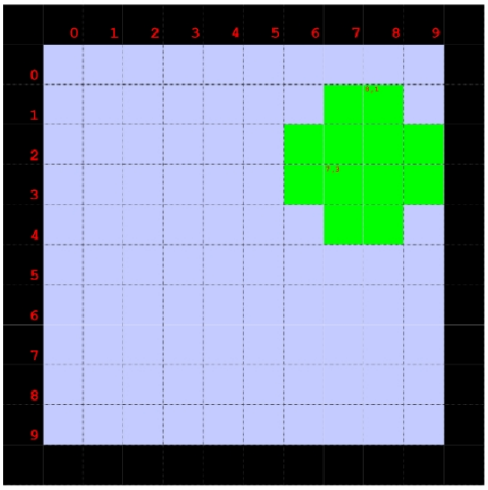
3.6.1 모션 벡터 포맷 및 계산

모션 벡터는 렌더링 타깃이 RG32_FLOAT 또는 RG16_FLOAT 형식의 플로트로 전송되어야 합니다. 2D 화면 공간 모션 벡터의 X 및 Y 값은 텍스처의 빨간색 및 녹색 채널에 32비트 또는 16비트 부동 소수점 형식(형식에 따라 다름)으로 저장됩니다.

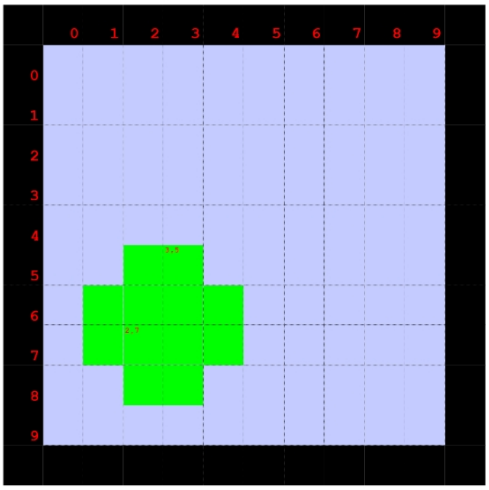
각 모션 벡터의 값은 화면 공간에서 계산된 픽셀 수(즉, **렌더링 해상도에서** 픽셀이 이동한 양)로 주어진 움직임의 양을 나타내며, 가정합니다:

1. 화면 공간 픽셀 값은 화면의 **왼쪽 상단을** [0,0]으로 사용하고 **렌더링** 대상의 전체 해상도까지 확장합니다. 예를 들어 렌더링 대상이 1080p 표면인 경우 오른쪽 하단의 픽셀은 [1919,1079]입니다.
    a. DLSS는 디스플레이 해상도에서 계산되는 전체 해상도 모션 벡터를 선택적으로 허용할 수도 있습니다. 자세한 내용은 섹션 3.6.2를 참조하세요.
2. 모션 벡터는 씬 오브젝트, 카메라 및 화면의 움직임에 따라 포지티브 또는 네거티브가 될 수 있습니다.
3. 모션 벡터에는 전체 및 부분 픽셀 이동이 포함될 수 있습니다(예: [3.0f,-1.0f] 및 [-0.65f,10.1f]가 모두 유효함).

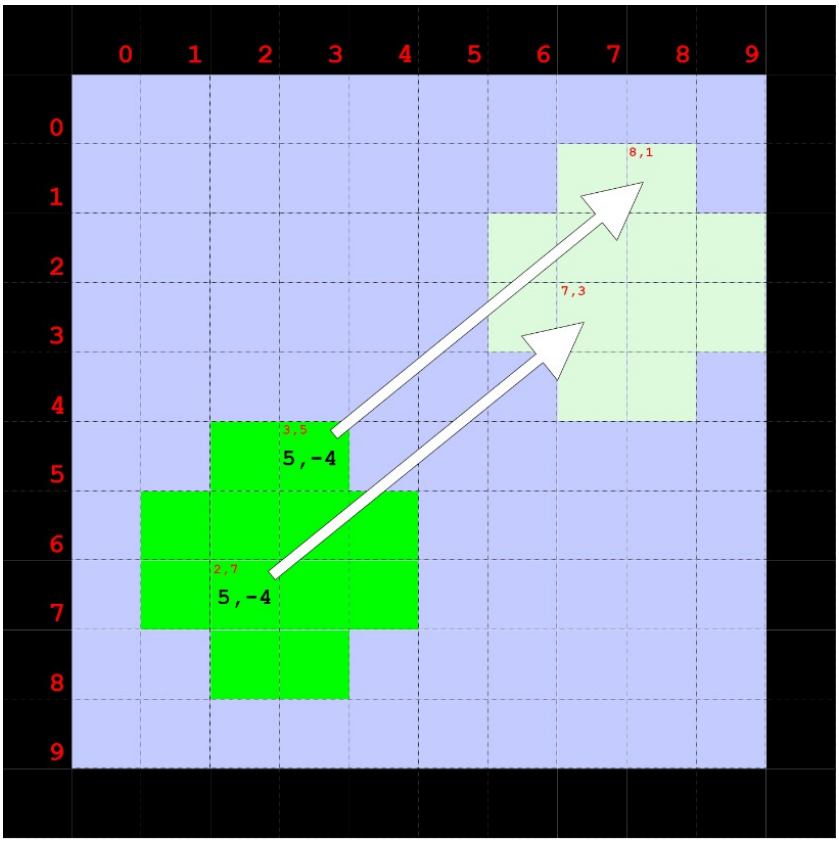
참고: 게임 또는 렌더링 엔진이 모션 벡터에 커스텀 포맷을 사용하는 경우, 이를 디코딩해야 합니다. **를 호출하기 전에**



프레임 "N-1"



프레임 "N"



프레임 "N"의 모션 벡터 값 예시

3.6.1.1 고밀도 모션 벡터 리졸브 셰이더

언리얼 엔진 4 또는 지오메트리 이동을 사용하여 모션 벡터를 계산하는 다른 엔진을 사용하는 게임의 경우 DLSS는 모션 벡터 계산 방식에 약간의 조정이 필요합니다. 기본 모션 벡터 버퍼에 아래와 같은 픽셀 셰이더를 적용합니다.

참고: DLSS가 통합된 NVIDIA의 UE4 커스텀 브랜치를 사용하거나 병합한 경우, 이 변경 사항(또는 매우 유사한 사항)이 이미 적용되었습니다.

```
텍스처2D 뎁스 텍스처; 텍스처2D
속도 텍스처;

float2 UVToClip(float2 UV)
{
  float2(UV.x * 2 - 1, 1 - UV.y * 2)를 반환합니다;
}

float2 ClipToUV(float2 ClipPos)
{
  float2(ClipPos.x * 0.5 + 0.5, 0.5 - ClipPos.y * 0.5)를 반환합니다;
}
```

```
float3 균질에서 유클리드까지(float4 V)
{
  V.xyz/V.w를 반환합니다;
}

void VelocityResolvePixelShader(
  float2 InUV : TEXCOORD0,
  float4 SvPosition : SV_Position,
  out float4 OutColor : SV_Target0
)
{
  OutColor = 0;

  float2 Velocity = VelocityTexture[SvPosition.xy].xy;
  float Depth = DepthTexture[SvPosition.xy].x;

  if (all(Velocity.xy > 0))
  {
    속도 = 텍스처에서 속도 디코딩(속도);
  }
  else
  {
    float4 ClipPos;
    ClipPos.xy = SvPositionToScreenPosition(float4(SvPosition.xyz, 1)).xy;
    ClipPos.z = Depth;
    ClipPos.w = 1;

    float4 PrevClipPos = mul(ClipPos, View.ClipToPrevClip);

    if (PrevClipPos.w > 0)
    {
      float2 PrevClip = 균질에서 유클리드까지(PrevClipPos).xy; 속
      도 = ClipPos.xy - PrevClip.xy;
    }
  }

  OutColor.xy = Velocity * float2(0.5, -0.5) * View.ViewSizeAndInvSize.xy;
  OutColor.xy = -OutColor.xy;
}
```

3.6.2 모션 벡터 플래그

보다 다양한 엔진에 쉽게 통합할 수 있도록 DLSS는 여러 가지 방법으로 모션 벡터를 허용합니다. 렌더링 엔진 요구 사항에 따라 이들 사이를 전환하려면 DLSS 기능 생성 중에 플래그를 적절히 설정하세요(5.3절 참조).

1. `NVSDK_NGX_DLSS_Feature_Flags_MVLowRes`: 모션 벡터는 일반적으로 입력 컬러 프레임과 동일한 해상도(즉, 렌더링 해상도)에서 계산됩니다. 렌더링 엔진이 디스플레이/출력 해상도에서 모션 벡터를 **계산하고** 모션 벡터를 확대하는 것을 지원하는 경우, DLSS는 이 플래그를 "0"으로 설정하여 이를 허용할 수 있습니다. 이 방법은 흔하지는 않지만 선호되며, 움직이는 물체의 앤티앨리어싱 품질이 향상되고 작은 물체와 얇은 디테일의 블러링이 줄어듭니다.

명확히 하기 위해 표준 입력 해상도 모션 벡터가 전송되면 확대할 필요가 없으며, DLSS가 내부적으로 확대합니다. 디스플레이 해상도 모션 벡터가 전송되는 경우 반드시 확대해야 합니다.

2. `NVSDK_NGX_DLSS_Feature_Flags_MVJittered`: 모션 벡터에 서브 픽셀 지터가 포함된 **경우** 이 플래그를 "1"로 설정합니다. 그러면 DLSS는 "평가" 호출 중에 제공된 지터 오프셋 값을 사용하여 내부적으로 모션 벡터에서 지터를 뺍니다. "0"으로 설정하면 DLSS는 별도의 조정 없이 모션 벡터를 직접 사용합니다.

3.6.3 모션 벡터 스케일

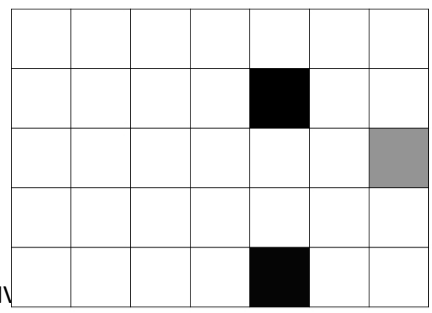
엔진에 DLSS에 필요한 스케일이나 방향과 일치하지 않는 기존 모션 벡터가 있는 경우가 있습니다. 한 가지 예로 모션 벡터가 이전 프레임이 아닌 모션 방향을 가리키는 경우입니다. 또 다른 예는 모션 벡터 값이 픽셀 공간이 아닌 UV 공간에 있는 경우입니다.

이 데이터를 직접 사용할 수 있도록 DLSS는 평가 중에 사용되는 모션 벡터 스케일 파라미터를 제공하여 DLSS로 전달될 때 모션 벡터 값을 일부 수정할 수 있도록 합니다. 이 파라미터는 일반적으로 평가 파라미터 구조체 `NVSDK_NGX_D3D11_DLSS_Eval_Params`에서 `InMVScaleX` 및 `InMVScaleY` 멤버를 통해 설정됩니다. 모션 벡터의 스케일을 조정할 필요가 없는 경우 1.0으로 설정해야 하며 0.0f가 되어서는 안 됩니다. (섹션 5.4 또는 *nvsdk_ngx_helpers.h* 참조).

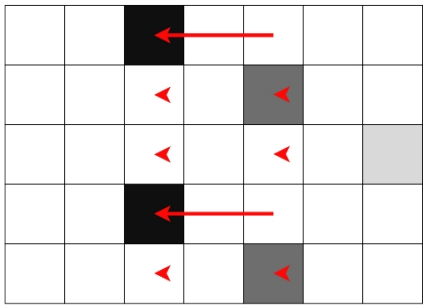
3.6.4 얇은 피처를 위한 모션 벡터의 보수적인 래스터

얇은 기하학적 특징을 가진 개체를 렌더링하는 경우 입력 버퍼의 해상도가 낮고 래스터라이저가 일부 기하학적 특징을 "누락"하기 때문에 시야에 들어오고 나가는 경향이 있습니다. 이로 인해 해당 오브젝트가 렌더링될 때 눈에 보이는 구멍이 생깁니다.

DLSS는 때때로 누락된 지오메트리를 재구성하는 데 도움이 될 수 있지만, 이를 위해서는 정확한 모션 벡터가 필요합니다. 이러한 특징이 보이지 않기 때문에 DLSS로 전송되는 입력에 누락된 특징의 색상이나 모션 벡터가 존재하지 않습니다. 아래 표시된 시나리오에서 DLSS는 이전에 그려진 해당 물체의 이미지를 배경의 모션 벡터와 잘못 연결합니다. 이 시나리오에서는 객체 드로잉의 구멍을 수정하는 대신 해당 구멍이 계속 유지됩니다.



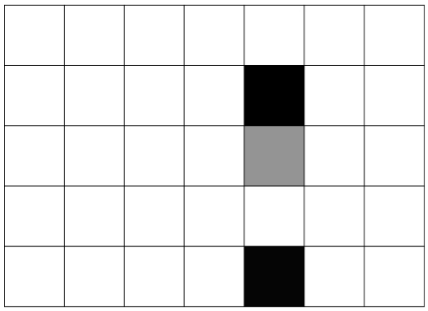
Frame 0

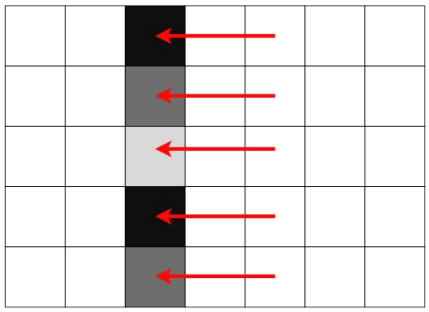


Frame 1

이 특정 문제에 대한 잠재적인 해결 방법은 보수 래스터라는 기능(예: Direct3D 11.3 및 Direct3D 12에 도입됨)을 사용하여 해당 오브젝트에 대한 모션 벡터를 그리는 것입니다(이 기능만!). 보수적 래스터화는 프리미티브가 픽셀에 아주 조금이라도 닿으면 그 픽셀이 그려지도록 합니다. 따라서 색상 정보가 오브젝트가 존재한다는 것을 포착하지 못하면 보수 래스터로 그려진 모션 벡터가 해당 정보를 포착합니다.

이를 사용하면 이전에 그려진 오브젝트 부분의 움직임이 화면에서 오브젝트가 움직인 정확한 양을 표시하고 DLSS가 재구성할 때 더 나은 기회를 가질 수 있습니다.





다음은 게임 내 비교입니다. 왼쪽은 모션 벡터에 보수적 래스터를 사용하고 있으며, 그 결과 보수적 래스터를 사용하지 않은 오른쪽 이미지보다 재구성 시 눈에 보이는 구멍이 약간 적다는 것을 알 수 있습니다.

DLSS는 입력 데이터의 많은 힌트를 사용하여 이미지를 더 잘 재구성하고 때로는 잘못 판단하여 재구성된 기록을 삭제하는 것이 최선이라고 생각할 수 있으므로 이 방법이 깜박이는 개체를 수정하는 데 반드시 만병통치약은 아니라는 점에 유의하세요.

3.7 서브 픽셀 지터

DLSS에는 앤티 앨리어싱, 기능 향상 및 업스케일링 알고리즘에 여러 시간적 구성 요소가 포함되어 있습니다. 높은 이미지 품질을 얻기 위해 DLSS는 렌더링 엔진이 매 프레임마다 원하는 샘플 위치의 하위 집합을 생성하고 이를 시간적으로 통합하여 최종 이미지를 생성함으로써 더 높은 샘플 속도를 에뮬레이션합니다.

렌더러는 뷰포트 또는 메인 카메라 뷰에 서브 픽셀 지터를 적용하여 추가 샘플을 제공하여 시간에 따라 래스터화된 프레임을 변경합니다.

다시 말해, 움직임이 없는 렌더링 및 래스터화된 픽셀이 4프레임 이상인 경우 DLSS가 생성하는 이미지는 4배 슈퍼 샘플링된 이미지와 동일해야 합니다. 이것이 DLSS의 목표이지만 실제로 항상 달성할 수 있는 것은 아닙니다.

이 섹션에서는 설명합니다:

1. 각 프레임에 사용할 샘플 위치를 선택하는 방법.
2. 다양한 샘플 오프셋을 사용하여 렌더링하는 방법.
3. 지터 정보를 DLSS로 전송하는 방법.

3.7.1 지터 샘플 패턴

3D 장면에 지터를 적용하는 데 사용할 수 있는 다양한 패턴이 있습니다. 최상의 결과를 얻으려면 지터 패턴이 전체 픽셀에 걸쳐 잘 적용되어야 합니다. NVIDIA는 지터 패턴에 할튼 시퀀스 (https://en.wikipedia.org/wiki/Halton_sequence)를 사용하는 것이 최상의 결과를 제공한다는 사실을 발견했습니다. 따라서 DLSS를 위해 수행되는 딥 러닝 훈련은 훈련 데이터에 Halton을 사용합니다.

가능하면 TAA 또는 대체 AA 모드가 활성화되어 있을 때 다른 패턴이 사용되더라도 DLSS가 활성화된 경우 하위 픽셀 지터에 할튼 시퀀스를 사용하십시오. 다른 패턴을 사용하는 경우에도 이 섹션의 나머지 지터 요구 사항을 준수하는 경우 DLSS는 여전히 올바르게 작동해야 하지만 NVIDIA는 다른 패턴을 테스트하지 않았습니다.

3.7.1.1 패턴 단계

시퀀스 유형 외에도 시간이 지남에 따라 전체 픽셀 영역을 잘 커버할 수 있도록 패턴을 효과적으로 순환하는 것이 중요합니다.

단계 수(즉, 반복하기 전 패턴의 고유 샘플 수)에 대한 좋은 선택은 다음과 같습니다:

```
총 페이즈 수 = 기본 페이즈 수 * (타깃 해상도/렌더 해상도) ^ 2
```

기본 페이즈 수는 일반 템포럴 안티앨리어싱(예: TXAA 또는 TAA)에 자주 사용되는 페이즈 수입니다. 베이스의 좋은 시작 값은 "8"이며, 스케일링을 적용하지 않았을 때 좋은 픽셀 커버리지를 제공합니다. 그런 다음 DLSS의 경우 베이스는 픽셀 영역의 스케일링 비율에 따라 스케일링됩니다.

예를 들어 렌더링 타겟이 1080p이고 타겟 출력이 2160p(일명 4k)인 경우 1080p 픽셀은 4K 픽셀의 4배 크

기입니다. 따라서 각 4K 픽셀을 8개의 샘플로 커버하려면 4배의 페이즈가 필요합니다. 따라서 이 예시에서는 총 위상 수가 32개가 됩니다:

```
총 위상 = 8 * (2160 / 1080) ^ 2
```

3.7.2 지터 오프셋을 사용한 렌더링

사용하는 렌더링 엔진에 따라 실제로 지터링된 프레임을 생성하는 방법은 다를 수 있습니다. 렌더러에 이미 TAA 또는 TXAA가 포함되어 있는 경우 해당 렌더러가 사용 중일 때 지터가 어떻게 적용되는지 살펴보고 DLSS가 활성화된 경우에도 동일한 작업을 수행하세요.

일반적으로 지터를 사용하여 렌더링하려면 3D 장면이 래스터화될 때 결과 프레임(특히 가장자리)에 약간의 변화가 생기도록 카메라나 뷰포트에 사람이 감지할 수 없는 움직임을 적용합니다. 지터 오프셋을 적용하는 일반적인 절차는 렌더러의 투영 매트릭스를 수정하는 것입니다:

```
// 각 프레임마다, 지터 시퀀스의 값으로 ProjectionJitter 값을 업데이트합니다.
// 현재 위상에 대해. 그런 다음 프레임의 주 투영 행렬을 오프셋합니다.
ProjectionMatrix.M[2][0] += ProjectionJitter.X;
ProjectionMatrix.M[2][1] += ProjectionJitter.Y;
```

이 방법을 사용하면 월드-공간 좌표를 이동한 다음 투영하는 것이 아니라 카메라-공간 좌표를 이동해야 합니다.

