DLSS Frame Generation SDK

Version 310.4.0

August 2025

Table of Contents

Introduction

NVIDIA DLSS Frame Generation (DLSS-FG) is an AI-based technology designed to increase framerates by generating high-quality frames in real-time. Utilizing a deep-learning algorithm, DLSS-FG uses data provided by the game engine to generate an intermediate frame between two traditionally rendered frames. This process aims to achieve higher frame rates and improve the smoothness of motion.

DLSS Multi-Frame Generation (DLSS-MFG) extends this technology, enabling the generation of up to three intermediate frames for each traditionally rendered frame to increase framerates by up to 4x.

Purpose

This documentation is intended for developers who wish to integrate NVIDIA DLSS Frame Generation into their existing applications. It serves as the complete technical resource for integrating and troubleshooting DLSS-FG using the NGX SDK.

This documentation is divided into three sections:

- **Integration Guide**: A step-by-step guide to integrating DLSS-FG into an application.
- **Troubleshooting and Optional Features**: A collection of guides for using advanced features of DLSS-FG.
- **API Reference**: Detailed breakdown of functions, structures, etc.

DLSS-FG can also be integrated via the **NVIDIA Streamline SDK**.

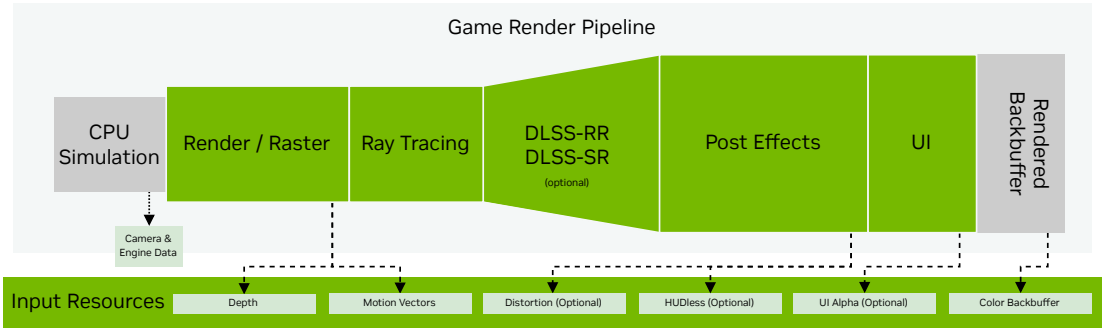
System Requirements

- **GPU**: An NVIDIA GeForce RTX™ 40-series GPU or higher is required for DLSS Frame Generation. DLSS Multi-Frame Generation requires an NVIDIA GeForce RTX 50-series GPU or equivalent.
- **Operating System**: Windows 10 20H1 (build 19041) or higher.
- **Drivers**: NVIDIA Game Ready or Studio Drivers, version 520 or higher.

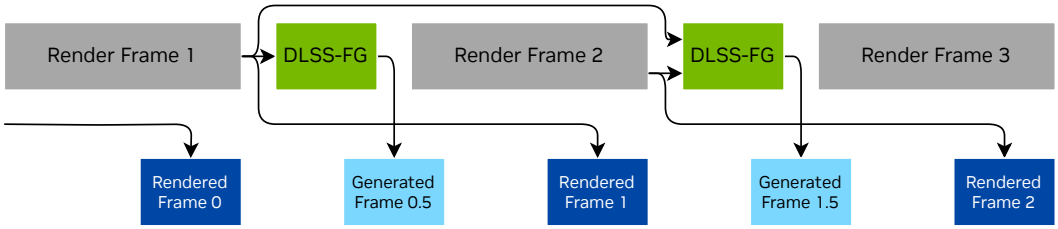
DLSS Frame Generation Overview

The DLSS-FG algorithm leverages deep learning to generate intermediate frames based on consecutive pairs of rendered frames. Instead of relying solely on traditional rendering to produce every frame, DLSS-FG analyzes the motion and changes between the input frames to synthesize new frames. This process operates as a post-processing step on the GPU, running after the application has completed its rendering pipeline for a given frame.

DLSS Frame Generation is designed to improve performance by generating frames faster than the game engine can render a full frame. This is particularly effective in CPU-bound scenarios, where the algorithm can leverage spare GPU cycles to generate frames. In GPU-bound applications, the actual frame rate multiplier may be slightly lower as the AI algorithm competes for GPU resources.