3.7.3 필수 지터 정보

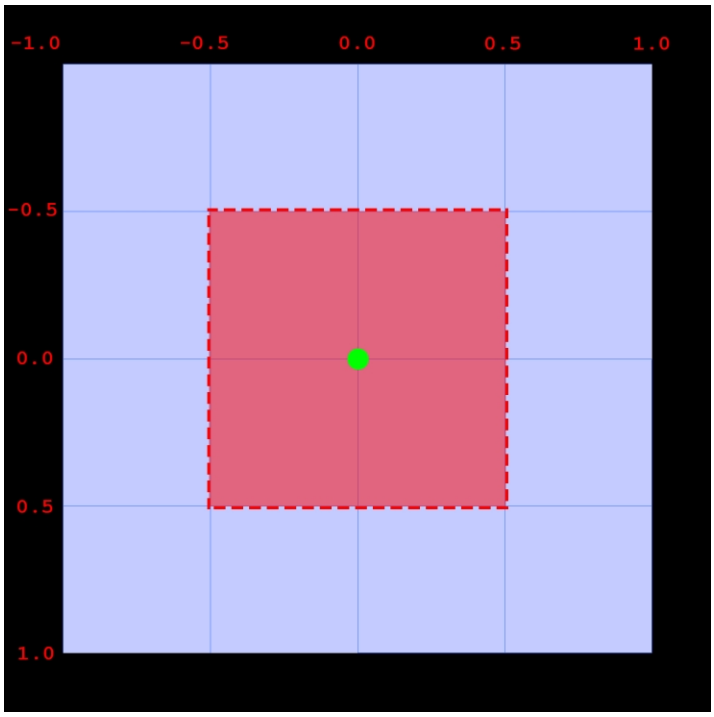
현재 프레임을 과거 프레임과 올바르게 통합하려면 DLSS가 이 프레임의 지터가 어떻게 적용되었는지 알아야 합니다. DLSS 평가를 호출할 때마다 현재 프레임의 지터 양을 파라미터 목록의 지터 오프셋으로 설정합니다(사용 중인 그래픽 API에 맞는 올바른 유형 구조 사용):

```
                                    유형 정의 구조체
                NVSDK_NGX_D3D11_DLSS_Eval_Params
{
        NVSDK_NGX_D3D11_Feature_Eval_Params 기능;
    /* 픽셀 공간의 지터 오프셋 */
    float                           InJitterOffsetX;
    float                           InJitterOffsetY;
}
```

지터 오프셋 값입니다:

1. 항상 -0.5에서 +0.5 사이여야 합니다(지터는 항상 다음과 같은 움직임을 초래해야 하므로 소스 픽셀 **내**).
2. 렌더링 타깃 크기의 픽셀 공간으로 제공됩니다(출력 또는 타깃 크기가 아닌 지터). 모션 벡터에서 선택적으로 가능한 출력 해상도로 제공될 수 있습니다).
3. 오프셋이 모션 벡터에도 적용되었는지 여부에 관계없이 뷰포트 지터에 대한 투영 매트릭스에 적용된 지터를 나타냅니다.
    a. 모션 벡터에 지터가 적용된 경우, 섹션 3.6.2에 설명된 대로 DLSS 생성 호출 중에 적절한 플래그를 설정해야 합니다.
4. 모션 벡터와 동일한 좌표 및 방향 시스템(위의 3.6.1 참조)을 다음과 같이 사용합니다.

"InJitterOffsetX == 0" 및 "InJitterOffsetY == 0"은 지터가 적용되지 않았음을 의미합니다.



지터 오프셋 스케일 다이어그램. 음영 처리된 빨간색 영역은 ***전체 픽셀 1 개입니다****.*
녹색 표시는 원점이며 빨간색은 예상 값의 범위를 나타냅니다.

3.7.4 지터 문제 해결

화면이 흔들리거나 멀리 있는 물체가 해결되지 않거나 출력에 '스크린 도어'가 나타나거나 정적인 물체(특히 얇은 물체와 미세한 텍스처 디테일)가 '흐릿하게' 보이는 등의 문제가 있는 경우 문제가 있는 것일 수 있습니다:

1. 렌더러에서 지터가 적용되는 방식입니다.
2. 제공된 모션 벡터.
3. DLSS로 전송된 지터 오프셋 값입니다.

디버깅을 지원하기 위해 NVIDIA는 DLSS 라이브러리의 SDK 버전에 여러 도구와 비주얼라이저를 추가했습니다. 이러한 도구를 사용하는 방법과 지터 관련 문제를 디버깅하는 방법에 대한 자세한 내용은 섹션 8.3을 참조하십시오.

3.8 뎁스 버퍼

더 나은 오브젝트 추적과 픽셀 정렬을 지원하기 위해 DLSS는 래스터화 중에 엔진에서 생성한 깊이 버퍼를 사용합니다. 이 알고리즘은 근거리 평면이 0.0이고 원거리 평면이 1.0이라고 가정합니다(아래에서 반전될 수 있음).

이를 필요로 하는 포스트 프로세싱 셰이더에는 가장 가까운 업샘플링을 사용하는 것이 좋습니다.

3.8.1 뎁스 버퍼 플래그

보다 다양한 엔진에 쉽게 통합할 수 있도록 DLSS는 다양한 깊이 구성을 허용합니다. 렌더러에 따라 DLSS 기능 생성 중에 플래그를 적절히 설정하세요(5.3절 참조).

1. `NVSDK_NGX_DLSS_Feature_Flags_DepthInverted`: 엔진이 근거리 평면을 1.0, 원거리 평면을 0.0으로 하여 깊이를 사용하는 경우 이 플래그를 "1"로 설정합니다.

3.9 노출 파라미터

HDR로 처리할 때 DLSS는 현재 프레임에 대한 렌더러의 노출 값이 필요합니다. 이 값은 입력 색상 값에 곱하면 중간 회색을 예상 수준으로 가져오는 값입니다. 이 값은 일반적으로 렌더러의 톤매퍼에 제공되는 값과 동일하여 HDR을 LDR 색상으로 압축할 때(예: 표준 LDR 모니터로 출력할 때) 일관되게 처리합니다.

참고: 기본 노출 값은 다음 함수를 통해 계산할 수 있습니다(여기서 `MidGray`는 전체 반사 밝기와 전체 흡수 밝기 사이의 지각적 중간을 나타내는 반사광의 양으로, `MidGray`의 경우 "0.18"이 일반적이며, AverageLuma는 전체 프레임의 평균 휘도를 나타냅니다):

```
노출 값 = 중간 회색 / (평균 루마 * (1.0 - 중간 회색))
```

렌더러는 각 DLSS 평가 호출 시 `pInExposureTexture` 파라미터에서 참조된 1x1 텍스처를 사용하여 올바른 `ExposureValue`를 DLSS에 제공해야 합니다. 텍스처에서 첫 번째 채널만 샘플링되므로 여러 형식이 작동하지만 `R16F`와 같은 형식이 선호됩니다.

참고: 이 값은 일반적으로 GPU에서 프레임 단위로 계산되므로 CPU로 왕복하는 것을 피하기 위해 1x1 텍스처로 DLSS에 전송됩니다.

노출값이 누락되거나 DLSS가 올바른 값을 받지 못하면(일부 게임 엔진에서는 눈의 적응에 따라 달라질 수 있음), DLSS가 고해상도 프레임을 잘못 재구성하여 다음과 같은 아티팩트를 생성할 수 있습니다:

1. 움직이는 물체의 고스트.
2. 최종 프레임의 흐릿함, 밴딩 또는 픽셀화 또는 너무 어둡거나 너무 밝은 경우.
3. 특히 움직이는 오브젝트의 앨리어싱.
4. 노출 지연.

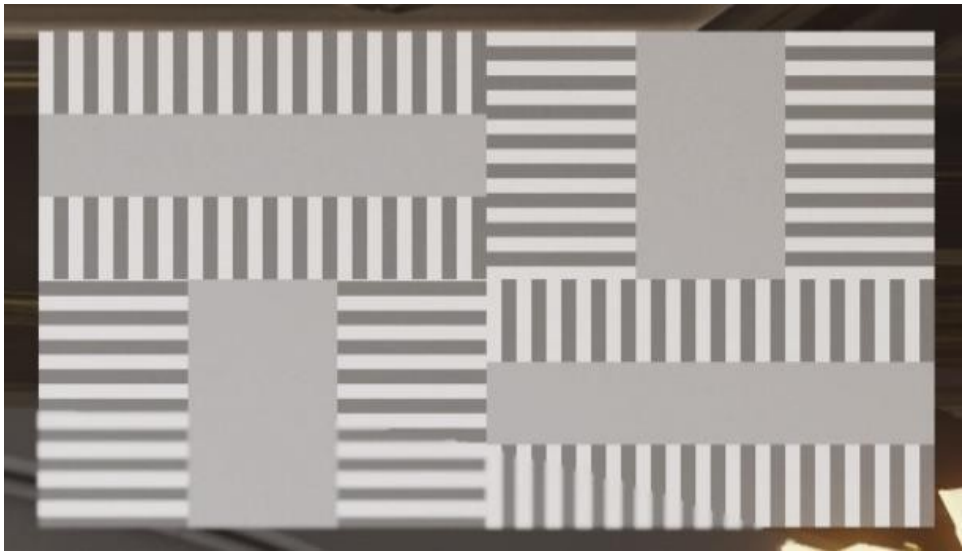
노출값을 제공하더라도 위의 문제 중 하나 또는 그 이상이 발생하면 스케일 계수 및/또는 편향이 필요할 수 있습니다.

3.9.1 비주얼라이저로 불량 노출 진단하기

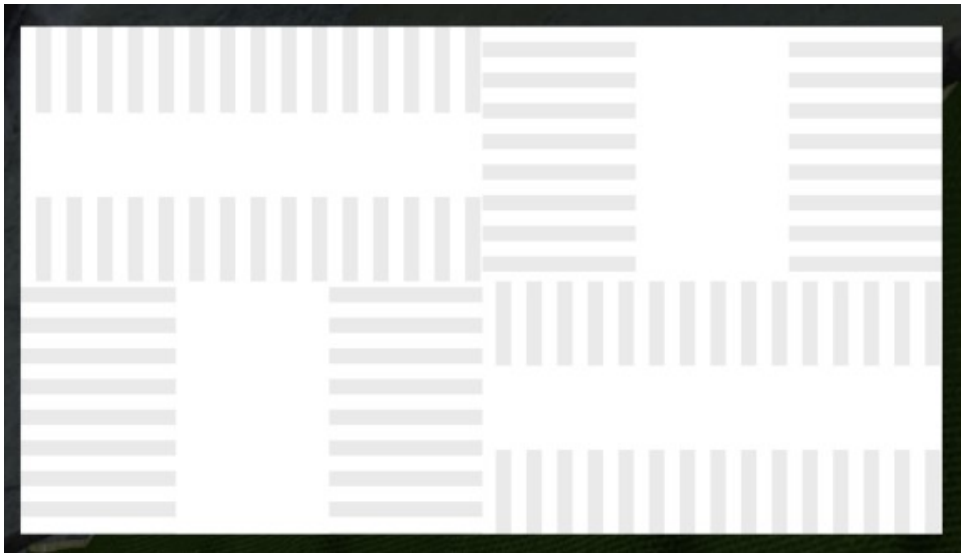
최근 SDK에서는 노출 규모 문제를 진단하는 데 도움이 되는 시각적 보조 기능을 추가했습니다.

디버그 비주얼라이저를 순환하여 오른쪽 상단 모서리에 해당 패턴을 표시할 수 있습니다(8.2절 참조).

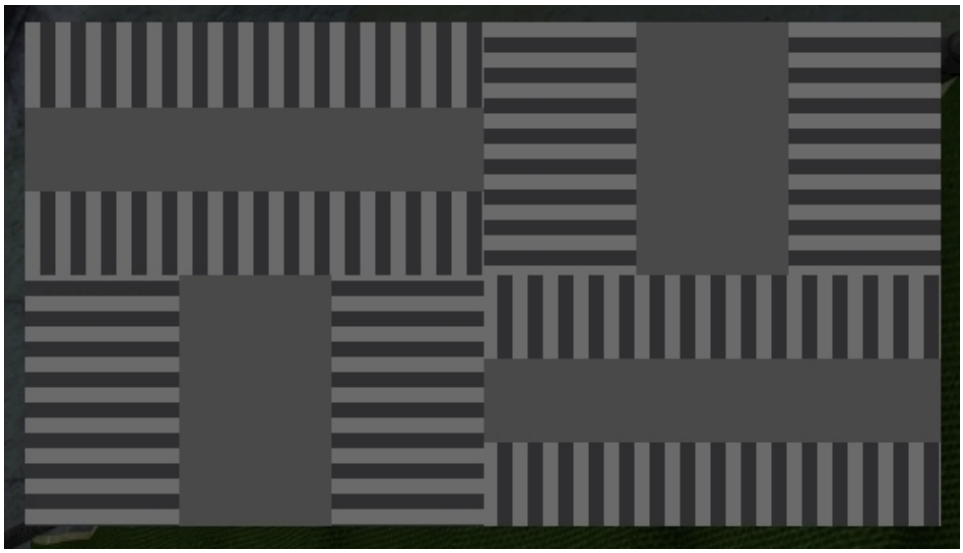
DLSS에 전달되는 노출 스케일 값(및 사전 노출)이 정확하다면 아래와 같은 패턴을 볼 수 있습니다. 이 패턴은 밝기 값 187(sRGB 8비트 공간에서 0과 255의 스케일 기준)을 중심으로 합니다.



반면에 노출 스케일 텍스처를 통과하는 값이 저평가되어 DLSS에서 노출이 크게 부족한 이미지가 표시되는 경우 모든 것이 매우 밝은 아래 패턴을 볼 수 있습니다:



반대쪽에서 노출 스케일에 값을 과대 평가하여 전달하면 DLSS가 노출이 과다한 이미지를 보게 되며, 이 경우 모든 것이 매우 어두운 아래 패턴이 다시 나타납니다:



3.9.2 사전 노출 계수

일부 엔진은 톤 매핑 중에 나중에 제거(분할)되는 "사전 노출 계수"로 프레임을 미리 곱하는 방식으로 추가 노출 튜닝을 구현합니다. DLSS 알고리즘은 이전 프레임 기록을 많이 사용하기 때문에 이 요소를 고려하지 않으면 입력이 사실상 '이중 노출'이 되어 고해상도 프레임의 재구성이 제대로 이루어지지 않을 수 있습니다. 시각적 아티팩트는 잘못된 `pInExposureTexture` 텍스처가 사용된 경우와 동일합니다(섹션 3.9 참조).

이는 일반적인 사용 사례가 아니므로 개발자는 잘못된 서브 픽셀 지터 또는 잘못된 모션 벡터와 같은 문제로 인해 시각적 아티팩트가 발생하지 않는지 확인하는 것이 좋습니다. 먼저 이를 확인하고 사전 노출 값이 사용되었는지 또는 DLSS에 전달된 입력 버퍼가 이미 '눈 적응'되어 있는지 핵심 렌더링 엔진 리드에게 확인하시기 바랍니다.

사전 노출 값이 있는 경우 모든 DLSS 평가 호출 중에 `InPreExposure` 매개변수(DLSS v2.1.0부터 사용 가능).

참고: 사전 노출 값은 선택 사항이며 파라미터 맵을 통해 전달되는 CPU 측 플로트로 DLSS에 전송됩니다. 기본 노출 값과 같은 텍스처가 아닙니다.

3.10 자동 노출

기본 노출 파라미터(위 참조)를 사용하는 것이 선호되는 방법이지만, 경우에 따라 DLSS 라이브러리의 옵션

자동 노출 기능을 활용하면 더 나은 결과를 얻을 수 있습니다. `pInExposureTexture` 대신 DLSS 라이브러리 내에서 계산된 자동 노출 값을 사용하려면 개발자가 직접 계산해야 합니다:

1. DLSS 피처 생성 시 `NVSDK_NGX_DLSS_Feature_Flags_AutoExposure` 파라미터를 설정합니다.

참고: 자동 노출을 활성화하면 DLSS 알고리즘 버전, GPU 및 출력 해상도에 따라 처리 시간이 약간 증가(~0.02ms)할 수 있습니다.

자동 노출은 게임 내에서 CTRL+ALT+Y 단축키를 눌러 DLSS SDK DLL을 사용하여 토글할 수 있습니다(생성 시 전달된 플래그 재정의).

3.11 추가 선명도 향상

DLSS에 사용되는 딥 러닝 모델은 선명한 이미지를 생성하도록 학습되지만, 더 선명한 이미지를 제공하고자 하는 개발자를 위해 NVIDIA 이미지 스케일링 SDK는 이제 DLSS SDK의 일부로 공개적으로 제공되며 Github의 NVIDIA 이미지 스케일링 저장소에서 사용할 수 있습니다. 샤프닝 필터를 사용하는 개발자는 샤프닝 값을 제어할 수 있는 최종 사용자 슬라이더를 제공할 것을 적극 권장합니다. (자세한 내용은 이미지 스케일링 설명서를 참조하세요.)

참고: 이전 버전의 SDK에서 제공된 샤프닝 및 소프트닝 패스는 이제 더 이상 사용되지 않습니다. 모든 개발자는 DLSS SDK를 통해 또는 GitHub에서 직접 이미지 스케일링 SDK를 사용할 것을 권장합니다.

3.12 DLSS 사전 설정

기본 설정은 여전히 최고의 화질을 제공하지만, 애플리케이션에 맞는 최고의 화질을 원하는 경우 다양한 DLSS 사전 설정을 활용할 수 있습니다. 사전 설정은 다양한 스케일링 비율, 게임 콘텐츠 등을 충족하기 위해 DLSS의 다양한 측면을 조정합니다. DLSS 프리셋은 스케일링 비율에 따라 활성화됩니다. 예를 들어 퍼포먼스 모드에서 프리셋 A를 활성화합니다:

```
자동 프리셋 = NVSDK_NGX_DLSS_Hint_Render_Preset::NVSDK_NGX_DLSS_Hint_Render_Preset_A;
NVSDK_NGX_Parameter_SetUI(ngxParams, NVSDK_NGX_DLSS_Hint_Render_Preset_성능, 프리셋);
```

다음은 NVSDK_NGX_Parameter_SetUI API를 사용하여 퍼포먼스 모드별로 설정할 수 있는 프리셋입니다:

```
// NVSDK_NGX_Parameter_SetUI를 통해 설정할 수 있으며 0 - 4 값(포함)을 취할 수 있습니다 유형 정의 enum NVSDK_NGX_DLSS_Hint_Render_Preset
{
    NVSDK_NGX_DLSS_Hint_Render_Preset_Default, // 기본 동작, 가중치는 OTA 후 업데이트될 수도 있고 업데이트되지 않을 수도 있습니다.
    NVSDK_NGX_DLSS_Hint_Render_Preset_A,
    NVSDK_NGX_DLSS_Hint_Render_Preset_B,
    NVSDK_NGX_DLSS_Hint_Render_Preset_C,
    NVSDK_NGX_DLSS_Hint_Render_Preset_D,
    NVSDK_NGX_DLSS_Hint_Render_Preset_E,
    NVSDK_NGX_DLSS_Hint_Render_Preset_F,
    NVSDK_NGX_DLSS_Hint_Render_Preset_G,

} NVSDK_NGX_DLSS_Hint_Render_Preset;

#define NVSDK_NGX_Parameter_DLSS_Hint_Render_Preset_DLAA "DLSS.Hint.Render.Preset.DLAA"
#define NVSDK_NGX_Parameter_DLSS_Hint_Render_Preset_Quality "DLSS.Hint.Render.Preset.Quality"
#define NVSDK_NGX_Parameter_DLSS_Hint_Render_Preset_Balanced "DLSS.Hint.Preset.Balanced"
#정의 NVSDK_NGX_Parameter_DLSS_Hint_Render_Preset_Performance
"DLSS.Hint.Render.Preset.Performance"
```

```
#정의 NVSDK_NGX_Parameter_DLSS_Hint_Render_Preset_UltraPerformance
"DLSS.Hint.Render.Preset.UltraPerformance"
```

DLSS 기본값과 프리셋은 새 버전의 DLSS에 따라 변경될 수 있습니다. 입력 및 출력 렌더링 크기가 위에서 설명한 것과 동일한 값으로 설정된 경우에만 NVSDK_NGX_DLSS_Hint_Render_Preset_DLAA가 작동한다는 점에 유의하세요.

다시 말하지만, 사전 설정은 개정될 때마다 변경될 수 있지만 다음은 현재 사전 설정으로 실험할 수 있는 일반적인 가이드 역할을 합니다:

- 프리셋 A(퍼프/밸런스/품질 모드용):
    - 모션 벡터와 같이 입력이 누락된 요소의 고스팅을 방지하는 데 가장 적합한 구형 변형입니다.
- 프리셋 B(울트라 퍼프 모드용):
    - 프리셋 A와 유사하지만 울트라 퍼포먼스 모드의 경우
- 프리셋 C(퍼프/밸런스/품질 모드용):
    - 일반적으로 현재 프레임 정보를 선호하는 프리셋입니다. 일반적으로 빠른 속도의 게임 콘텐츠에 적합합니다.
- 프리셋 D(퍼프/밸런스/품질 모드용):
    - 퍼프/균형/품질 모드의 기본 프리셋입니다. 일반적으로 이미지 안정성을 선호합니다.
- 프리셋 E(미사용)
- 프리셋 F(울트라 퍼프/DLAA 모드용):
    - 울트라 퍼프 및 DLAA 모드의 기본 프리셋입니다.
- 프리셋 G(미사용)

3.12.1 OTA 업데이트를 통한 사전 설정 선택 동작

이 섹션에서는 (2.4절)에서 설명한 OTA 업데이트가 애플리케이션 프리셋 선택에 미치는 영향에 대해 설명합니다.

- OTA 업데이트를 옵트인하고 특정 모델을 선택한 타이틀은 앞으로도 해당 모델을 유지합니다.
- 업데이트가 원래 통합된 기본 프리셋과 다른 기본 프리셋을 제공하는 경우 - OTA 업데이트를 옵트인하고 기본 모델을 사용 중인 타이틀은 OTA 업데이트의 일부로 기본 모델이 변경될 수 있습니다.

3.13 장면 전환

DLSS는 시공간 재구성 기술을 기반으로 하며 이전 프레임에서 수집한 시간 정보를 활용합니다. 프레임마다 장면에 완전한(또는 상당한) 변화가 있는 경우 알고리즘이 혼동될 수 있습니다. 이로 인해 지연된 전환이나 이전 장면의 '고스트 이미지'가 새 장면으로 넘어가는 등의 시각적 아티팩트가 발생할 수 있습니다.

이러한 아티팩트를 방지하려면 주요 전환 후 첫 번째 프레임에 대한 DLSS 평가 호출 중에 `DLSS_Eval` 파라미터 목록에서 `InReset` 파라미터를 0이 아닌 값으로 설정하세요. 이렇게 하면 DLSS가 내부 템포럴 컴포넌트를 무효화하고 들어오는 입력으로 처리를 다시 시작하도록 지시합니다.

참고: 이 기능을 잘못 사용하면 일시적인 깜박임, 심한 에일리어싱 또는 기타 시각적 아티팩트가 발생할 수 있습니다.

3.14 VRAM 사용량

최신 렌더링 엔진은 종종 VRAM 사용량을 추적하고 점유율에 따라 렌더링 파라미터와 아트 에셋을 수정합니다. 이러한 추적을 지원하기 위해 DLSS는 현재 DLSS에서 내부적으로 할당된 VRAM의 양을 쿼리하는 데 사용할 수 있는 NVSDK_NGX_DLSS_GetStatsCallback() 함수를 제공합니다. 이 함수를 사용하는 가장 편리한 방법은 NGX_DLSS_GET_STATS() 인라인 헬퍼를 사용하는 것입니다.

함수가 반환하는 메모리 양은 전역 변수라는 점에 **유의하세요.** 따라서 여러 DLSS 인스턴스를 생성한 경우 이 함수는 모든 인스턴스에서 현재 사용 중인 총 메모리 양을 반환합니다. 여기에는 DLSS가 내부적으로 캐시할 수 있는 모든 메모리도 포함됩니다. 따라서 모든 DLSS 인스턴스를 해제하더라도 할당된 메모리가 0이 아닌 값으로 표시될 수 있습니다. 하지만 Shutdown()이 호출되면 캐시된 모든 메모리가 해제됩니다.

또한 기본적으로 ReleaseFeature()를 호출하면 기능에 사용된 메모리는 해제되지 않고 내부적으로 캐시되어 있다가 기능을 즉시 다시 생성할 때 이 메모리가 재사용된다는 점에 **유의하세요.** 그러나 DLSS는 릴리스 기능() 호출 시마다 메모리를 강제로 해제하는 NVSDK_NGX_Parameter_FreeMemOnReleaseFeature 힌트를 지원합니다. 이 힌트는 다음과 같이 설정할 수 있습니다:

```
NVSDK_NGX_파라미터_설정I(ngxParams, NVSDK_NGX_파라미터_프리멤온릴리즈기능, 1)).
```

3.15 현재 프레임 바이어스

NVIDIA는 DLSS의 특징 추적을 개선하기 위한 방법을 지속적으로 연구하고 있습니다. 때때로 DLSS 피처 트래킹 품질이 저하되어 특정 피처가 "지워지거나" 움직이는 피처 뒤에 "고스트" 또는 흔적이 생길 수 있습니다. 이는 다음에서 발생할 수 있습니다:

1. 작은 입자(예: 눈 또는 먼지 입자).
2. 애니메이션/스크롤 텍스처를 표시하는 오브젝트입니다.
3. 매우 얇은 물체(예: 전선).
4. 모션 벡터가 누락된 오브젝트(많은 파티클 이펙트).

5. 매우 큰 값을 갖는 모션 벡터를 가진 오브젝트 제외(예: 빠르게 움직이는 자동차 아래에서 노면이 제외되는 경우).

테스트 중에 문제가 있는 에셋이 발견되면 DLSS에 이전 프레임의 색상에 대해 수신 컬러 버퍼를 바이어스하도록 지시하는 것도 한 가지 옵션입니다. 이렇게 하려면 문제가 있는 에셋을 나타내는 가려지지 않은 픽셀에 플래그를 지정하는 2D 바이너리 마스크를 생성하고 이를 `BiasCurrentColor` 파라미터로 DLSS 평가 호출에 전송합니다. 그러면 DLSS 모델은 마스크에 플래그가 지정된 픽셀에 대해 대체 기술을 사용합니다.

마스크 자체는 컬러 버퍼 입력과 동일한 해상도여야 하며, R8G8B8A8_UNORM을 사용해야 합니다. /R16_FLOAT/R8_UNORM 형식(또는 R 구성 요소가 있는 모든 형식)이며 값이 "1.0"이어야 합니다. 다른 모든 픽셀이 "0.0"으로 설정된 마스크된 픽셀입니다.

참고: 이 마스크는 DLSS 통합이 완료되고 완전히 작동하는 것으로 확인된 후에 문제가 있는 자산에만 사용하세요. 마스크 테두리 내에서 에일리어싱이 증가할 수 있습니다.

3.16 멀티뷰 및 가상 현실 지원

DLSS는 기본적으로 가상 현실(VR)을 포함한 여러 보기를 지원하며, 각 보기에 하나씩 연결된 여러 DLSS 인스턴스를 생성합니다(VR의 경우 각 눈에 해당할 수 있음).

여러 뷰에서 DLSS를 사용하려면 `NGX_<API>_CREATE_DLSS_EXT`에 대한 표준 호출을 사용하여 여러 DLSS 인스턴스를 생성하고 현재 업샘플링 중인 뷰와 연결된 DLSS 인스턴스 핸들에서 DLSS 평가 호출을 수행합니다.

많은 VR 타이틀은 양쪽 눈을 동일한 렌더링 타겟에 렌더링합니다. 이 경우 하위 직사각형을 사용하여 렌더링 대상의 적절한 관심 영역으로 DLSS를 제한합니다(자세한 내용은 섹션 5.4 참조). 하위 사각형 매개변수는 다음과 같이 지정합니다:

1. 하위 사각형의 '밑면'(왼쪽 상단 모서리)입니다.
2. 동적 해상도를 사용하는 경우 *입력 하위 직*사각형의 경우 인스턴스 핸들 생성 시 지정된 입력/출력 차원 또는 평가 시 지정된 입력 차원으로 가정되는 하위 직사각형의 크기입니다(자세한 내용은 3.2.2 섹션 참조).
3. *출력 하위 사*각형의 크기는 동적 해상도 사용 여부와 관계없이 인스턴스 핸들 생성 시 지정된 출력 치수로 가정합니다.

참고: 하위 직사각형을 사용하여 DLSS가 출력할 *출력 리소스의* 하위 영역을 지정하려면 DLSS 인스턴스 핸들 생성 시 `NVSDK_NGX_DLSS_Create_Params`의 `InEnableOutputSubrects` 플래그를 `true`로 설정해야 합니다(자세한 내용은 섹션 5.3 참조). 하위 직사각형은 `InEnableOutputSubrects` 설정 여부와 관계없이 *입력 리소스에서* 지원됩니다.

3.17 현재 DLSS 설정

DLSS SDK의 일부로 포함된 DLSS 라이브러리에는 영구 워터마크(아래 빨간색으로 표시됨)와 현재 DLSS 매개변수 및 설정에 대한 특정 정보를 화면에 표시하는 옵션(아래 녹색으로 표시됨)이 포함되어 있습니다. 이는 디버깅 목적으로 제공되며 프로덕션 라이브러리에는 워터마크가 포함되지 **않습니다.**

참고: 최종 사용자 컴퓨터에는 DLSS 매개변수와 디버그 줄이 표시되지 않지만 프로덕션 DLSS 라이브러리에는 버전 번호와 같은 일부 DLSS 정보가 표시됩니다.

개발자 또는 테스터의 컴퓨터에 적절한 레지스트리 키가 설정되어 있는지 확인합니다(다음 단락 참조). 이것은 버그가 아닙니다.

DLSS 디버그 정보 줄(위 녹색으로 표시된 부분)을 활성화하려면 SDK 아카이브에 있거나 GitHub 리포지토리의 "`Utils`" 폴더에 있는 레지스트리 파일 "`dlss_debug_onscreendisplay_on.reg`"를 찾습니다. Windows 탐색기에서 파일을 두 번 클릭하고 변경 사항을 Windows 레지스트리에 병합합니다. 디버그 줄을 비활성화하려면 "`dlss_debug_onscreendisplay_off.reg`" 파일에 대해서도 동일한 절차를 따릅니다.

개발 및 디버그 빌드의 경우, DLSS 온스크린 표시기를 활성화하는 레지스트리 키는

HKEY_LOCAL_MACHINE\SOFTWARE\NVIDIA Corporation\Global\NGXCore\ShowDlssIndicator

입니다.

DWORD여야 합니다. 양수 값이면 DLSS의 개발 및 디버그 빌드에서 DLSS 표시기를 활성화합니다. 반대로 DLSS의 릴리스 빌드에서는 **1024** 값이 있어야 DLSS 표시기를 활성화할 수 있습니다.

Linux의 경우 환경 변수 NGX_SHOW_INDICATOR=1을 설정합니다.

3.17.1 DLSS 정보 라인 및 디버그 핫키

왼쪽에서 오른쪽으로, 위에서 아래로 DLSS 디버깅 정보가 표시됩니다:

1. DLSS 버전(이 예제에서는 v2.0.11).
2. 현재 사용 중인 3D API(DirectX 11, DirectX 12 또는 Vulkan).
3. 대상 배율로 렌더링. 첫 번째 숫자는 입력 버퍼 크기(즉, 현재 렌더링 대상 크기)이고 두 번째 숫자는 출력 버퍼 크기(즉, 디스플레이 해상도)입니다.
4. 나머지 줄에는 타이밍 정보가 표시됩니다(사용 중인 드라이버 및 시스템 설정에 따라 타이밍이 0 또는 예상 실행 시간으로 표시될 수 있음).
5. 디버깅 단축키는 두 번째 줄에서 시작하여 세 번째 줄로 이어집니다:
    5.1. 현재 디버그 오버레이 이름이 먼저 출력됩니다. 사용 가능한 디버그 오버레이를 순환하려면 CTRL+ALT+F12를 사용합니다. 디버그 오버레이 목록과 사용 방법은 섹션 8.2를 참조하세요.

5.2. 화면 오른쪽 상단에 창으로 표시되는 디버그 오버레이와 전체 화면 표시 사이를 전환하려면 CTRL+ALT+F11을 사용하세요.

5.3. 디버그 '누적 모드'를 토글하려면 CTRL+ALT+F6을 사용합니다. 누적 모드 사용 방법에 대한 자세한 내용은 섹션 8.3을 참조하세요.

5.4. NaN 시각화를 전환하려면 CTRL+ALT+O를 사용합니다. NaN은 밝은 빨간색으로 표시됩니다. 입력의 NaN을 디버깅하려면 위에서 설명한 대로 디버그 오버레이를 순환합니다.

5.5. 다양한 지터 오프셋을 주기적으로 무효화하려면 CTRL+ALT+F10을 사용하세요. 지터 디버깅에 대한 자세한 내용은 섹션 8.4를 참조하고 다양한 조합에 대한 전체 목록은 아래 3.17.1.1을 참조하세요.

5.6. X 및 Y 지터 오프셋을 교환하려면(즉, DLSS가 들어오는 X 오프셋을 Y 오프셋으로 사용하거나 그 반대로 사용하도록 하는 경우) CTRL+ALT+F10을 사용합니다. 지터 디버깅에 대한 자세한 내용은 섹션 8.4를 참조하세요.

3.17.1.1 지터 오프셋 구성

서브 픽셀 지터 관련 문제를 디버깅하는 데 도움을 주기 위해 DLSS SDK 라이브러리에서 선택적으로 CTRL+ALT+F9 단축키를 사용하여 지터 오프셋 구성 요소를 조정할 수 있습니다. 구성은 다음과 같습니다:

```
// 기본적으로 지터 오프셋은 엔진에서 전송된 대로 사용됩니다
. 구성 0: OFF

// 두 벡터 컴포넌트를 반으로 줄이고 음수화하는 조합 구성 1:
        지터오프셋X *= 0.5f; 지
        터오프셋Y *= 0.5f;
구성 2:
        지터오프셋X *= 0.5f; 지터오프셋
        Y *= -0.5f;
구성 3:
        지터오프셋X *= -0.5f; 지터오프
        셋Y *= 0.5f;
구성 4:
        지터오프셋X *= -0.5f; 지터오프셋
        Y *= -0.5f;

// 두 벡터 컴포넌트를 배가 및 음수화하는 조합 구성 5:
        지터오프셋X *= 2.0f; 지
        터오프셋Y *= 2.0f;
구성 6:
        지터오프셋X *= 2.0f; 지터오프셋
        Y *= -2.0f;
구성 7:
        지터오프셋X *= -2.0f; 지터오프
        셋Y *= 2.0f;
구성 8:
        지터오프셋X *= -2.0f; 지터오프셋
        Y *= -2.0f;

// 벡터 구성 요소 하나 또는 둘 모두를 부정하는 조합 구성 9:
        지터오프셋X *= 1.0f; 지터오프셋
        Y *= -1.0f;
구성 10:
        지터오프셋X *= -1.0f; 지터오프
        셋Y *= 1.0f;
구성 11:
        지터오프셋X *= -1.0f; 지터오프셋
        Y *= -1.0f;

// 개별 벡터 컴포넌트를 반으로 나누고 무효화하는 조합 구성 12:
        JitterOffsetX *= 0.5f; 구
성 13:
        JitterOffsetY *= 0.5f; 구
성 14:
        지터오프셋X *= -0.5f;
구성 15:
        지터오프셋Y *= -0.5f;
```

```
// 개별 벡터 컴포넌트를 배가 및 음수화하는 조합 구성 16:
        지터오프셋X *= 2.0f; 구
성 17:
        JitterOffsetY *= 2.0f; 구
성 18:
        JitterOffsetX *= -2.0f;
Config 19:
        지터오프셋Y *= -2.0f;
```

3.18 NGX 로깅

DLSS 화면 디버깅 정보에 충분한 세부 정보가 제공되지 않는 경우 NGX에서 생성되는 자세한 로그가 더 있습니다. 이러한 로그는 NGX 초기화 시 포함된 경로의 파일에 저장되며, 원하는 경우 별도의 콘솔 창에 표시할 수도 있습니다.

최신 DLSS 디버깅 레지스트리 키는 "`utils`" 디렉터리에서 사용할 수 있습니다.

1. "`ngx_log_on.reg`" 파일은 NGX 로깅 시스템을 활성화합니다(즉, 로그 파일이 생성됨).
2. "`ngx_log_off.reg`" 파일은 NGX 로깅 시스템을 비활성화합니다(즉, 로그 파일이 생성되지 않습니다. 생성됨).
3. "`ngx_log_verbose.reg`"는 NGX 로깅의 상세 수준을 활성화합니다.
4. "`ngx_log_window_on.reg`"를 사용하면 로그를 실시간으로 보여주는 별도의 온스크린 콘솔 창을 표시할 수 있습니다.
    a. 특정 전체 화면 게임과 애플리케이션에서 NGX 로깅 창을 사용할 때 예기치 않은 동작이 나타날 수 있습니다. 이 경우 창 모드에서 게임을 실행하거나 "`ngx_log_window_off.reg`"를 실행하여 NGX 로깅 창을 비활성화하세요.
5. "`ngx_log_window_off.reg`"는 별도의 온스크린 콘솔 창을 비활성화합니다.

중요: 로깅을 위해 NGX에 제공된 경로가 쓰기 불가능한 경로인 경우 NGX 또는 DLSS가 자동으로 로드 또는 초기화에 실패할 수 있습니다. 개발자는 로깅이 활성화된 경우 유효한 경로를 제공해야 합니다. "`%사용자프로파일%\앱데이터\로컬\`"에 디렉터리를 생성하고 로깅에 사용하는 것이 일반적인 옵션입니다.

또한 앱이 설정할 수 있는 다른 로깅 관련 설정 중에서 앱이 직접 NGX에서 사용하는 로깅 수준을 높이고

콜백을 통해 앱으로 NGX 로그 라인을 다시 파이프하도록 할 수도 있습니다. 이러한 기능은 앱이 NGX를 초기화할 때 전달할 수 있는 `NVSDK_NGX_FeatureCommonInfo` 구조체에서 추가 파라미터를 설정하여 앱에서 사용할 수 있습니다. 자세한 내용은 아래 섹션 5.2를 참조하세요.

Linux의 경우 환경 변수 NGX_LOG_LEVEL=1을 설정합니다.

3.19 샘플 코드

최신 샘플 앱은 DLSS 개발자 영역 페이지에서 찾을 수 있으며, DLSS SDK의 각 릴리스에 포함된 독립된 ZIP 파일로 번들로 제공됩니다:

- https://developer.nvidia.com/dlss-getting-started

컴파일 및 빌드 지침은 `.../DLSS_Sample_App/README.md` 에 있습니다.

샘플 앱은 NVIDIA "도넛" 프레임워크를 사용하여 작성되었습니다. 애플리케이션 코드는 "`.../DLSS_Sample_App/ngx_dlss_demo`"에 있습니다. "`NGXWrapper.cpp`" 파일에는 "`DemoMain.cpp`"에서 호출되는 NGX 호출이 포함되어 있습니다.

3.20 Streamline SDK를 사용하여 DLSS 통합

DLSS는 Streamline SDK를 통해 모든 애플리케이션에 통합할 수 있습니다. 다음이 필요합니다:

- SDK 간소화
- Streamline용 DLSS 플러그인 - Streamline SDK에서 *sl.dlss.dll* 사용 가능
- DLSS 바이너리 - sl.dlss.dll에서 사용할 수 있는 호환 가능한 *nvngx_dlss.dll 입니다.*

Streamline용 DLSS 플러그인(*sl.dlss.dll)*은 *nvngx_dlss.dll 의* 래퍼 역할을 합니다. Streamline SDK를 사용하여 애플리케이션에 DLSS를 통합하는 방법에 대한 자세한 내용은 해당 SDK의 일부로 제공되는 프로그래밍 가이드를 참조하세요:

https://github.com/NVIDIAGameWorks/Streamline

4 게임 내 DLSS 배포

NVIDIA는 개발자가 최소한의 빌드 또는 패키징 변경으로 기능을 사용할 수 있도록 DLSS 기술을 캡슐화했습니다. 이 섹션의 단계에 따라 게임 또는 애플리케이션 빌드에 DLSS를 포함할 수 있습니다.

4.1 DLSS 릴리스 프로세스

NVIDIA는 모든 개발 파트너를 위해 DLSS를 공개적으로 호스팅했습니다. 최종 사용자에게 최상의 경험을 제공하기 위해 개발자는 다양한 DLSS 모드, 출력 해상도 및 다양한 게임 콘텐츠에서 DLSS를 철저

하게 테스트할 것을 권장합니다.

4.2 배포 가능한 라이브러리

DLSS용 릴리스 라이브러리는 GitHub 저장소의 "`./lib/<plat>_<arch>/rel/`" 폴더에 있습니다. 게임 설치 후 파일이 게임 실행 파일(또는 플러그인을 빌드하는 경우 DLL)과 같은 폴더에 있는지 확인하세요.

중요: 개발 DLL은 "`./lib/<plat>_<arch>/dev/`"에 위치하며 **공개 릴리스에 포함하거나 배포해서는 안 됩니다**. 이 라이브러리에는 화면 알림이 포함되어 있으며 디버깅 전용으로 설계된 오버레이가 있습니다(섹션 8.2 참조).

NGX AI 기반 기능	NGX 기능 DLL
DLSS	nvngx_dlss.dll / libnvidia-ngx-dlss.so

참고: 필요한 경우 대체 디렉토리 경로에서 DLSS 라이브러리를 로드할 수 있습니다. 진입점인 `NVSDK_NGX_D3D12_Init_ext`는 검색할 경로 목록이 포함된 `NVSDK_NGX_PathListInfo` 항목이 있는 `InFeatureInfo` 파라미터를 받습니다. 자세한 내용은 아래 섹션 5.1과 5.2를 참조하세요.

4.2.1 DLSS 라이브러리 제거

게임 또는 애플리케이션의 인스톨러는 다른 구성 요소와 동일한 방식으로 DLSS 라이브러리를 처리하고 제거할 때 라이브러리를 제거해야 합니다.

4.2.2 DLSS 라이브러리 서명

DLSS DLL은 암호화 방식으로 서명되어 NVIDIA 드라이버(NGX Core)에 의해 안전하게 로드됩니다. 빌드 또는 패키징 프로세스 중에 게임 또는 애플리케이션 DLL이 서명되는 경우, 기존 NVIDIA 서명에 서명을 추가해야 합니다(NVIDIA 서명을 제거하지 마십시오).

중요: DLSS DLL에 NVIDIA 서명이 누락되거나 손상된 경우 NGX Core가 라이브러리를 로드할 수 없으며 DLSS 기능이 실패합니다.

4.2.3 타사 코드 포함에 대한 공지

게임 또는 애플리케이션 패키지의 일부가 될 DLSS DLL에는 최종 제품(게임 또는 애플리케이션)의 공개 문서에서 승인해야 하는 타사 코드가 포함되어 있습니다.

섹션 9.5에 있는 저작권 및 라이선스 고지 전문을 포함하세요.

5 DLSS 코드 통합

5.1 프로젝트에 DLSS 추가하기

NGX DLSS SDK에는 4개의 헤더 파일이 포함되어 있습니다:

- nvsdk_ngx.h
- nvsdk_ngx_defs.h
- nvsdk_ngx_params.h
- nvsdk_ngx_helpers.h

헤더 파일은 "`./include`" 폴더에 있습니다. 게임 프로젝트에서 다음을 포함합니다.
`nvsdk_ngx_helpers.h`.

참고: 벌칸 헤더는 위의 명명 규칙에 따라 "_vk" 접미사를 사용하며 벌칸 애플리케이션에 반드시 포함되어야 합니다.

개발 중에는 NGX 런타임이 DLL을 제대로 찾아서 로드할 수 있도록 GitHub의 "`./lib/<plat>_<arch>/dev/`" 디렉터리에서 메인 게임 실행 파일 또는 DLL이 있는 폴더에 `nvngx_dlss.dll / libnvidia-ngx-dlss.so`를 복사합니다. 게임 또는 게임 빌드 또는 패키징 시스템에서 요구하는 경우 DLSS 라이브러리를 메인 실행 파일과 다른 위치에 패키징할 수 있습니다. 이 경우 엔트리 포인트인 `NVSDK_NGX_D3D12_Init_ext`는 검색할 경로 목록이 포함된 `NVSDK_NGX_PathListInfo` 항목이 있는 `InFeatureInfo` 파라미터를 받습니다. 자세한 내용은 섹션 5.1을 참조하십시오.

5.1.1 Windows에서 NGX SDK 연결하기

프로젝트는 NGX 헤더 파일을 포함할 뿐만 아니라 링크 대상도 포함해야 합니다:

1. **`nvsdk_ngx_s.`**`lib`(프로젝트가 정적 런타임 라이브러리(**/MT**) 링크를 사용하는 경우),

   또는

2. **`nvsdk_ngx_d.`**`lib`(프로젝트가 동적 런타임 라이브러리(**/MD**) 링크를 사용하는 경우).

두 파일은 모두 "`./lib/<plat>_<arch>`" 폴더에 있습니다.

5.1.2 Linux에서 NGX SDK 연결하기

프로젝트는 NGX 헤더 파일을 포함할 뿐만 아니라 링크 대상도 포함해야 합니다:

`libsdk_nvngx.a`

및 `libstdc++.`**so**

와

`libdl.so`

5.2 NGX SDK 오브젝트 초기화

DLSS는 NVIDIA 디스플레이 드라이버의 일부로 제공되는 NGX의 기능입니다. NGX SDK는 지원되는 API(Vulkan, D3D11, D3D12 및 CUDA)에 유사한 호출 집합을 사용하므로 다음 샘플 코드와 일치하거나 매우 유사한 코드를 사용하여 NGX를 초기화(그리고 DLSS를 초기화)할 수 있습니다. 호출이 사용 중인 게임 또는 렌더링 엔진에서 사용하는 API와 일치하는지 확인합니다. API 간의 상호 운용성(예: D3D11에서 CUDA로)은 NGX SDK 외부에서 게임 또는 애플리케이션이 처리해야 합니다.

벌칸에 대한 추가 참고 사항: 애플리케이션은 2.3.1절에 언급된 API에서 쿼리한 대로 인스턴스 및 디바이스 확장을 활성화해야 합니다.

NGX SDK 인스턴스를 초기화하려면 다음 방법 중 하나를 사용합니다:

```
유형 정의 구조체 NVSDK_NGX_PathListInfo
{
    wchar_t **경로;
    // 경로 목록 길이
    부호 없는 int 길이;
} NVSDK_NGX_PathListInfo;

유형 정의 열거형 NVSDK_NGX_Logging_Level
{
    nvsdk_ngx_logging_level_off = 0, nvsdk_ngx_logging_level_on,
    nvsdk_ngx_logging_level_verbose, nvsdk_ngx_logging_level_num
} NVSDK_NGX_Logging_Level;

// 앱에서 제공하는 로깅 콜백으로, 로그 라인을 앱으로 다시 파이프할 수 있도록 합니다.
// 서명 및 호출 규칙에 유의하세요.
// 콜백은 모든 스레드에서 호출할 수 있어야 합니다.
// 또한 완전히 스레드에 안전해야 하며 여러 스레드가 동시에 호출할 수 있어야 합니다.
// 반환될 때까지 메시지를 완전히 처리해야 하며, 다음과 같이 보장할 수 없습니다.
// 메시지가 반환된 후에도 여전히 유효하거나 할당됩니다.
// 메시지는 널로 끝나는 문자열이며 여러 바이트 문자를 포함할 수 있습니다. #if
defined( __GNUC__ ) || defined( __clang__ )
typedef void NVSDK_CONV(*NVSDK_NGX_AppLogCallback)(const char* message,
NVSDK_NGX_Logging_Level loggingLevel, NVSDK_NGX_Feature sourceComponent); #else
typedef void(NVSDK_CONV* NVSDK_NGX_AppLogCallback)(const char* message,
NVSDK_NGX_Logging_Level loggingLevel, NVSDK_NGX_Feature sourceComponent);
#endif

유형 정의 구조체 NGSDK_NGX_LoggingInfo
{
    // 아래 필드는 SDK 버전 0x0000014에 도입되었습니다.

    // 앱 제공 로깅 콜백 NVSDK_NGX_AppLogCallback 로깅 콜백;

    // 사용할 최소 로깅 수준입니다. 이보다 높은 경우
    // 로깅 수준을 다르게 설정하면 재정의됩니다.
    // 해당 로깅 수준입니다. 그렇지 않으면 해당 로깅 레벨이 사용됩니다.
    NVSDK_NGX_Logging_Level MinimumLoggingLevel;

    // 앱 로그 이외의 싱크에 로그 줄 쓰기를 비활성화할지 여부
    //콜백.
    // 앱이 로깅 콜백을 제공하는 경우 유용할 수 있습니다. 로깅 콜백은
    //null 및 포인트
    // 참으로 설정하면 유효한 로깅 콜백으로 전환합니
    다. bool DisableOtherLoggingSinks;

} NGSDK_NGX_LoggingInfo;

유형 정의 구조체 NVSDK_NGX_FeatureCommonInfo
{
    // 기능 dll을 찾기 위한 모든 경로를 검색 순서에 따라 내림차순으로 나열합니다.
    //에 기본 경로인 애플리케이션 폴더가 아닌 다른 경로
    를 입력합니다. NVSDK_NGX_PathListInfo PathListInfo;

    // NGX에서 내부적으로 사용
    // SDK 버전 0x0000013에 도입된
    NVSDK_NGX_FeatureCommonInfo_Internal* 내부 데이터;
```

```
    // 아래 필드는 SDK 버전 0x0000014에 도입되었습니다.

    NGSDK_NGX_LoggingInfo 로깅 정보;

} NVSDK_NGX_FeatureCommonInfo;

// NVSDK_NGX_Init ----------
//
//
// InApplicationId:
//         NVIDIA에서 제공한 고유 ID
//
// 인앱애플리케이션데이터경로:
//         로그 및 기타 임시 파일을 저장할 폴더(쓰기 권한 필요)
//
// InDevice: [d3d11/12 전용]
//         사용할 DirectX 장치
//
// InFeatureInfo:
//         모든 기능에 공통된 정보를 포함하며, 현재는 모든 경로의 목록만 있습니다.
//         기본 경로인 애플리케이션 디렉터리 외에 기능 dll이 위치할 수 있는 경로를 지정할 수 있습
니다.
//
// InSDKVersion:
//          설치된 드라이버가 지원하는 데 필요한 최소 API 버전입니다.
//          사용자 컴퓨터에 설치됩니다. 특정 SDK 기능을 지원하려면 최소 API 버전이 필요합니다.
//          를 호출합니다. SDK 초기화 함수에 대한 호출
//          드라이버가 API 버전 InSDKVersion 이상을 지원하지 않는 경우 실패합니다. 드라이버의
//          애플리케이션은 필요한 InSDKVersion 세트에 적합한
//          SDK 기능.
//
// 설명:
//         새 SDK 인스턴스를 초기화합니다.
//

NVSDK_NGX_Result NVSDK_CONV NVSDK_NGX_D3D11_Init(부호 없는 긴 긴 InApplicationId, const wchar_t
*InApplicationDataPath, ID3D11Device *InDevice, const NVSDK_NGX_FeatureCommonInfo)
*InFeatureInfo = nullptr, NVSDK_NGX_Version InSDKVersion = NVSDK_NGX_Version_API);

NVSDK_NGX_Result NVSDK_CONV NVSDK_NGX_D3D12_Init(부호 없는 긴 긴 InApplicationId, const wchar_t
*InApplicationDataPath, ID3D12Device *InDevice, const NVSDK_NGX_FeatureCommonInfo)
*InFeatureInfo = nullptr, NVSDK_NGX_Version InSDKVersion = NVSDK_NGX_Version_API);



#IFDEF C플러스플러스
NVSDK_NGX_Result NVSDK_CONV NVSDK_NGX_VULKAN_Init(부호 없는 long long InApplicationId, const
wchar_t *InApplicationDataPath, VkInstance InInstance, VkPhysicalDevice InPD, VkDevice InDevice,
const NVSDK_NGX_FeatureCommonInfo *InFeatureInfo = nullptr, NVSDK_NGX_Version InSDKVersion =
NVSDK_NGX_Version_API);
#기타
NVSDK_NGX_Result NVSDK_CONV NVSDK_NGX_VULKAN_Init(부호 없는 long long InApplicationId, const
wchar_t *InApplicationDataPath, VkInstance InInstance, VkPhysicalDevice InPD, VkDevice InDevice,
const NVSDK_NGX_FeatureCommonInfo *InFeatureInfo, NVSDK_NGX_Version InSDKVersion); #endif

// NGX 반환 코드를 문자열로 변환하는 유틸리티는 공식적인 용도가 아닌 디버그/로깅 보
조 도구로만 사용됩니다.
const wchar_t* NVSDK_CONV GetNGXResultAsString(NVSDK_NGX_Result InNGXResult);
```

유형 정의 열거형 NVSDK_NGX_EngineType
{

```
    custom = 0,
    언리얼, 유니
    티, 옴니버스
    ,
  NUM_GENERIC_ENGINEERS
} NVSDK_NGX_EngineType;


//////////////////////////////////////////////////////////////////////////////////////////////////
/////
// NVSDK_NGX_Init_with_ProjectID -------
//
//
// InProjectId:
//      사용된 렌더링 엔진에서 제공한 고유 ID
//
// InEngineType:
//      애플리케이션/플러그인에서 사용하는 렌더링 엔진입니다.
//      특정 엔진 유형이 명시적으로 지원되지 않는 경우
NVSDK_NGX_ENGINE_TYPE_CUSTOM을 사용합니다.
//
// InEngineVersion:
//      애플리케이션/플러그인에서 사용하는 렌더링 엔진의 버전 번호입니다.
//
// 인앱애플리케이션데이터경로:
//      로그 및 기타 임시 파일을 저장할 폴더(쓰기 권한 필요),
//      일반적으로 이 위치는 문서 또는 프로그램 데이터의 위치입니다.
//
// InDevice: [d3d11/12 전용]
//      사용할 DirectX 장치
//
// InFeatureInfo:
//      모든 기능에 공통된 정보를 포함하며, 현재는 모든 경로의 목록만 있습니다.
//      기본 경로인 애플리케이션 디렉터리 외에 기능 dll이 위치할 수 있는 경로를 지정할 수 있습
니다.
//
// 설명:
//      새 SDK 인스턴스를 초기화합니다.
//
NVSDK_NGX_Result NVSDK_CONV NVSDK_NGX_D3D11_Init_with_ProjectID(const char *InProjectId,
NVSDK_NGX_EngineType InEngineType, const char *InEngineVersion, const wchar_t)
*InApplicationDataPath, ID3D11Device *InDevice, const NVSDK_NGX_FeatureCommonInfo
*InFeatureInfo = nullptr, NVSDK_NGX_Version InSDKVersion = NVSDK_NGX_Version_API);

NVSDK_NGX_Result NVSDK_CONV NVSDK_NGX_D3D12_Init_with_ProjectID(const char *InProjectId,
NVSDK_NGX_EngineType InEngineType, const char *InEngineVersion, const wchar_t)
*InApplicationDataPath, ID3D12Device *InDevice, const NVSDK_NGX_FeatureCommonInfo
*InFeatureInfo = nullptr, NVSDK_NGX_Version InSDKVersion = NVSDK_NGX_Version_API);

NVSDK_NGX_Result NVSDK_CONV NVSDK_NGX_CUDA_Init_with_ProjectID(const char *InProjectId,
NVSDK_NGX_EngineType InEngineType, const char *InEngineVersion, const wchar_t)
*InApplicationDataPath, const NVSDK_NGX_FeatureCommonInfo *InFeatureInfo = nullptr, NVSDK_NGX_Version
InSDKVersion = NVSDK_NGX_Version_API)로 설정합니다;

NVSDK_NGX_Result NVSDK_CONV NVSDK_NGX_VULKAN_Init_with_ProjectID(const char *InProjectId,
NVSDK_NGX_EngineType InEngineType, const char *InEngineVersion, const wchar_t)
*InApplicationDataPath, VkInstance InInstance, VkPhysicalDevice InPD, VkDevice InDevice, const
NVSDK_NGX_FeatureCommonInfo *InFeatureInfo = nullptr, NVSDK_NGX_Version InSDKVersion =
NVSDK_NGX_Version_API)를 설정합니다;
```

특정 SDK 기능에는 특정 최소 드라이버 버전이 필요할 수 있습니다. 드라이버가 지원해야 하는 필수 기능은 각 API의 SDK 초기화 함수에 전달되는 `InSDKVersion` 값에 따라 게이트됩니다. 따라서 애플리케이션에 SDK API 버전 0x0000014 이상에서만 사용할 수 있는 기능이 필요한 경우 `InSDKVersion`에 `0x0000014`를 전달해야 합니다. 해당 기능이 필요하지 않은 경우에는 더 낮은 값(예: `0x0000013`)을 전달해야 하며, 이는 대부분의 SDK 기능을 지원하고 대부분의 사용자 컴퓨터에 설치된 드라이버에서도 널리 지원되므로 일반적으로 대부분의 애플리케이션에서 좋은 기준 API 버전으로 사용됩니다. 애플리케이션에서 더 높은 API 버전이 필요한 SDK 기능을 사용하는지 확인하려면 SDK 헤더를 참조하세요. 다양한 SDK 기능에 필요한 최소 API 버전은 NGX SDK 헤더에 문서화되어 있으며, 구조체 `NVSDK_NGX_FeatureCommonInfo`의 정의를 예시로 참조하시기 바랍니다.

5.2.1 프로젝트 ID

프로젝트 ID는 언리얼 또는 옴니버스와 같은 특정 3사 엔진(rd)에 고유한 고유 ID(InProjectId)를 의미합니다. 해당 엔진의 DLSS 통합은 이미 해당 값을 DLSS에 전달하는 작업을 처리하고 있을 것입니다. 엔진의 에디터에서 프로젝트 ID가 설정되어 있는지 확인하세요.

예를 들어 프로젝트 ID는 GUID와 같은 것으로 가정합니다:

“a0f57b54-1daf-4934-90ae-c4035c19df04"

커스텀 엔진 타입의 경우 드라이버는 전달된 프로젝트 ID가 GUID 유사성 요구 사항을 충족하는지 확인합니다. NVSDK_NGX_*_Init_with_ProjectID()에서 오류가 발생하면 NGX 로그를 검사하여 정확한 문제를 확인할 수 있습니다.

각 섹션에 설명된 대로 프로젝트 ID, 엔진 유형 및 엔진 버전과 함께 NVSDK_NGX_*_Init_with_ProjectID() 함수를 사용합니다.

***참고**: NVIDIA 개발자 기술 담당자가 NVIDIA 애플리케이션 ID(`InApplicationId`)를 제공한 경우, NVSDK_NGX_*_Init() 함수를 사용하여 NGX를 초기화하고 프로젝트 ID 대신 지정된 애플리케이션 ID를 전달해 주세요.*

5.2.2 엔진 유형

애플리케이션에서 사용하는 렌더링 엔진(InEngineType)을 나타냅니다.

5.2.3 엔진 버전

엔진 버전(InEngineVersion)은 코어 엔진에서 보고하는 버전과 동일해야 합니다.

5.2.4 스레드 안전

NGX API는 스레드 안전하지 **않습니다.** 클라이언트 애플리케이션은 필요에 따라 스레드 안전이 적용되도록 해야 합니다. 여러 스레드에서 동일한 NGX 기능 및 관련 NGX 파라미터 객체에 대한 평가, 호출 또는 호출을 수행하면 예측할 수 없는 동작이 발생할 수 있습니다.

5.2.5 컨텍스트 및 명령 목록

DirectX 11을 사용하는 DLSS 타이틀의 경우, NGX API는 즉각적인 D3D11 컨텍스트의 상태를 유지합니다. D3D12 명령 목록이나 Vulkan 명령 버퍼는 **그렇지 않습니다.** 다음을 사용하는 애플리케이션의 경우

DirectX 12 또는 Vulkan을 사용하는 경우 클라이언트 애플리케이션은 필요에 따라 명령 목록과 명령 버퍼 상태를 관리해야 합니다.

5.2.6 NGX 기능의 가용성 확인 및 파라미터 맵 할당

NGX SDK 인스턴스를 성공적으로 초기화하면 대상 시스템에서 NGX 기능을 실행할 수 있음을 나타냅니다. 그러나 각 기능에는 최소 드라이버 버전과 같은 추가 종속성이 있을 수 있습니다. 따라서 특정 기능(예: DLSS)을 사용할 수 있는지 확인하는 것이 좋습니다. 이를 위해 NGX는 NGX 런타임에서 제공하고 다음 API를 사용하여 얻을 수 있는 읽기 전용 파라미터를 쿼리하는 데 사용할 수 있는 NVSDK_NGX_Parameter 인터페이스를 제공합니다:

```
//////////////////////////////////////////////////////////////////////////////////////////////////
// NVSDK_NGX_AllocateParameters
// -------------------------------------------------------
//
// OutParameters:
//      SDK에 필요한 파라미터를 설정하는 데 사용되는 파라미터 인터페이스
//
// 설명:
//      이 인터페이스를 사용하면 명명된 필드를 사용하여 간단한 매개변수 설정을 할당할 수 있습니다.
//      앱이 관리해야 하는 수명입니다.
//      예를 들어 Set(NVSDK_NGX_Parameter_Denoiser_Width,100)을 호출하여 너비를 설정할 수 있습니다.
//      를 호출하여 CUDA 버퍼 포인터를 제공하거나
//      Set(NVSDK_NGX_파라미터_디노이저_색상,쿠다버퍼)
//      자세한 내용은 샘플 코드를 참조하세요.
//      NVSDK_NGX_AllocateParameters에 의해 출력되는 파라미터 맵은 다음을 사용하여 해제해서는 안 됩니다.
//      매개변수 맵을 해제하려면 해제/삭제 연산자
//      출력은 NVSDK_NGX_AllocateParameters, NVSDK_NGX_DestroyParameters를 사용해야 합니다.
//      NVSDK_NGX_GetParameters와 달리 다음을 사용하여 할당된 파라미터 맵은
//      NVSDK_NGX_AllocateParameters
//      는 앱에서 NVSDK_NGX_DestroyParameters를 사용하여 삭제해야 합니다.
//      또한 NVSDK_NGX_GetParameters와 달리 파라미터 맵은 다음과 같이 출력됩니다.
//      NVSDK_NGX_AllocateParameters
//      에는 NGX 기능 및 사용 가능한 기능이 미리 채워져 있지 않습니다.
//      이러한 정보로 미리 채워진 새 매개변수 맵을 만들려면 다음과 같이 하세요,
//      NVSDK_NGX_GetCapabilityParameters
//      를 사용해야 합니다.
//      이 함수는 이전 드라이버를 사용하는 경우 NVSDK_NGX_Result_FAIL_OutOfDate를 반환할 수 있습니다.
//      이 API 호출을 지원하지 않는 경우 사용 중입니다. 이러한 경우 NVSDK_NGX_GetParameters
//      를 대체할 수 있습니다.
//      이 함수는 NVSDK_NGX_Init 호출에 성공한 후에만 호출할 수 있습니다.
//
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV
NVSDK_NGX_D3D11_AllocateParameters(NVSDK_NGX_Parameter** OutParameters);
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV
NVSDK_NGX_D3D12_AllocateParameters(NVSDK_NGX_Parameter** OutParameters);
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV
NVSDK_NGX_VULKAN_AllocateParameters(NVSDK_NGX_Parameter** OutParameters);

//////////////////////////////////////////////////////////////////////////////////////////////////
// NVSDK_NGX_GetCapabilityParameters ------------------------
```

```
//
//
// OutParameters:
//          NGX 및 기능으로 채워진 파라미터 인터페이스
```

```
//
// 설명:
//        이 인터페이스를 통해 앱에서 새 매개변수 맵을 만들 수 있습니다.
//        NGX 기능 및 사용 가능한 기능으로 미리 채워져 있습니다.
//        출력 매개변수 맵은 어떤 용도로도 사용할 수 있습니다.
//        NVSDK_NGX_AllocateParameters가 출력하는 파라미터 맵은 다음 용도로 사용할 수 있습니다.
//        를 사용하는 것이 좋지만, NVSDK_NGX_GetCapabilityParameters
//        NGX 기능 및 사용 가능한 기능을 쿼리하지 않는 한
//        매개변수 맵을 미리 채우는 것과 관련된 오버헤드 때문입니다.
//        NVSDK_NGX_GetCapabilityParameters에서 출력되는 파라미터 맵은 다음을 사용하여 해제해서는 안 됩니다.
//        매개변수 맵을 해제하려면 해제/삭제 연산자
//        NVSDK_NGX_GetCapabilityParameters, NVSDK_NGX_DestroyParameters의 출력은 다음과 같아야 합니다.
//        사용됨.
//        NVSDK_NGX_GetParameters와 달리 다음을 사용하여 할당된 파라미터 맵은
//        NVSDK_NGX_GetCapabilityParameters
//        는 앱에서 NVSDK_NGX_DestroyParameters를 사용하여 삭제해야 합니다.
//        이 함수는 이전 드라이버를 사용하는 경우 NVSDK_NGX_Result_FAIL_OutOfDate를 반환할 수 있습니
다.
//        이 API 호출을 지원하지 않습니다. 이 함수는 다음 경우에만 호출할 수 있습니다.
//        를 성공적으로 호출한 후 NVSDK_NGX_Init을 호출합니다.
//        NVSDK_NGX_GetCapabilityParameters가 NVSDK_NGX_Result_FAIL_OutOfDate로 실패하는 경우,
//        파라미터 맵을 미리 가져오기 위해 NVSDK_NGX_GetParameters를 폴백으로 사용할 수 있습니다.
//        NGX 기능 및 사용 가능한 기능으로 채워집니다.
//
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV
NVSDK_NGX_D3D11_GetCapabilityParameters(NVSDK_NGX_Parameter** OutParameters);
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV
NVSDK_NGX_D3D12_GetCapabilityParameters(NVSDK_NGX_Parameter** OutParameters);
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV
NVSDK_NGX_VULKAN_GetCapabilityParameters(NVSDK_NGX_Parameter** OutParameters);

//////////////////////////////////////////////////////////////////////////////////////////////////
//////
// NVSDK_NGX_DestroyParameters ----------------------------
//
//
// InParameters:
//        삭제할 매개변수 인터페이스
//
// 설명:
//        이 인터페이스를 사용하면 앱에서 전달된 매개변수 맵을 삭제할 수 있습니다. 일단
//        매개변수 맵에서 NVSDK_NGX_DestroyParameters가 호출됩니다.
//        를 다시 사용해서는 안 됩니다.
//        반환된 파라미터 맵에서 NVSDK_NGX_DestroyParameters를 호출해서는 안 됩니다.
//        의 수명을 관리할 수 있습니다.
//        매개변수 맵.
//        이 함수는 이전 드라이버를 사용하는 경우 NVSDK_NGX_Result_FAIL_OutOfDate를 반환할 수 있습니
다.
//        이 API 호출을 지원하지 않습니다. 이 함수는 다음 경우에만 호출할 수 있습니다.
//        를 성공적으로 호출한 후 NVSDK_NGX_Init을 호출합니다.
//
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV
NVSDK_NGX_D3D11_DestroyParameters(NVSDK_NGX_Parameter* InParameters);
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV
NVSDK_NGX_D3D12_DestroyParameters(NVSDK_NGX_Parameter* InParameters);
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV
NVSDK_NGX_VULKAN_DestroyParameters(NVSDK_NGX_Parameter* InParameters);

//////////////////////////////////////////////////////////////////////////////////////////////////
//////----------------------------------------------------
// NVSDK_NGX_GetParameters
//
//
```

```
// OutParameters:
//         SDK에 필요한 파라미터를 설정하는 데 사용되는 파라미터 인터페이스
//
// 설명:
//         이 인터페이스를 사용하면 명명된 필드를 사용하여 간단한 매개변수 설정을 할 수 있습니다.
//         예를 들어 Set(NVSDK_NGX_Parameter_Denoiser_Width,100)을 호출하여 너비를 설정할 수 있습니다.
//         를 호출하여 CUDA 버퍼 포인터를 제공하거나
//         Set(NVSDK_NGX_파라미터_디노이저_색상,쿠다버퍼)
//         자세한 내용은 샘플 코드를 참조하세요. 할당된 메모리
//         는 NGX에 의해 해제되므로 해제/삭제 연산자를 호출해서는 안 됩니다.
//         NVSDK_NGX_GetParameters에 의해 출력되는 파라미터 맵도 미리 채워져 있습니다.
//         NGX 기능 및 사용 가능한 기능을 제공합니다.
//         NVSDK_NGX_AllocateParameters와 달리, 파라미터 매핑은
//         NVSDK_NGX_GetParameters
//         NGX가 수명을 관리하며, 다음과 같이 해서는 안 됩니다.
//         를 사용하여 앱에서 NVSDK_NGX_DestroyParameters를 사용하여 삭제할 수 있습니다.
//         NVSDK_NGX_GetParameters는 더 이상 사용되지 않으며 앱은 다음을 사용하도록 전환해야 합니다.
//         가능한 경우 NVSDK_NGX_AllocateParameters 및 NVSDK_NGX_GetCapabilityParameters를 사용합니다.
//         그럼에도 불구하고 사용자가 구형 드라이버를 사용할 가능성이 있기 때문에
//         버전에서 다음과 같은 경우 NVSDK_NGX_GetParameters를 폴백으로 계속 사용할 수 있습니다.
//         NVSDK_NGX_AllocateParameters
//         또는 NVSDK_NGX_GetCapabilityParameters는 NVSDK_NGX_Result_FAIL_OutOfDate를 반환합니다.
//
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV NVSDK_NGX_D3D11_GetParameters(NVSDK_NGX_Parameter)
**아웃파라미터);
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV NVSDK_NGX_D3D12_GetParameters(NVSDK_NGX_Parameter)
**아웃파라미터);
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV NVSDK_NGX_VULKAN_GetParameters(NVSDK_NGX_Parameter)
**아웃파라미터);
```

앱은 NVSDK_NGX_GetCapabilityParameters 인터페이스를 사용하여 NGX에서 제공하는 위에서 언급한 읽기 전용 파라미터로 미리 채워진 새 파라미터 맵을 할당해야 합니다. 사용자가 구형 드라이버를 사용하는 경우 NVSDK_NGX_GetCapabilityParameters가 NVSDK_NGX_Result_FAIL_OutOfDate로 실패할 수 있다는 점에 유의하세요. 이 경우 앱은 NVSDK_NGX_GetParameters를 사용하여 위에서 언급한 읽기 전용 파라미터로 미리 채워진 파라미터 맵을 가져오는 것으로 되돌아갈 수 있습니다. 그러나 NVSDK_NGX_GetParameters는 더 이상 사용되지 않으며, 새로운 드라이버 버전이 널리 설치될 때까지 앱에서 이를 대체 수단으로 사용하는 것은 임시방편으로 사용된다는 점에 유의하세요.

특정 앱에서 NGX 기능이 거부되는 경우가 있습니다. NVSDK_NGX_GetParameters를 열거형과 함께 사용하여 쿼리할 수도 있습니다: NVSDK_NGX_Parameter_SuperSampling_ FeatureInitResult.

> **참고**: NVSDK_NGX_GetParameters에 의해 출력된 NVSDK_NGX_Parameter 객체는 종료 호출 중에 NGX에 의해 자동으로 해제됩니다. 그러나 NVSDK_NGX_GetCapabilityParameters에 의해 출력된 파라미터 객체는 앱에서 NVSDK_NGX_DestroyParameters를 사용하여 소멸시켜야 합니다.

예를 들어, 벌칸에서 실행할 때 사용자 시스템에서 DLSS를 사용할 수 있는지 확인하려면 다음을

사용하세요:

```
int DLSS_Supported = 0; int
needsUpdatedDriver = 0;
부호 없는 int minDriverVersionMajor = 0; 부
호 없는 int minDriverVersionMinor = 0;
```

```
NVSDK_NGX_Parameter *Params = nullptr;
bool bShouldDestroyCapabilityParams = true;
NVSDK_NGX_Result 결과 결과 = NVSDK_NGX_VULKAN_GetCapabilityParameters(&Params);

if(Result != NVSDK_NGX_Result_Success)
{
   bShouldDestroyCapabilityParams = false;
   NVSDK_NGX_VULKAN_GetParameters(&Params);
}

NVSDK_NGX_Result 결과 업데이트 드라이버 =
 Params->Get(NVSDK_NGX_Parameter_SuperSampling_NeedsUpdatedDriver, &needsUpdatedDriver);

NVSDK_NGX_Result 결과 최소 드라이버 버전 메이저 =
 Params->Get(NVSDK_NGX_Parameter_SuperSampling_MinDriverVersionMajor, &minDriverVersionMajor);
NVSDK_NGX_Result ResultMinDriverVersionMinor =
 Params->Get(NVSDK_NGX_Parameter_SuperSampling_MinDriverVersionMinor, &minDriverVersionMinor);

if (NVSDK_NGX_SUCCEED(ResultUpdatedDriver))
{
    if (needsUpdatedDriver)
    {
        // 오래된 드라이버로 인해 NVIDIA DLSS를 로드할 수 없습
        니다. if
        (NVSDK_NGX_SUCCEED(ResultMinDriverVersionMajor) &&
            NVSDK_NGX_SUCCEED(ResultMinDriverVersionMinor))
        {
            // 필요한 최소 드라이버 버전: minDriverVersionMajor.minDriverVersionMinor
        }
        // 기본 AA 솔루션(TAA 등)으로의 폴백
    }
    else
    {
        // 드라이버 업데이트가 필요하지 않으므로 응용 프로그램은 다음과 같이 예상되지 않습니
        다.
    }   // 이 경우 minDriverVersion을 쿼리합니다.
}

NVSDK_NGX_Result ResultDlssSupported ==
 Params->Get(NVSDK_NGX_Parameter_SuperSampling_Available,&DLSS_Supported);

if (NVSDK_NGX_FAILED(ResultDlssSupported) || !DLSS_Supported )
{
 // 이 하드웨드/플랫폼에서는 NVIDIA DLSS를 사용할 수 없습니다.

 // 기본 AA 솔루션(TAA 등)으로의 폴백
}

ResultDlssSupported =
 Params->Get(NVSDK_NGX_Parameter_SuperSampling_FeatureInitResult,&DLSS_Supported);

if (NVSDK_NGX_FAILED(ResultDlssSupported) || !DLSS_Supported )
{
 // 이 애플리케이션에 대해 NVIDIA DLSS가 거부되었습니다.

 // 기본 AA 솔루션(TAA 등)으로의 폴백
}

if(bShouldDestroyCapabilityParams)
{
   NVSDK_NGX_VULKAN_DestroyParameters(Params);
```

```
    매개변수 = nullptr;
}
```

5.2.7 기능 거부 재정의

DLSS를 사용하는 애플리케이션 개발자는 애플리케이션에서 해당 기능을 비활성화할 경우 발생할 수 있는 상황을 테스트할 수 있도록 기능 거부 여부를 재정의할 수 있는 기능이 있어야 합니다.

기능 거부 메커니즘에 대한 오버라이드는 Windows 레지키를 통해 지원되어야 합니다(NGX에서 대부분의 다른 오버라이드가 지원되는 방식과 유사). 이 오버라이드를 위한 모든 regkey는 HKLM\SOFTWARE\NVIDIA Corporation\Global\NGXCore에 기록해야 합니다.

이 오버라이드를 개발하는 데 필요한 다양한 레지스트리 키는 다음과 같습니다:

오버라이드 기능 거부

a. 유형: REG_DWORD
b. 값: 재정의 활성화 여부 - 0 = 재정의 비활성화, 0이 아닌 경우: 재정의 활성화
c. 기본 동작: 재정의 비활성화

오버라이드기능거부_CMSID

a. 유형: REG_SZ
b. 값입니다: 재정의되는 특정 애플리케이션의 CMS ID; 앱에 고유한 CMS ID가 없거나 앱 개발자가 앱의 CMS ID를 모르는 경우 이 regkey를 생성하지 마십시오.
c. 기본 동작: 모든 앱에 재정의 동작 적용

오버라이드 기능 거부_기능

a. 유형: REG_SZ
b. 값: 재정의할 특정 기능의 기능 이름; 모든 기능을 재정의할 경우 regkey를 만들지 마십시오.
c. 기본 동작: 모든 기능에 재정의 동작 적용

오버라이드 기능 거부_거부

a. 유형: REG_DWORD
b. 값: 기능을 거부할지 여부 - 0 = 허용, 1 = 거부
c. 기본 동작: 허용 Linux에

서는 재정의가 아직 지원되지 않습니다.

5.2.8 DLSS에 대한 최적의 설정 얻기

사용자 시스템에서 DLSS를 사용할 수 있는 것으로 확인되면 DLSS에 대한 최적의 렌더링 타겟 크기를 구합니다. 이 크기는 GPU 유형, RTX(레이트레이싱) 설정 여부 등 다양한 요인에 따라 달라집니다.

DLSS에 대한 최적의 해상도 설정을 얻으려면 아래 쿼리를 실행하세요. 개발자는 각 조합에 대해 반환된 `DLSSMode`를 확인해야 합니다:

-대상 디스플레이 해상도 크기

- `PerfQualityValue`

참고: 현재 NGX DLSS 헤더에 5개의 가능한 값이 있는 것으로 정의된 PerfQualityValue가 있습니다. 5개의 `PerfQualityValue` 값을 모두 확인해야 하며, 활성화된 경우 게임 UI에서 선택할 수 있어야 합니다(비활성화된 경우 숨겨져 있어야 함). 이러한 값에 대한 자세한 내용은 3.2.1 섹션을 참조하십시오.

```
//     게임이 다음과 같은 모든 조합을 반복한다고 가정합니다:
//
//     해상도(목표 너비, 목표 높이)
//     PerfQualityValue
//       [0 최대 성능, 1 균형, 2 최대 품질, 3 울트라 성능, 4 울트라 품질]

부호 없는 int RenderWidth, RenderHeight;
float Sharpness = 0.0f; // 선명도는 더 이상 사용되지 않습니다, 섹션 3.11을 참고하세요.

DLSSMode = NGX_DLSS_GET_OPTIMAL_SETTINGS(
      Params,
      타겟 너비, 타겟 높이, 퍼펙트 퀄리티 값,
      권장 최적 렌더링 폭, 권장 최적 렌더링 높이, 동적 최대 렌더링 크기 폭, 동적 최
      대 렌더링 크기 높이, 동적 최소 렌더링 크기 폭, 동적 최소 렌더링 크기 높이, &
      선명도);

if (RecommendedOptimalRenderWidth == 0 || RecommendedOptimalRenderHeight == 0) (
{
      // 이 PerfQuality 모드는 아직 사용할 수 없습니다.
       // 다른 퍼펙트 퀄리티 모드를 요청하세요.

}
else
{
      // 이 조합에 DLSS 사용
      // - 권장 최적 렌더 너비, 권장 최적 렌더 높이로 기능 만들기
      // - 최소와 최대를 포함한 (RenderWidth, RenderHeight)로 렌더링합니다.
      // - DLSS를 호출하여 (TargetWidth, TargetHeight)로 업스케일링합니다.
}
```

5.2.8.1 최적의 해상도 설정 쿼리:

5.3 기능 생성

필요한 모든 매개변수가 설정되면 DLSS 기능을 만듭니다. 다음 코드를 사용하여 기능을 만들 수 있습니다:

```
// 참조용 - nvngx_ngx_defs.h에서 가져옴
유형 정의 열거형 NVSDK_NGX_DLSS_Feature_Flags
{
    NVSDK_NGX_DLSS_Feature_Flags_IsInvalid          = 1 << 31,

    NVSDK_NGX_DLSS_Feature_Flags_None               = 0,
    NVSDK_NGX_DLSS_Feature_Flags_IsHDR              = 1 << 0,
    NVSDK_NGX_DLSS_Feature_Flags_MVLowRes           = 1 << 1,
```

```
    NVSDK_NGX_DLSS_Feature_Flags_MVJittered      = 1 << 2,
    NVSDK_NGX_DLSS_Feature_Flags_DepthInverted = 1 << 3,
    NVSDK_NGX_DLSS_Feature_Flags_Reserved_0     = 1 << 4,
    NVSDK_NGX_DLSS_Feature_Flags_AutoExposure  = 1 << 6,
} NVSDK_NGX_DLSS_Feature_Flags;

유형 정의 구조체 NVSDK_NGX_Feature_Create_Params
{
    부호 없는 int InWidth;              // NGX_DLSS_GET_OPTIMAL_SETTINGS 를 통해 얻습
    니다;                               // NGX_DLSS_GET_OPTIMAL_SETTINGS 를 통해 얻습
    니다 unsigned int InTargetWidth;    // 타깃 너비 (최종 해상도)
    부호 없는 int InTargetHeight;    //    타깃    높이(최종    해상도)
    NVSDK_NGX_PerfQuality_Value InPerfQualityValue;
      // UI에서 사용자 선택: 최대 성능, 밸런스, 최대 품질, 울트라 성능, 울트라 품질
} NVSDK_NGX_Feature_Create_Params;

유형 정의 구조체 NVSDK_NGX_DLSS_Create_Params
{
    NVSDK_NGX_Feature_Create_Params 기능;
    /*** 선택 사항 ***/
    int     InFeatureCreateFlags;    // NVSDK_NGX_DLSS_Feature_Flags의 조합
    bool    InEnableOutputSubrects;// 출력 리소스에서 서브 레코드를 사용할지 여부
} NVSDK_NGX_DLSS_Create_Params;

ngx_d3d11_create_dlss_ext(
CommandCtx,             // 명령 컨텍스트
&FeatureHandle[Node], // 새 기능에 대한 핸들
Params,                 // NVSDK_NGX_AllocateParameters &DlssCreateParams로 얻
은 파라미터             // NVSDK_NGX_DLSS_Create_Params 구조체의 파라미터
);

ngx_d3d12_create_dlss_ext(
CommandList[Node],  // 현재 GPU 노드에 대한 명령 목록
CreationNodeMask,   // 멀티 GPU 전용 (기본값 1)
VisibilityNodeMask  // 멀티 GPU 전용 (기본값 1)
&FeatureHandle[Node], // 새 기능에 대한 핸들
Params,                 // NVSDK_NGX_AllocateParameters &DlssCreateParams로 얻
은 파라미터             // NVSDK_NGX_DLSS_Create_Params 구조체의 파라미터
);

ngx_vulkan_create_dlss_ext(
VkCommandBuffer,    // 벌칸 명령 버퍼 생성 노드 마
스크,               // 멀티 GPU 전용 (기본값 1)
VisibilityNodeMask  // 멀티 GPU 전용 (기본값 1)
&FeatureHandle[Node], // 새 기능에 대한 핸들
Params,                 // NVSDK_NGX_AllocateParameters &DlssCreateParams로 얻
은 파라미터             // NVSDK_NGX_DLSS_Create_Params 구조체의 파라미터
);
```

참고: 사용자가 이전 드라이버 버전을 사용하는 경우 NVSDK_NGX_AllocateParameters가

NVSDK_NGX_Result_FAIL_OutOfDate와 함께 실패할 수 있습니다. 이 경우 앱은 위의 함수에 사용할 파라미터 맵을 가져오기 위해 NVSDK_NGX_GetParameters를 다시 사용해야 합니다. 그러나 NVSDK_NGX_GetParameters는 곧 지원 중단될 예정이며, 앱에서 이를 대체 수단으로 사용하는 것은 충분한 새 드라이버 버전이 널리 설치될 때까지 임시방편으로 사용하기 위한 것임을 유의하세요.

5.4 기능 평가

기능은 특정 알고리즘과 딥러닝 모델에 대한 추론 호출을 실행하여 평가됩니다. DLSS의 경우 평가 호출은 다음과 같습니다:

```
// D3D12        - NVSDK_NGX_D3D12_Feature_Eval_Params
// VULKAN       - NVSDK_NGX_VK_Feature_Eval_Params
typedef 구조체
NVSDK_NGX_D3D11_Feature_Eval_Params
{
    ID3D11resource* pInColor;          // 컬러 버퍼
    ID3D11resource* pInOutput;         // 출력 버퍼
    /*** DLSS의 경우 선택 사항 ***/.
    float          선명도;             // 폐기, 섹션 3.11 참조
} NVSDK_NGX_D3D11_Feature_Eval_Params

유형 정의 열거형 NVSDK_NGX_톤매퍼 유형
{
    nvsdk_ngx_tonemapper_string = 0,
    nvsdk_ngx_tonemapper_reinhard,
    nvsdk_ngx_tonemapper_oneoverluma,
    nvsdk_ngx_tonemapper_aces,
    nvsdk_ngx_tonemappertype_num
} NVSDK_NGX_ToneMapperType;

유형 정의 열거형 NVSDK_NGX_GBufferType
{
    nvsdk_ngx_gbuffer_albedo = 0,
    nvsdk_ngx_gbuffer_roughness,
    nvsdk_ngx_gbuffer_metallic,
    nvsdk_ngx_gbuffer_specular,
    nvsdk_ngx_gbuffer_subsurface,
    nvsdk_ngx_gbuffer_normals입니다,
    // 그려진 오브젝트의 고유 식별자 또는 오브젝트가 그려지는 방식
    NVSDK_NGX_GBUFFER_SHADINGMODELID,
    NVSDK_NGX_GBUFFER_MATERIALID,  // 머티리얼의 고유 식별자
    NVSDK_NGX_GBUFFERTYPE_NUM = 16
} NVSDK_NGX_GBufferType;

// D3D12        - D3D12_GBuffer
// VULKAN       - VK_GBuffer
유형 정의 구조체 NVSDK_NGX_D3D11_GBuffer
{
    ID3D11Resource* pInAttrib[GBUFFERTYPE_NUM];
} NVSDK_NGX_D3D11_GBuffer;

유형 정의 구조체 NVSDK_NGX_Coordinates
{
     부호 없는 int
     X; 부호 없는
         int Y;
} NVSDK_NGX_Coordinates;

// D3D12        - NVSDK_NGX_D3D12_DLSS_Eval_Params
// VULKAN       - NVSDK_NGX_VK_DLSS_Eval_Params
```

유형 정의 구조체

```
NVSDK_NGX_D3D11_DLSS_Eval_Params
{
    NVSDK_NGX_D3D11_Feature_Eval_Params 기능;
    ID3D11자원*                 pInDepth;           // 깊이 버퍼
    ID3D11Resource*             pInMotionVectors;   // MV 버퍼
```

```
    /* 픽셀 공간의 지터 오프셋 */
    float                   InJitterOffsetX;
    float                   InJitterOffsetY;
    NVSDK_NGX_Dimensions    InRenderSubrectDimensions;
    /*** 선택 사항 ***/
    int                     InReset;// 씬 전환에 대한 히스토리 버퍼를 초기화합니다.
    float                   InMVScaleX;         // 모션벡터 스케일 X
    float                   InMVScaleY;         // 모션벡터 스케일 Y
    ID3D11Resource*         투명도 마스크;
    ID3D11Resource*         피노출 텍스처;      // 노출 텍스처 1x1
    NVSDK_NGX_Coordinates   InColorSubrectBase;// 입력/출력 버퍼로 오프셋
    NVSDK_NGX_Coordinates   InDepthSubrectBase;
    NVSDK_NGX_Coordinates    InMVSubrectBase;
    NVSDK_NGX_Coordinates   InTranslucencySubrectBase;
    NVSDK_NGX_Coordinates   InOutputSubrectBase;
    float                   InPreExposure;// 해당되는 경우 사전 노출
    /*** 선택 사항 - 연구 목적으로만 ***/.
    /* 파티클 등과 같은 알파 블렌딩된 오브젝트를 식별하는 픽셀당 마스크입니다.
    */ NVSDK_NGX_D3D11_GBuffer   GBufferSurface;
    NVSDK_NGX_톤매퍼 유형       InToneMapperType;
    ID3D11Resource*         pInMotionVectors3D;
    ID3D11Resource*         pInIsParticleMask;
    ID3D11Resource*         애니메이션 텍스처 마스크;
    ID3D11Resource*         pInDepthHighRes;
    ID3D11Resource*         pInPositionViewSpace;
    float                   인프레임 시간 델타 인 엠섹;
    ID3D11Resource*         pInRayTracingHitDistance;
    ID3D11Resource*         피인모션벡터리플렉션;
} NVSDK_NGX_D3D11_DLSS_Eval_Params;

ngx_d3d11_evaluate_dlss_ext(
컨텍스트,               // 명령 컨텍스트
FeatureHandle,          // 새 기능에 대한 핸들
Params,                 // NVSDK_NGX_AllocateParameters로 얻은 파라미터
D3D11DlssEvalParams // NVSDK_NGX_D3D11_DLSS_Eval_Params 구조체의 파라미터
);

ngx_d3d12_evaluate_dlss_ext(
CommandList[Node],  // GPU 노드에 대한 명령 목록
FeatureHandle[Node], // 새 기능에 대한 핸들
Params,                 // NVSDK_NGX_AllocateParameters로 얻은 파라미터
D3D12DlssEvalParams // NVSDK_NGX_D3D12_DLSS_Eval_Params 구조체의 파라미터
);

ngx_vulkan_evaluate_dlss_ext(
VkCommandBuffer,       // 벌칸 명령 버퍼
&FeatureHandle[Node], // 기능에 대한 핸들
Params,                 // NVSDK_NGX_AllocateParameters로 얻은 파라미터
VkDlssEvalParams       // NVSDK_NGX_VK_DLSS_Eval_Params 구조체의 파라미터
);
```

중요: NGX는 Vulkan 및 D3D12 명령 목록 상태를 수정합니다. 호출 프로세스는 NGX가 기능을 호출하기 전과 후에 자체 Vulkan 또는 D3D12 상태를 저장하고 복원해야 합니다.

참고: 사용자가 이전 드라이버 버전을 사용하는 경우 NVSDK_NGX_AllocateParameters가 NVSDK_NGX_Result_FAIL_OutOfDate와 함께 실패할 수 있습니다. 이 경우 앱은 위의 함수에 사용할 파라미터 맵을 가져오기 위해 NVSDK_NGX_GetParameters를 다시 사용해야 합니다. 참고

그러나 곧 지원 중단될 예정이며, 새로운 드라이버 버전이 널리 설치될 때까지 앱에서 대체 수단으로 사용하는 것은 임시방편입니다.

5.4.1 벌칸 리소스 래퍼

리소스 래퍼는 다른 방법으로는 사용할 수 없는 메타데이터를 전달하는 데 사용됩니다. 이 래퍼 구조는 Vulkan 리소스를 직접 전달하는 대신 인자로 사용해야 하며, VkImageView 및 VkBuffer 리소스에 사용할 수 있습니다. 현재는 VkImageView 리소스만 NGX의 파라미터로 사용됩니다. 다음은 래퍼 구조체와 생성 함수 서명입니다:

```
유형 정의 열거형 NVSDK_NGX_Resource_VK_Type
{
    nvsdk_ngx_resource_vk_type_vk_imageview,
    nvsdk_ngx_resource_vk_type_vk_buffer
} NVSDK_NGX_Resource_VK_Type;

유형 정의 구조체 NVSDK_NGX_ImageViewInfo_VK {
    VkImageView ImageView;
    VkImage 이미지;
    VkImageSubresourceRange 서브리소스 범위; VkFormat
    포맷;
    부호 없는 int 폭; 부
    호 없는 int 높이;
} NVSDK_NGX_ImageViewInfo_VK;

유형 정의 구조체 NVSDK_NGX_BufferInfo_VK {
    VkBuffer 버퍼;
    부호 없는 int SizeInBytes;
} NVSDK_NGX_BufferInfo_VK;

유형 정의 구조체 NVSDK_NGX_Resource_VK {
    union {
        NVSDK_NGX_ImageViewInfo_VK 이미지뷰정보;
        NVSDK_NGX_BufferInfo_VK 버퍼정보;
    } 리소스;
    NVSDK_NGX_Resource_VK_Type 유형;
    bool ReadWrite; // 리소스에 읽기 및 쓰기 액세스가 가능하면 참입니다.
                    // VkImage의 경우: 연관된 VkImage에 대한 VkImageUsageFlags에는 다음이 포함됩니다.
                    // vk_image_usage_storage_bit
} NVSDK_NGX_Resource_VK;

static NVSDK_NGX_Resource_VK NVSDK_NGX_Create_ImageView_Resource_VK(VkImageView imageView, VkImage image, VkImageSubresourceRange subresourceRange, VkFormat 형식, 부호 없는 int width, 부호 없는 int height, bool readWrite)를 만듭니다;
```

5.5 기능 폐기

기능이 더 이상 필요하지 않은 경우 다음 메서드를 호출하여 기능을 해제해야 합니다:

```
// NVSDK_NGX_Release
// ---------------------------------
//
// InHandle:
//          출시될 기능에 대한 처리
//
```

```
// 설명:
//        지정된 핸들로 기능을 릴리스합니다.
//        핸들은 참조 카운트되지 않으므로
//        이 호출 이후에는 제공된 핸들을 사용하는 것은
//        유효하지 않습니다.

NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV NVSDK_NGX_D3D11_ReleaseFeature(NVSDK_NGX_Handle)
*인핸들);

NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV NVSDK_NGX_D3D12_ReleaseFeature(NVSDK_NGX_Handle)
*인핸들);

NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV NVSDK_NGX_VULKAN_ReleaseFeature(NVSDK_NGX_Handle)
*인핸들);
```

기능 핸들이 해제되면 더 이상 사용할 수 없습니다.

기능 핸들은 해당 명령 목록이 여전히 기능과 관련된 내부 상태 및 리소스를 참조할 수 있으므로 Evaluate_XXX() 호출에 사용된 명령 목록(Direct3D) 또는 명령 버퍼(Vulkan)가 더 이상 비행 중이 아닌 경우에만 해제해야 합니다.

5.6 종료

NGX SDK 인스턴스와 함께 할당된 모든 리소스를 해제하려면 다음 방법을 사용하세요:

```
// NVSDK_NGX_Shutdown
// ------------------------------------
//
// 설명:
//        현재 SDK 인스턴스를 종료하고 모든 리소스를 해제합니다.
//

NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV NVSDK_NGX_D3D11_Shutdown();
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV NVSDK_NGX_D3D12_Shutdown();
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV NVSDK_NGX_VULKAN_Shutdown(void);
```

Shutdown() 함수는 모든 관련 기능이 릴리스된 경우에만 한 번만 호출해야 합니다. 이는 해당 기능과 관련된 작업이 아직 실행 중이 아니라는 의미이기도 합니다.

5.7 두 번 이상 초기화 및 종료

SDK 버전 0x15부터 여러 기기를 지원합니다. 즉, 디바이스당 한 번씩 Init()을 두 번 이상 호출할 수 있습니다. 기존 Shutdown() 호출에는 디바이스 포인터가 없으므로 모든 디바이스에서 NGX를 종료합니다. 한 기기에서는 NGX를 종료하지만 다른 기기에서는 종료하지 않도록 지원하기 위해 Shutdown1(Device)라는 새로운 API가 도입되었습니다.

6 리소스 관리

DLSS를 사용하려면 특정 내부 리소스를 생성해야 합니다. 게임에서 NGX 리소스 수명 주기 및 VRAM 할당을 완전히 제어해야 하는 경우, 리소스 생성을 위한 특정 콜백을 등록할 수 있습니다.

및 파기. 또한 NVSDK_NGX_AllocateParameters 또는 NVSDK_NGX_GetCapabilityParameters를 사용하여 할당된 파라미터 맵은 앱에서 NVSDK_NGX_DestroyParameters를 사용하여 파기해야 합니다. 이를 위해 다음을 수행하면 됩니다:

6.1 D3D11 특정

콜백 등록하기:

```
void NGXBufferAllocCallback(D3D11_BUFFER_DESC *InDesc, ID3D11Buffer **OutResource)
{
    *OutResource = nullptr;
    HRESULT hr = MyDevice->CreateBuffer(InDesc, nullptr, OutResource); if
    (hr != S_OK)
    {
        // 오류 처리;
    }
}

파라미터->설정(NVSDK_NGX_파라미터_버퍼 할당 콜백, NGX버퍼 할당 콜백);
```

매개변수 맵 삭제하기:

```
NVSDK_NGX_D3D11_DestroyParameters(Params); // 참고: Params는 다음과 같아야 합니다.
// NVSDK_NGX_D3D11_AllocateParameters를 사용하여 할당되었습니다.
// 또는 곧 지원 중단될 NVSDK_NGX_D3D11_GetCapabilityParameters가 아닌
// NVSDK_NGX_D3D12_GetParameters. 할당된 파라미터 맵
// NVSDK_NGX_D3D11_GetParameters를 사용하면 NGX에서 해당 메모리를 관리합니다.
```

6.2 D3D12 특정

콜백 등록하기:

```
void NGXResourceAllocCallback(D3D12_RESOURCE_DESC *InDesc, int InState,
CD3DX12_HEAP_PROPERTIES *InHeap, ID3D12Resource **OutResource)
{
    *OutResource = nullptr;
    // 또는 캐시 등에서 일치하는 리소스를 가져옵니다.
    HRESULT hr = MyDevice->CreateCommittedResource(InHeap, D3D12_HEAP_FLAG_NONE, InDesc,
        (D3D12_RESOURCE_STATES)InState),
        nullptr, IID_PPV_ARGS(OutResource));
    if (hr != S_OK)
    {
        // 오류 처리
    }
}

파라미터->설정(NVSDK_NGX_파라미터_자원 할당 콜백, NGX자원 할당 콜백);
```

매개변수 맵 삭제하기:

```
NVSDK_NGX_D3D12_DestroyParameters(Params); // 참고: Params는 다음과 같아야 합니다.
// NVSDK_NGX_D3D12_AllocateParameters를 사용하여 할당되었습니다.
// 또는 곧 사용되지 않을 NVSDK_NGX_D3D12_GetCapabilityParameters가 아닙니다.
// NVSDK_NGX_D3D12_GetParameters. 할당된 파라미터 맵
```

```
// NVSDK_NGX_D3D12_GetParameters를 사용하면 NGX에서 해당 메모리를 관리합니다.
```

6.3 벌칸 전용

매개변수 맵 삭제하기:

```
NVSDK_NGX_VULKAN_DestroyParameters(Params); // 참고: Params는 다음과 같아야 합니다.
// NVSDK_NGX_VULKAN_AllocateParameters를 사용하여 할당되었습니다.
// 또는 곧 사용되지 않을 NVSDK_NGX_VULKAN_GetCapabilityParameters가 아닙니다.
// NVSDK_NGX_VULKAN_GetParameters. 할당된 파라미터 맵
// NVSDK_NGX_VULKAN_GetParameters를 사용하면 NGX에서 해당 메모리를 관리합니다.
```

6.4 공통

```
void NGXResourceReleaseCallback(IUnknown *InResource)
{
    // 필요에 따라 릴리스를 지연하거나 사용 사례에 따라 관리
    SAFE_RELEASE(InResource);
}

파라미터->설정(NVSDK_NGX_파라미터_리소스 릴리스 콜백, NGX
    리소스 릴리스 콜백);
```

참고: NGX는 클라이언트 측에서 전달된 DirectX 리소스에 대한 참조를 보유하지 않습니다.

7 멀티 GPU 지원

7.1 연결 모드

연동 모드에서 DLSS 기능을 생성할 때는 DLSS 기능이 생성되는 노드를 지정해야 합니다. 이를 위해 CreationNodeMask를 사용합니다. 기능 생성 시 사용되는 VisibilityNodeMask는 DLSS 기능이 내부적으로 생성하는 리소스를 나타냅니다.

기본적으로 DLSS 기능은 생성된 GPU에서 평가됩니다. VisibilityNodeMask에 포함된 모든 노드에서 DLSS 기능을 평가할 수 있습니다. 다른 노드에서 DLSS 기능을 평가하려면 NVSDK_NGX_EParameter_EvaluationNode 파라미터를 사용합니다.

아래 코드 예시는 연결된 모드에서 DLSS를 사용하는 방법을 보여줍니다.

```
UINT NodeCount = Device->GetNodeCount();
UINT VisibilityMask = (1 << NodeCount) - 1;
UINT CreationNodeMask = (1 << Node);

Status = NGX_D3D12_CREATE_DLSS(&FeatureHandle[Node], Params,
        MyCommandList[Node], Width, Height, PerfQualityValue,
        RTXValue, CreationNodeMask, VisibilityNodeMask));

// 기본 노드에서 평가
Status = NGX_D3D12_EVALUATE_DLSS(FeatureHandle[Node], Params,
   MyCommandList[Node], Color, MV, Output, PrevOutput, Depth));
```

```
if (NVSDK_NGX_FAILED(Status)) { // 오류 처리 };

// 기본 노드가 아닌 노드에서 평
가하면 (Node + 1 < NodeCount)
{
   Params->Set(NVSDK_NGX_EParameter_EvaluationNode, Node + 1);
   Status = NGX_D3D12_EVALUATE_DLSS(FeatureHandle[Node], Params,
      MyCommandList[Node + 1], Color, MV, Output, PrevOutput, Depth));
   if (NVSDK_NGX_FAILED(Status)) { // 오류 처리 };
}
```

참고: D3D11 기본 멀티 GPU/SLI(대체 프레임 렌더링 - AFR)를 사용하는 경우 멀티 GPU 지원은 NVIDIA 드라이버에서 직접 제공되며 추가 통합 요구 사항이 없습니다.

7.2 연결 해제 모드

연결 해제 모드에서는 각 GPU가 독립적이므로 각 GPU에서 다른 GPU와 독립적으로 NGX_Init()을 호출할 수 있습니다. 여기서 중요한 점은 NGX를 종료하는 방법이 여러 가지가 있다는 것입니다.

- NGX_Shutdown() 호출은 모든 GPU에서 NGX를 종료합니다;
- NGX_Shutdown1(Device *) API는 파라미터로 전달된 단일 디바이스에 대해서만 NGX를 종료합니다;
- NGX_Shutdown1(nullptr) 호출은 NGX_Shutdown() 호출과 동일하게 작동합니다;

이 경우와 관련된 또 다른 API는 NGX_VULKAN_CreateFeature1()입니다. 이 API는 VULKAN의 언링크 모드에서 피처를 생성하기 위해 특별히 도입되었습니다. 여기서 VULKAN이 다른 이유는 명령 버퍼 포인터에서 디바이스 포인터를 유추하는 것을 허용하지 않기 때문입니다. D3D11과 D3D12는 이를 허용하므로 연결 해제 모드에서 NGX_CreateFeature()가 정상적으로 작동합니다.

참고: 이 기능은 470+ 드라이버에서만 지원되며 NGX API 버전을 명시적으로 지정한 경우에만 지원됩니다. >= 0x15로 설정됩니다.

8 문제 해결

8.1 시각적 아티팩트를 유발하는 일반적인 문제

1. 잘못된 모션 벡터: 항상 모션 벡터를 시각화하고 검증할 수 있는지 확인하세요(아래 DLSS 디버

그 오버레이 섹션 참조).

a. 모션 벡터가 화면 공간의 픽셀을 현재 프레임에서 화면 공간의 이전 프레임 위치로 가져오는 방법을 나타내는 방향을 가리키고 있는지 확인합니다. 따라서 게임이 1080p로 실행 중이고 오브젝트가 화면 왼쪽 가장자리에서 오른쪽 가장자리로 이동한 경우 오른쪽 가장자리에 있는 해당 픽셀의 모션 벡터의 x 값은 -1080.0이 됩니다.

b. 모션 벡터가 16비트 또는 32비트 부동 소수점 값으로 표현되었는지 확인합니다. 정수 값을 사용하면 하위 픽셀 값이 고려되지 않습니다.

c. 모션 벡터가 화면 공간에 있는지 확인하고 동적 오브젝트의 움직임을 고려하세요.

d. 화면에 표시되는 모든 개체에 모션 벡터가 올바르게 계산되었는지 확인하세요. 모션 벡터가 누락되기 쉬운 일반적인 장소로는 애니메이션이 적용된 나뭇잎, 하늘 등이 있습니다.

2. 잘못되었거나 동기화되지 않은 지터 패턴: 섹션에 제공된 시퀀스를 사용하고 있는지 확인하세요. 3.7. 지터 패턴은 프레임 시간 등에 관계없이 일치해야 합니다.

3. TAA를 비활성화하면 엔진의 렌더링 방식(지터, 깊이, 모션 등)이 변경되어 결과적으로 DLSS가 중단될 수 있습니다. DLSS를 통합할 때는 TAA 렌더링 패스를 DLSS로 대체해야 하지만, TAA가 켜져 있을 때 활성화된 렌더링 파이프라인의 **다른 모든 항목은** 여전히 TAA가 활성화된 것처럼 실행되어야 합니다.

   a. DLSS에 대한 모든 입력이 제대로 지터링되었는지 확인합니다. 여기에는 지터가 적용되지 않았을 수 있는 보조 패스(예: 오브젝트 강조 표시)가 포함됩니다.

4. 잘못된 노출/사전 노출(3.9절 참조): 항상 올바른 노출 텍스처 또는 사전 노출 값을 전달해야 방지할 수 있습니다:

   a. 움직이는 물체의 고스트.

   b. 최종 프레임의 흐릿함, 밴딩 또는 픽셀화 또는 너무 어둡거나 너무 밝은 경우.

   c. 특히 움직이는 오브젝트의 앨리어싱.

   d. 어두운 장면에서 밝은 장면으로(또는 그 반대로) 이동할 때 장면 밝기가 뚜렷하게 지연되는 현상입니다.

5. VRAM이 부족한 상황에서 필요한 리소스(컬러, 모션 벡터 등)가 GPU에서 제거된 상태에서 DLSS를 호출하면 게임이 충돌할 수 있습니다. 이 경우 DLSS를 사용하기 전에 모든 리소스가 상주하도록 설정되어 있는지 확인하세요.

6. 화면 디버그 텍스트(섹션 3.8 참조)를 사용하여 DLSS 모듈로 전달되는 버퍼의 해상도를 확인합니다.

7. 잘못된 리소스 설정: 필요한 사용 플래그(3.4장 참조)를 사용하여 DLSS에 전달되는 버퍼를 생성

했는지 확인하세요. 그렇지 않으면 무엇이 잘못되었는지에 대한 추가 표시 없이 검은색으로 출력될 수 있습니다.

8.2 DLSS 디버그 오버레이

DLSS SDK 개발 라이브러리에는 DLSS에서 사용하는 입력을 시각화할 수 있는 디버깅 오버레이가 포함되어 있습니다. 다양한 디버그 레이어를 활성화하고 순환하려면 CTRL+ALT+F12 단축키를 사용하세요. 디버그 레이어는 화면 오른쪽 상단에 오버레이로 표시되며 다음과 같은 여러 가지 상태를 포함합니다:

1. 현재 입력 색상 버퍼
2. 현재 프레임의 모션 벡터

3. 현재 프레임의 깊이 버퍼(검은색은 파란색, 보라색, 분홍색을 통과하는 근거리 평면, 흰색은 원거리 평면)
4. 제출된 지터 오프셋의 이력을 XY 산점도 [-1, 1]로 표시합니다(녹색은 현재 오프셋, 흰색과 빨간색은 각각 올바른 경계[-0.5, 0.5] 내에 있는 오래된 오프셋과 경계 밖에 있는 오래된 오프셋을 나타냄). 오래된 내부 오프셋이 노란색으로 표시되면 권장 위상 수를 아직 충족하지 못했음을 의미합니다(자세한 내용은 3.7항 참조).
5. 현재 프레임 노출 스케일 텍스처
6. DLSS로 전달되는 불량 노출 텍스처를 진단하는 데 도움이 되는 패턴입니다.

디버그 시각화를 화면 오른쪽 상단의 오버레이 창과 전체 화면 간에 전환하려면 CTRL+ALT+F11 단축키를 사용하세요.

> **참고**: DLSS 프로덕션 라이브러리에는 디버그 오버레이가 포함되어 있지 **않습니다**.

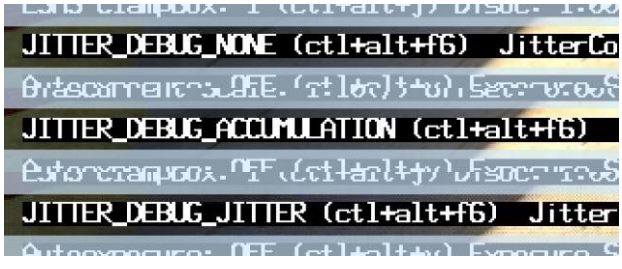
8.3 DLSS 디버그 누적 모드

DLSS SDK 라이브러리에는 모든 입력을 영구적으로 축적하는 특수 모드가 포함되어 있습니다. 이 모드는 정적 장면을 검사하여 모션 벡터 및 지터 관련 문제를 디버깅할 때 유용할 수 있습니다. 이 모드를 활성화하면 정적 장면이 에일리어싱, 흐릿함 또는 불분명한 영역 없이 *완벽하게* 해결되어야 합니다. 정적 장면은 DLSS(및 기타 AA 및 포스트)를 사용하지 않고 전체 해상도로 렌더링한 동일한 장면과 동일하게 보여야 합니다.

누적 모드를 활성화하려면 CTRL+ALT+F6 단축키를 누릅니다.

CTRL+ALT+F6 키는 세 가지 모드 사이를 순환합니다:

- JITTER_DEBUG_NONE : 일반 DLSS 표시
- JITTER_DEBUG_ACCUMULATION : 단순 누적 모드
- JITTER_DEBUG_JITTER: 특수 고급 지터 디버깅 모드입니다. 자세한 내용은 **오류! 참조 소스를 찾을 수 없습니다.** 아래에서 자세한 내용을 확인하세요.



참고: 누적 모드가 활성화된 상태에서 씬의 어떤 것이 움직이거나 카메라가 움직이면 고스트와 흔적이 *심하게* 나타납니다. 이는 예상되는 현상입니다.

8.4 지터 문제 해결

섹션 3.7에서 설명한 대로 DLSS는 저해상도 렌더링 프레임에서 고해상도 출력을 적절히 해결하기 위해 서브픽셀 지터가 필요합니다. 렌더링 엔진마다 서브픽셀 지터를 구현하는 방식이 다릅니다.

방식이 있습니다. 일부는 오프셋의 양을 뒤집거나 스케일을 조정하고, 일부는 카메라가 정적일 때만 지터를 적용하며, 일부는 모션 벡터에 지터를 추가합니다. 이러한 차이로 인해 DLSS가 프레임에 적용될 때 그래픽이 손상될 수 있습니다. 이러한 경우 다음을 시도하여 문제를 디버그하세요.

8.4.1 초기 지터 디버깅

1. 지터 오프셋 디버그 오버레이를 사용하여 지터 오프셋 값이 **항상** -0.5에서 +0.5(파란색 테두리 안쪽, 빨간색 픽셀 없음)로 설정합니다. 또한 분포 범위와 위상 수가 타겟에 충분한지 확인하고 첫 번째 "# 지터 오프셋" 값과 두 번째 값을 비교하여 해상도를 렌더링합니다(자세한 내용은 섹션 3.7 참조).
2. 모든 엔진 내 AA를 끄고 모든 포스트 프로세싱을 끕니다(또는 g버퍼에서 알베도만 렌더링).
3. 정적인 장면에 모션이 없는 경우, DLSS의 출력은 기본 전체 해상도로 렌더링된 동일한 장면과 일치해야 합니다(AA 및 포스트 프로세싱이 없는 경우).
4. 게임 또는 엔진에 디버그 핫키를 추가하여 전환할 수 있습니다:
    a. 렌더링 50%로 DLSS 켜기(일반적으로 "DLSS 성능 모드"라고 함); 그리고
    b. AA가 없는(그리고 DLSS가 비활성화된) 전체 해상도 렌더링.
5. 게임이 실행되는 동안 내장된 Microsoft Windows "화면 돋보기" 유틸리티를 사용하여 확대하고 네이티브 렌더링과 DLSS 사이를 전환할 수 있습니다.
    a. 화면 돋보기 단축키를 활성화합니다: 키보드의 Windows 로고 키 + 더하기 기호(+)
    b. 화면 돋보기 단축키를 비활성화합니다: Windows 로고 키 + Esc
    c. 화면 돋보기 설정으로 이동하여 "이미지 및 텍스트의 부드러운 가장자리"를 비활성화합니다.
    d. 전체 정보는 여기에서 확인할 수 있습니다: https://support.microsoft.com/en- us/help/11542/windows-사용-확대경으로-사물을-더-보기-쉬운-환경 만들기
6. 정적 장면을 렌더링하거나 장면에서 정적 오브젝트를 볼 때 DLSS의 출력이 흐릿하게 보이거나 깜빡이거나 정렬이 잘못되었다면 렌더러와 DLSS 사이에 무언가 문제가 있는 것입니다.
    a. 지터 오프셋 값이 **항상** -0.5에서 +0.5 사이인지 확인합니다. 이 범위를 벗어나지 **않아야** 합

니다. 이 범위를 벗어나면 렌더러가 뷰포트를 픽셀 외부로 이동하는 것입니다(일반적으로 DLSS를 켜고 끌 때 전체 화면이 흔들리거나 DLSS가 켜져 있을 때 심한 흐림이 발생합니다).

b. DLSS SDK 라이브러리에 내장된 CTRL+ALT+F10 단축키를 사용하여 지터 X 및 Y 오프셋을 교환해 보세요(자세한 내용은 섹션 3.8 참조).

    i. 축을 교환하여 문제가 해결되면 게임이나 엔진에서 지터 오프셋을 교환한 후 DLSS를 호출하세요.

c. DLSS SDK 라이브러리에 내장된 CTRL+ALT+F9 단축키를 사용하여 지터 입력 값을 무효화하고 스케일을 조정해 보세요(자세한 내용은 섹션 3.8 참조).

i. 한쪽 또는 양쪽 축을 무효화하거나 스케일링하면 문제가 해결되는 경우, 게임 또는 엔진에서 DLSS를 호출하기 전에 해당 컴포넌트의 값을 무효화하거나 스케일링합니다.

d. DLSS 디버그 오버레이(위 섹션 8.2 참조) 또는 다른 방법을 사용하여 전체 화면(또는 적어도 움직임이 없는 화면의 모든 정적 개체와 영역)에 대해 모션 벡터가 [0.0,0.0]인지 확인합니다.

i. 모션 벡터가 정확하지 않은 경우 필요에 따라 머티리얼, 오브젝트, 지형 또는 기타 시스템에 맞게 수정합니다. DLSS는 엔진이 매 프레임마다 정확한 픽셀 단위 모션 벡터를 제공한다고 가정합니다.

8.4.2 심층적인 지터 디버깅

위의 간단한 디버깅 단계로 문제가 해결되지 않으면 보다 심층적인 프로세스가 필요할 수 있습니다. 첫 번째 단계는 렌더링 엔진에서 구성 가능한 지터 패턴을 구현하는 것입니다.

1. 먼저, 렌더러가 매 프레임마다 특정 사분면으로 항상 지터링하는 "1 사분면 지터"를 추가하여 핫키로 네 개의 사분면을 모두 순서대로 순환시킵니다.
    a. 가능하면 렌더러가 현재 지터링 중인 사분면을 나타내는 화면 표시기 또는 디버그 텍스트를 추가합니다.
2. 둘째, 렌더러가 각 사분면을 자동으로 순환하면서 각 사분면에 한 프레임씩 지터를 추가하는 '4사분면 지터'를 추가합니다.

8.4.2.1 단일 쿼드런트 지터 디버깅

1. CTRL+ALT+F6 단축키를 사용하여 JITTER_DEBUG_JITTER 모드를 활성화합니다(자세한 내용은 위 8.3항 참조).
2. DLSS의 출력이 화면에 1:1 비율로 표시되는지 확인합니다. 이 디버그 모드는 엔진 자체의 업스케일링이나 엔진 창 크기 조정 등 DLSS가 적용된 후 스케일 변경이 발생하는 경우 유용하지 않습니다.
3. 게임에서 정적 장면을 찾아 렌더러가 알려진 고정된 사분면으로 지터링하는 장면을 렌더링합니다.

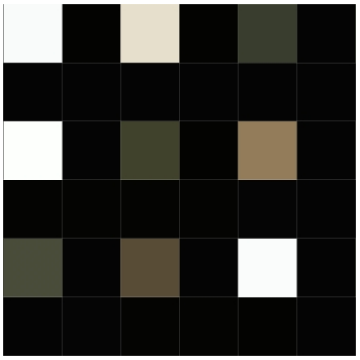
4. 화면 출력을 살펴보고 화면 돋보기를 사용하여 개별 픽셀을 더 잘 살펴보세요.

5. 렌더러가 올바르게 지터링을 수행하면 각 그룹의 한 픽셀은 컬러로, 세 픽셀은 완전히 검은색으로 구성된 4개의 픽셀 그룹이 표시되어야 합니다.

    a. 각 지터 사분면을 순환하고 색상이 올바른 사분면으로 이동하는지 확인합니다.
    를 누르면 3개의 픽셀이 완전히 검은색이 됩니다.

    b. 각 그룹에 검은색 픽셀이 3개 미만인 경우 지터의 양, 지터의 배율 또는 전송되는 지터의 부호가 올바르지 않습니다.

단일 사분면 지터가 있는 출력 예시.



최대 줌 상태에서 올바른 지터로 출력되는 단일 사분면 지터. 4개의 픽셀 그룹(3개의 검은색과 1개의 컬러/채움)에 주목하세요.

6. 3단계의 테스트 결과 지터가 올바르지 않은 것으로 나타나면 DLSS 디버그 단축키를 사용하여 지터 벡터 컴포넌트를 교환, 무효화 또는 스케일링해 보세요(위 8.4.1절 참조).

7. 지터 스케일을 변경하거나 컴포넌트를 무효화해도 문제가 해결되지 않으면 정적 씬을 찾아 보세요:

    a. 렌더러가 각 사분면으로 흔들리는 동안 스크린샷을 하나씩 캡처합니다(총 4개의 스크

린샷이 생성됩니다).

b. 전체 해상도 렌더링, AA, 포스트, DLSS를 켜고 스크린샷 한 장을 캡처합니다.

c. Adobe Photoshop 또는 다른 도구를 사용하여 4개의 DLSS 스크린샷을 결합하고 결과 이미지를 살펴봅니다(에일리어싱이 심하게 나타나는 것은 당연한 결과입니다). 결합된 DLSS 이미지를 전체 해상도 화면 캡처와 비교합니다. 한 방향으로 에일리어싱이 더 심하거나 세로 또는 가로로 에일리어싱이 더 뚜렷한지 살펴봅니다. 이러한 경우 렌더링 엔진에서 투사 매트릭스에 적용된 지터의 양을 확인하세요:

   i. 수직 에일리어싱이 더 많으면 Y축 지터를 확인합니다.

   ii. 마찬가지로 가로 앨리어싱이 더 많으면 X 축을 확인합니다.

8.4.2.2 4사분면 지터 디버깅

1. CTRL+ALT+F6 단축키를 사용하여 DLSS JITTER_DEBUG_JITTER 모드를 활성화합니다(자세한 내용은 위 섹션 8.3 참조).
2. DLSS의 출력이 화면에 1:1 비율로 표시되는지 확인합니다. 이 디버그 모드는 엔진 자체의 업스케일링이나 엔진 창 크기 조정 등 DLSS가 적용된 후 스케일 변경이 발생하는 경우 유용하지 않습니다.
3. 화면 돋보기를 사용하여 화면 출력을 검사하여 개별 픽셀을 더 잘 살펴봅니다.
4. 렌더러가 4개의 사분면 각각에 지터를 순환하고 DLSS를 활성화하면 최종 이미지가 에일리어싱되며 AA, DLSS 및 포스트 프로세싱 없이도 전체 해상도 렌더링과 일치합니다.
   a. 나무 판자, 나무 줄기, 금속 기둥, 울타리 기둥 또는 가장자리가 직선인 기타 물체와 같이 단순하고 정적인 평면 물체를 비교하는 것이 가장 쉽습니다.
   b. 입자, 투명한 물체 또는 그림자를 비교하지 마세요.
5. 3단계의 테스트에서 DLSS가 표시하는 내용과 기본 렌더링에 차이가 있는 경우 DLSS 디버그 핫키를 사용하여 지터 벡터 구성 요소를 교환, 무효화 또는 스케일링해 보세요(위 8.4.1 섹션 참조).
6. 지터 스케일을 변경하거나 컴포넌트를 무효화해도 문제가 해결되지 않으면 정적인 장면을 찾아 한 방향으로 에일리어싱이 더 심하거나 수직 또는 수평으로 에일리어싱이 더 뚜렷한지 살펴보세요. 이러한 경우 렌더링 엔진에서 투사 매트릭스에 적용된 지터의 양을 확인하세요:
   a. 수직 에일리어싱이 더 많으면 Y축 지터를 확인합니다.

b. 마찬가지로 가로 앨리어싱이 더 많으면 X 축을 확인합니다.

8.5 Linux

Linux의 경우 Ctrl+Alt+F<NR> 키를 누르면 터미널을 전환할 수 있습니다. 이 경우 ~/ngx/에 settings.ngxconfig라는 구성 파일을 사용하여 키프레스를 재정의할 수 있습니다. 예를 들어 다음과 같이 설정할 수 있습니다.

/root/ngx/settings.ngxconfig. settings.ngxconfig의 예입니다:

```
- dlss_debug
```

```
debug_accumulation_key = "ctl+alt+f6",                  ;
debug_jitter_config_rotate_key = "ctl+alt+f9",          ;
debug_jitter_swap_coordinates_key = "ctl+alt+f10", ;
디버그_시각화_크기_키 = "ctl+alt+fu",                   ; ctl+alt+f11 예제 대신
디버그_시각화_모드_키 = "ctl+alt+fv",                   ; ctl+alt+f12 대신 예제 보기
```

8.6 알려진 툴링 문제

1. RenderDoc은 DLSS 애플리케이션을 지원하지 않습니다.

8.7 오류 코드

NGX 실행 중에 오류가 감지되면 NGX는 다음 오류 코드 중 하나를 반환합니다:

- NVSDK_NGX_Result_FAIL_FeatureNotSupported
  기능이 현재 하드웨어에서 지원되지 않습니다.
- NVSDK_NGX_Result_FAIL_PlatformError
  플랫폼 오류 - 예: 자세한 내용은 d3d12 디버그 레이어 로그에서 확인하세요.
- NVSDK_NGX_Result_FAIL_FeatureAlreadyExists
  주어진 파라미터를 가진 기능이 이미 존재합니
  다.
- NVSDK_NGX_Result_FAIL_FeatureNotFound
  제공된 핸들이 있는 기능이 존재하지 않습니
  다.
- NVSDK_NGX_Result_FAIL_InvalidParameter
  잘못된 매개변수가 제공되었습니다.
- NVSDK_NGX_Result_FAIL_NotInitialized
  SDK가 제대로 초기화되지 않았습니다.
- NVSDK_NGX_Result_FAIL_Un지원되지 않는 입력 형
  식 입력/출력 버퍼에 사용되는 지원되지 않는 형식입

니다.

- NVSDK_NGX_Result_FAIL_RWFlagMissing
  피처 입력/출력에는 RW 액세스(UAV)가 필요합니다(d3d11/d3d12 전용).

- NVSDK_NGX_Result_FAIL_MissingInput
  특정 입력으로 기능이 생성되었지만 평가 시 제공되지 않는 기능입니다.

- NVSDK_NGX_Result_FAIL_UnableToInitializeFeature 기
  능이 잘못 구성되었거나 시스템에서 사용할 수 없습니다
  .

- NVSDK_NGX_Result_FAIL_OutOfDate
  NGX 런타임 라이브러리가 오래되어 업데이트가 필요합니다. 사용자에게 NVIDIA에서 제공하는 최신 디스플레이 드라이버를 설치하도록 안내합니다.

- NVSDK_NGX_Result_FAIL_OutOfGPUMemory
  이 기능을 사용하려면 시스템에서 사용할 수 있는 것보다 더 많은 GPU 메모리가 필요합니다.

- NVSDK_NGX_Result_FAIL_Un지원되는 형식
  입력 버퍼에 사용되는 형식은 기능에 따라 지원되지 않습니다.

- NVSDK_NGX_Result_FAIL_UnableToWriteToAppDataPath에
  제공된 경로에 쓸 수 없습니다.

- NVSDK_NGX_Result_FAIL_UnsupportedParameter
  지원되지 않는 매개변수가 제공됨(예: 특정 디스플레이 크기 또는 모드가 지원되지 않음)

- NVSDK_NGX_Result_FAIL_ 거부됨
  기능 또는 애플리케이션이 거부되었습니다(자세한 내용은 NVIDIA에 문의).

9 부록

9.1 DLSS 2.1.x에서 3.1.x로 전환하기

DLSS 버전 2.1.x에서 3.1.x로 이전하는 과정은 바이너리 호환이 가능하므로 원활하게 진행됩니다. v3.1.x에서는 몇 가지 새로운 기능을 사용할 수 있습니다:

1. OTA를 통한 사전 설정 업데이트(2.4_ 섹션 참조)
2. 모델 선택 API(섹션 3.12 참조)
3. 기능 지원 요구 사항 쿼리하기(2.3절 참조)
4. 벌칸 확장 요구 사항 쿼리하기(2.3.1 섹션 참조)
5. DLSS 선명화 사용 중단(섹션 3.11 참조)

9.2 DLSS 2.0.x에서 2.1.x로 전환하기

DLSS 버전 2.0.x에서 2.1로 이전하는 과정도 바이너리 호환이 가능하므로 원활하게 진행할 수 있습니다. 2.1.x 버전에서는 몇 가지 새로운 기능을 사용할 수 있습니다:

1. 동적 해상도 지원(3.2.2 섹션 참조)
2. 추가 DLSS 실행 모드 '초성능' 및 '초품질'(3.2.1절 참조)
3. VR 지원(섹션 3.14 참조)
4. 이제 렌더러에 필요한 경우 사전 노출 계수가 지원됩니다(3.9.2 섹션 참조).

9.3 마이너 개정 업데이트

이 섹션에서는 각 마이너 개정에 대한 주요 업데이트를 중점적으로 설명합니다:

1. 2.1.x~2.2.x - 자동 노출(섹션 3.10 참조)
2. 2.2.x ~ 2.3.x - 모델 개선 및 모든 개발자를 위한 DLSS 공개 공개

3. 2.3.x~2.4.x - 통합 간소화(섹션 3.19 참조)

9.4 향후 DLSS 매개변수

DLSS는 NVIDIA의 다양한 그룹에서 지속적으로 연구 및 개발 중인 알고리즘 및 신경망 제품군입니다. 이 연구의 일환으로 NVIDIA는 렌더링 엔진에서 생성되는 다양한 데이터를 사용하여 DLSS의 전반적인 이미지 품질과 성능을 개선하는 방법을 검토하고 있습니다.

다음은 DLSS 라이브러리가 선택적으로 허용할 수 있고 향후 DLSS 알고리즘에 사용될 수 있는 렌더링 엔진 리소스 목록입니다. 개발자가 이러한 파라미터의 일부 또는 전부를 포함할 수 있다면 NVIDIA의 지속적인 연구에 도움이 될 수 있으며 향후 개선된 알고리즘을 게임 또는 엔진 코드 변경 없이 사용할 수 있습니다.

이러한 리소스를 전달하는 방법에 대한 자세한 내용은 DLSS 헤더 파일을 참조하고 필요한 경우 NVIDIA 기술 담당자와 상의하시기 바랍니다. 또한 이러한 리소스를 준비하여 DLSS 라이브러리에 제공할 때 성능에 과도한 영향이 없는지 확인하세요.

1. G-버퍼:
    a. 알베도(지원 형식 - 8비트 정수)
    b. 거칠기(지원 형식 - 8비트 정수)
    c. 메탈릭(지원 형식 - 8비트 정수)
    d. 스페큘러(지원 형식 - 8비트 정수)1
    e. 서브서피스(지원 형식 - 8비트 정수)
    f. 노멀(지원 포맷 - RGB10A2, 엔진에 따라 다름)
    g. 셰이딩 모델 ID / 머티리얼 ID: 그려진 오브젝트/머티리얼의 고유 식별자, 기본적으로 오브젝트의 분할 마스크 - 일반적인 사용 사례는 가장 가까운 머티리얼 식별자가 현재와 다른 경우 누적하지 않는 것입니다(지원 형식 - 8비트 또는 16비트 정수, 엔진에 따라 다름).
2. HDR 톤매퍼 유형: 문자열, 라인하드, 원오버루마 또는 ACES
3. 3D 모션 벡터 - (지원 형식 - 16비트 또는 32비트 부동 소수점)
4. 파티클 마스크: 기본적으로 베이스 패스의 일부로 그려지지 않은 파티클이 포함된 픽셀을 식별합니다(지원되는 형식 - 8비트 정수).
5. 애니메이션 텍스처 마스크: 애니메이션 텍스처가 차지하는 픽셀을 덮는 바이너리 마스크(지원 형식 - 8비트 정수)
6. 고해상도 깊이: (지원 포맷 - D24S8)
7. 보기 공간 위치: (지원 형식 - 16비트 또는 32비트 부동 소수점)
8. 프레임 시간 델타(밀리초 단위): 이 델타에 의해 결정되는 모션 벡터 크기와 fps에서 오브젝트의 속도에 따라 노이즈 제거 또는 안티 에일리어스 양을 결정하는 데 도움이 됩니다.
9. 레이 트레이싱 적중 거리: 각 효과에 대해 - 레이 트레이스된 색상의 노이즈 양에 대한 좋은 근사치(지원 형식 - 16비트 또는 32비트 부동 소수점)

10. 반사를 위한 모션 벡터: 미러링된 표면과 같이 반사된 오브젝트의 모션 벡터(지원 형식 - 16 비트 또는 32비트 부동 소수점)

9.5 공지사항

본 사양에 제공된 정보는 제공된 날짜를 기준으로 정확하고 신뢰할 수 있는 것으로 간주됩니다. 그러나 NVIDIA Corporation("NVIDIA")은 그러한 정보의 정확성 또는 완전성에 대해 명시적 또는 묵시적으로 어떠한 진술이나 보증도 제공하지 않습니다. NVIDIA는 그러한 정보의 결과 또는 사용 또는 그 사용으로 인해 발생할 수 있는 제3자의 특허 또는 기타 권리 침해에 대해 어떠한 책임도 지지 않습니다. 본 문서는 이전에 제공되었을 수 있는 제품에 대한 다른 모든 사양을 대체하고 대체합니다.

NVIDIA는 언제든지 본 사양을 수정, 수정, 향상, 개선 및 기타 변경을 하거나 통지 없이 제품 또는 서비스를 중단할 수 있는 권리를 보유합니다. 고객은 주문하기 전에 최신 관련 사양을 입수해야 하며 해당 정보가 최신이고 완전한지 확인해야 합니다.

NVIDIA 제품은 NVIDIA와 고객의 공인 대리인이 서명한 개별 판매 계약에서 달리 합의하지 않는 한 주문 승인 시점에 제공된 NVIDIA 표준 판매 약관에 따라 판매됩니다. 이에 따라 NVIDIA는 본 사양에 언급된 NVIDIA 제품 구매와 관련하여 고객 일반 약관을 적용하는 것에 명시적으로 반대합니다.

NVIDIA 제품은 의료, 군사, 항공기, 우주 또는 생명 유지 장비 또는 NVIDIA 제품의 고장 또는 오작동으로 인해 부상, 사망 또는 재산 또는 환경 피해가 합리적으로 예상되는 애플리케이션에 사용하기에 적합하도록 설계, 승인 또는 보증되지 않았습니다. NVIDIA는 이러한 장비 또는 애플리케이션에 NVIDIA 제품을 포함 및/또는 사용하는 것에 대해 어떠한 책임도 지지 않으며, 따라서 그러한 포함 및/또는 사용에 따른 위험은 고객이 스스로 감수해야 합니다.

NVIDIA는 이러한 사양에 기반한 제품이 추가 테스트나 수정 없이 특정 용도에 적합하다는 것을 진술하거나 보증하지 않습니다. 각 제품의 모든 매개변수에 대한 테스트는 NVIDIA에서 반드시 수행하는 것은 아닙니다. 제품이 고객이 계획한 애플리케이션에 적합하고 적합한지 확인하고 애플리케이션 또는 제품의 결함을 피하기 위해 애플리케이션에 필요한 테스트를 수행하는 것은 전적으로 고객의 책임입니다. 고객 제품 설계의 약점은 NVIDIA 제품의 품질 및 신뢰성에 영향을 미칠 수 있으며 본 사양에 포함된 것 이외의 추가 또는 다른 조건 및/또는 요구 사항을 초래할 수 있습니다. NVIDIA는 이에 근거하거나 이에 기인할 수 있는 모든 불이행, 손상, 비용 또는 문제와 관련된 어떠한 책임도 지지 않습니다: (i) 본 사양에 위배되는 방식으로 NVIDIA 제품을 사용하는 경우, 또는
(ii) 고객 제품 디자인.

본 사양에 따라 명시적이든 묵시적이든 어떠한 라이선스도 NVIDIA 특허권, 저작권 또는 기타 NVIDIA 지적 재산권에 따라 부여되지 않습니다. 타사 제품 또는 서비스와 관련하여 NVIDIA가 게시한 정보는 해당 제품 또는 서비스를 사용할 수 있는 NVIDIA의 라이선스나 그에 대한 보증 또는 보증을 구성하지 않습니다. 이러한 정보를 사용하려면 제3자의 특허 또는 기타 지적 재산권에 따라 제3자로부터 라이선스를 받거나 NVIDIA의 특허 또는 기타 지적 재산권에 따라 NVIDIA로부터 라이선스를 받아야 할 수 있습니다. 본 사양의 정보 복제는 NVIDIA가 서면으로 승인하고, 변경 없이 복제하며, 모든 관련 조건, 제한 사항 및 고지를 동반하는 경우에만 복제가 허용됩니다.

모든 NVIDIA 설계 사양, 레퍼런스 보드, 파일, 도면, 진단, 목록 및 기타 문서(함께 또는 개별적으로, "자료")

는 "있는 그대로" 제공되고 있습니다. NVIDIA는 자료와 관련하여 명시적, 묵시적, 법적 또는 기타 어떠한 보증도 하지 않으며, 비침해성, 상품성 및 특정 목적에의 적합성에 대한 모든 묵시적 보증을 명시적으로 부인합니다.

어떠한 이유로든 고객에게 발생할 수 있는 손해에도 불구하고, 본 문서에 설명된 제품에 대한 NVIDIA의 고객에 대한 총 책임 및 누적 책임은 해당 제품에 대한 NVIDIA 판매 약관에 따라 제한됩니다.

9.5.1 상표

NVIDIA 및 NVIDIA 로고는 다음 국가에서 NVIDIA Corporation의 상표 및/또는 등록 상표입니다.
미국 및 기타 국가. 기타 회사 및 제품 이름은 해당 회사 및 관련 회사의 상표일 수 있습니다.

9.5.2 저작권

9.6 타사 소프트웨어

9.6.1 CURL

저작권 및 사용권 고지

9.6.2 8x13 비트맵 글꼴

https://courses.cs.washington.edu/courses/cse457/98a/tech/OpenGL/font.c

```
*  본 소프트웨어에 구현된 자료는 "있는 그대로" 귀하에게 제공됩니다.
*  명시적, 묵시적 또는 기타 어떠한 종류의 보증도 제공하지 않습니다,
*  상품성에 대한 보증을 포함하되 이에 국한되지 않고
*  특정 목적에 대한 적합성.  어떠한 경우에도 실리콘
*  GRAPHICS, INC.  는 귀하 또는 다른 누구에게도 직접적인 책임을 지지 않습니다,
*  특별, 우발적, 간접적 또는 결과적 손해에 대해
*  종류 또는 모든 손해를 포함하되 이에 국한되지 않는 모든 손해에 대해 책임을 지지 않습니다,
*  이익 손실, 사용 손실, 저축 또는 수익 손실 또는 다음의 청구
*  제3자, 실리콘 그래픽스, INC. HAS BEEN
*  그러한 손실의 가능성에 대해 알려드립니다.
*  로 인해 또는 이와 관련하여 발생하는 모든 책임 이론
*  이 소프트웨어의 소유, 사용 또는 성능.
```

9.6.3 d3dx12.h

https://github.com/microsoft/DirectX-Graphics-Samples/blob/master/Samples/Desktop/D3D12Multithreading/src/d3dx12.h

```
*       저작권 © 2017 NVIDIA Corporation.  모든 권리 보유.
*
*   사용자에게 통지합니다:
*
*   이 소프트웨어는 미국 및 국제 저작권법에 따라 NVIDIA 소유권의 적용을 받습니다.
*
*   이 소프트웨어 및 여기에 포함된 정보는 NVIDIA의 독점적이며 기밀입니다.
*   NVIDIA 소프트웨어 라이선스 계약의 조건에 따라서만 제공됩니다.
*   및/또는 비공개 계약에 동의해야 합니다.  그렇지 않으면 귀하는 어떤 방식으로든 이 소프트웨어를 사용하거나 액세스할 권한이 없습니다.
*
*   해당 NVIDIA 소프트웨어 라이선스 계약이 적용되지 않는 경우:
*   NVIDIA는 이 소프트웨어의 어떤 목적에 대한 적합성에 대해서도 어떠한 진술도 하지 않습니다.
*   어떠한 종류의 명시적 또는 묵시적 보증 없이 "있는 그대로" 제공됩니다.
*   엔비디아는 이 소프트웨어와 관련된 모든 보증을 부인합니다,
*   상품성, 비침해성 및 특정 목적에의 적합성에 대한 모든 묵시적 보증을 포함합니다.
*   어떠한 경우에도 NVIDIA는 특별, 간접, 부수적 또는 결과적 손해에 대해 책임을 지지 않습니다,
*   또는 사용, 데이터 또는 이익의 손실로 인해 발생하는 모든 손해는 계약에 관계없이 배상하지 않습니다,
*   이 소스 코드의 사용 또는 성능으로 인해 또는 이와 관련하여 발생하는 과실 또는 기타 불법 행위.
*
*   미국 정부 최종 사용자.
*   이 소프트웨어는 48 C.F.R. 2.101(1995년 10월)에 정의된 용어인 "상업용 품목"에 해당합니다,
*   "상업용 컴퓨터 소프트웨어"와 "상업용 컴퓨터 소프트웨어 문서"로 구성됩니다.
*   이러한 용어는 48 C.F.R. 12.212(1995년 9월) 에서 사용되며 미국 정부에만 상업용 최종 품목으로 제공됩니다.
*   48 C.F.R. 12.212 및 48 C.F.R. 227.7202-1 ~ 227.7202-4(1995년 6월)와 일치합니다,
*   모든 미국 정부 최종 사용자는 본 계약에 명시된 권리만을 가지고 소프트웨어를 취득합니다.
*
*   개인 및 상업용 소프트웨어에서 이 소프트웨어를 사용하려면 반드시 다음 사항을 포함해야 합니다,
*   를 사용자 문서와 코드에 대한 내부 주석에 추가하세요,
*   위의 면책 조항( 해당되는 경우) 및 미국 정부 최종 사용자 고지를 준수해야 합니다.
*

*  Microsoft Corporation의 원본 코드는 다음에서 가져왔습니다.
*    https://github.com/Microsoft/DirectX-Graphics-Samples/tree/master/Samples/Desktop/D3D12Multithreading
*    MIT 라이선스(아래)에 따라 라이선스가 부여되었습니다.
*    원본 코드와 관련된 변경 사항에는 NVIDIA 라이선스가 적용됩니다.

*  저작권 (c) Microsoft. 모든 권리 보유.
*  이 코드는 MIT 라이선스(MIT)에 따라 라이선스가 부여됩니다.
*  이 코드는 다음 사항에 대한 보증 없이 *있는 그대로* 제공됩니다.
*  명시적이든 묵시적이든 모든 종류를 포함하여
*  특정 제품에 대한 적합성에 대한 묵시적 보증
*  목적, 상품성 또는 비침해성.
```

9.6.4 xml

https://pugixml.org/license.html

```
* 이 라이브러리는 MIT 라이선스 조건에 따라 누구나 무료로 사용할 수 있습니다:
```

```
* 저작권 (c) 2006-2018 아르세니 카풀키네
* 이에 따라 이 문서의 사본을 얻는 모든 사람에게 무료로 사용할 수 있는 권한이 부여됩니다.
* 소프트웨어 및 관련 문서 파일("소프트웨어")을 취급할 수 있습니다.
* 사용, 복사, 수정, 병합할 수 있는 권리를 포함하되 이에 국한되지 않는 모든 권한을 제한 없이 부여합니다,
* 소프트웨어의 사본을 게시, 배포, 재라이선스 및/또는 판매하고 다른 사람에게 허용하는 행위
* 다음 조건에 따라 소프트웨어가 제공되는 대상에게 제공됩니다:
* 위의 저작권 고지 및 본 사용권 고지는 모든 복사본 또는
* 소프트웨어의 상당 부분을 차지합니다.
* 소프트웨어는 명시적이든 묵시적이든 어떠한 종류의 보증도 없이 "있는 그대로" 제공됩니다,
* 상품성 보증, 특정 상품에 대한 적합성 보증을 포함하되 이에 국한되지 않습니다.
* 목적 및 비침해. 어떠한 경우에도 저자 또는 저작권 소유자는 책임을 지지 않습니다.
* 계약, 불법행위 또는 기타 모든 청구, 손해배상 또는 기타 책임에 대하여
* 그렇지 않은 경우, 소프트웨어 또는 소프트웨어의 사용 또는 기타
* 소프트웨어에서 거래합니다.
* 즉, 오픈소스 및 애플리케이션에서 pugixml을 자유롭게 사용할 수 있습니다.
* 독점. 제품에서 pugixml을 사용하는 경우 다음과 같은 확인을 추가하는 것으로 충분합니다.
* 이와 같이 제품 배포에 대한 정보를 제공합니다:
* 이 소프트웨어는 pugixml 라이브러리(http://pugixml.org)를 기반으로 합니다. pugixml

은 저작권 (C) * 2006-2018 Arseny Kapoulkine입니다.
```

9.6.5 npy

https://github.com/llohse/libnpy

https://github.com/llohse/libnpy/blob/master/LICENSE

```
* MIT 라이선스

* 저작권 (c) 2021 Leon Merten Lohse

* 이에 따라 사본을 얻는 모든 사람에게 무료로 사용 권한이 부여됩니다.
* 본 소프트웨어 및 관련 문서 파일("소프트웨어")을 처리하기 위해
* 권리를 포함하되 이에 국한되지 않는 소프트웨어의 권리
* 사용, 복사, 수정, 병합, 게시, 배포, 재라이선스 및/또는 판매할 수 없습니다.
* 소프트웨어의 사본, 그리고 소프트웨어가 다음과 같은 사람에게 허용되는 경우
* 다음 조건에 따라 제공될 수 있습니다:

* 위의 저작권 고지 및 본 사용 허가 고지는 모든
* 소프트웨어의 사본 또는 상당 부분을 복사할 수 없습니다.

* 소프트웨어는 어떠한 종류의 명시적 또는 묵시적 보증 없이 "있는 그대로" 제공됩니다.
* 상품성에 대한 보증을 포함하되 이에 국한되지 않는 묵시적 보증을 포함합니다,
* 특정 목적에의 적합성 및 비침해성. 어떠한 경우에도
* 저자 또는 저작권 소유자는 모든 청구, 손해 또는 기타에 대해 책임을 집니다.
* 계약, 불법 행위 또는 기타 행위로 인해 발생하는 책임,
* 소프트웨어 또는 소프트웨어의 사용 또는 기타 거래와 관련하여 또는 이와 관련하여
* 소프트웨어.
```

9.6.6 stb

https://github.com/nothings/stb/blob/master/LICENSE

```
/*
* 대안 A - MIT 라이선스
* 저작권 (c) 2017 Sean Barrett
* 이에 따라 다음 사본을 얻는 모든 사람에게 무료로 사용할 수 있는 권한이 부여됩니다.
* 본 소프트웨어 및 관련 문서 파일("소프트웨어")을 취급하기 위해 다음을 수행합니다.
```

```
* 다음과 같은 권리를 포함하되 이에 국한되지 않고 제한 없이 소프트웨어를 사용할 수 있습니다.
* 사본의 사용, 복사, 수정, 병합, 게시, 배포, 재라이선스 및/또는 판매
* 소프트웨어의 사용 및 소프트웨어가 제공된 사람이 다음을 수행하도록 허용합니다.
* 따라서 다음 조건이 적용됩니다:
* 위의 저작권 고지 및 본 사용 허가 고지는 모든
* 소프트웨어의 사본 또는 상당 부분을 복사할 수 없습니다.
* 소프트웨어는 어떠한 종류의 명시적 또는 묵시적 보증 없이 "있는 그대로" 제공됩니다.
* 상품성에 대한 보증을 포함하되 이에 국한되지 않는 묵시적 보증을 포함합니다,
* 특정 목적에의 적합성 및 비침해성. 어떠한 경우에도
* 저자 또는 저작권 소유자는 모든 청구, 손해 또는 기타에 대해 책임을 집니다.
* 계약, 불법 행위 또는 기타 행위로 인해 발생하는 책임,
* 소프트웨어 또는 소프트웨어의 사용 또는 기타 거래와 관련하여 또는 이와 관련하여
* 소프트웨어.
*/
```

9.6.7 DirectX-그래픽-샘플

https://github.com/Microsoft/DirectX-Graphics- 샘플/트리/마스터/샘플/데스크톱/D3D12멀티스레딩

```
/*
* 저작권 (c) Microsoft. 모든 권리 보유.
* 이 코드는 MIT 라이선스(MIT)에 따라 라이선스가 부여됩니다.
* 이 코드는 다음 사항에 대한 보증 없이 *있는 그대로* 제공됩니다.
* 명시적이든 묵시적이든 모든 종류를 포함하여
* 특정 제품에 대한 적합성에 대한 묵시적 보증
* 목적, 상품성 또는 비침해성.
*/
```

9.7 Linux 드라이버 호환성

Linux 드라이버는 더 이상 `NVSDK_NGX_Parameter_SuperSampling_MinDriverVersionMajor`로 쿼리할 수 있는 엄격한 최소 드라이버 버전 호환성 모델을 준수하지 않습니다.

2022년 3월 22일부터 릴리스되는 모든 NVIDIA Linux 드라이버(예: NVIDIA Linux 드라이버 510.60.02)는 DLSS 스니펫 버전 >= 2.4.0만 지원합니다.

해당 날짜 이전에 릴리스된 모든 NVIDIA Linux 드라이버는 2.4.0 미만의 DLSS 스니펫 버전만 지원합니다. 애플리케이션은 두 개의 스니펫(2.4.0 미만 1개, 2.4.0 이상 1개)을 함께 제공함으로써 이전 드라이버와 새 드라이버를 모두 지원할 수 있습니다.

>= 2.4.0). 이 경우 애플리케이션은 `NVSDK_NGX_PathListInfo`에 두 스니펫의 경로가 모두 있어야 합니다(먼저 - 이전 스니펫, 그다음 - 최신 스니펫). 드라이버는 이러한 코드조각에 순서대로 액세스하여 로드할 수 있는 첫 번째 코드조각을 사용합니다.

Windows 버전의 DLSS 스니펫(Steam 플레이 프로톤에서 실행되는 스니펫 포함)은 이 제한의 영향을 받지 않습니다.