The core inputs to the DLSS-FG algorithm are the color buffers of the current and previous frames. These are augmented by engine data provided by the application, including the corresponding depth buffers, per-pixel motion vectors (describing the screen-space displacement of each pixel from the current to the previous frame, for motion tracking), camera transformation matrices (providing information about the camera's movement between frames), and optional UI hint data. The frame generation algorithm combines the motion derived from the game engine data with predicted optical flow information to generate a predicted intermediate frame that occurs temporally between the two rendered input frames. This frame then needs to be sequenced with the traditionally rendered frames for display, which introduces certain timing considerations.



The above diagram shows the necessary ordering of the generated and interpolated frames for a given pair of rendered frames, frame 1 and frame 2. Because DLSS-FG is an interpolation process, the algorithm requires both frame 1 and 2 as input to produce the intermediate frame 1.5. However, the frames must be sequenced in the order they were intended to occur in time, with frame 1.5 falling in between frames 1 and 2. To ensure a smooth visual output, these frames must be presented with roughly equal temporal spacing. This occurs asynchronously from the main render thread.

Maintaining the correct sequence and spacing inherently introduces a degree of latency. In a traditional rendering pipeline without frame generation, frame 2 would be rendered and presented immediately. With DLSS-FG, however, the presentation of frame 2 must be delayed by the time taken to generate and present frame 1.5. This means that frame 2 is essentially held back to allow for the insertion and display of the interpolated frame in the correct temporal order. To minimize the impact of the added latency, DLSS-FG works with NVIDIA Reflex to maintain a responsive experience.

When Multi-Frame Generation is enabled, the deep learning algorithm generates up to three intermediate frames between the two rendered frames. The pacer then paces all four frames (three generated frames followed by one rendered frame) to effectively quadruple the output frame rate. Because the DLSS-MFG algorithm is able to leverage information from the first generated frame to generate the subsequent frames, these additional frames are much faster to generate, and the resulting additional latency is minimal.

Performance

The performance characteristics of DLSS-FG, including execution time and memory usage, depend on the specific game engine and integration details. To help developers understand typical performance metrics, NVIDIA has benchmarked DLSS-FG across multiple games using different NVIDIA GeForce RTX graphics cards. The following measurements provide representative data for execution times and resource utilization that developers can use to evaluate potential performance benefits.

Execution Time

GeForce SKU	Resolution	2x Cost (ms)	4x Cost (ms)
RTX 4090	1080p	1.35	-
RTX 4090	1440p	1.78	-
RTX 4090	4k	2.77	-
RTX 5080	1080p	1.45	2.43
RTX 5080	1440p	2.04	3.61
RTX 5080	4k	2.84	5.25
RTX 5090	1080p	1.07	1.78
RTX 5090	1440p	1.46	2.48
RTX 5090	4k	1.72	3.32

GPU Memory Utilization

The amount of VRAM used by DLSS-FG does not change significantly between GPUs or between single-frame and multi-frame generation.

Resolution	VRAM Usage Estimate (MB)
1080p	218
1440p	384
4k	598

Integration Guide

Introduction

Welcome to the DLSS Frame Generation (DLSS-FG) Integration Guide. This document is designed to help game developers integrate NVIDIA's DLSS Frame Generation technology into their applications.

What This Guide Covers

This guide covers the steps necessary to integrate DLSS-FG into an existing application. It will show you how to:

1. **Initialize the NGX SDK**: Set up the necessary components to use DLSS-FG and properly release resources when done.
2. **Check Feature Support**: Determine if DLSS-FG and DLSS-MFG are supported on the end-user's system.
3. **Manage the Feature Lifecycle**: Create DLSS-FG feature instances, and release them when done.
4. **Evaluate the DLSS-FG Feature**: Execute the DLSS-FG algorithm to generate interpolated frames.

Prerequisites

This guide assumes you have the following:

- **Existing Application**: A game or other graphics application that uses D3D12 or Vulkan.
- **Basic Understanding of Graphics Programming**: Familiarity with concepts such as rendering pipelines, command lists, and resource management.
- **NVIDIA RTX 40-series GPU or Higher**: Required for DLSS Frame Generation. DLSS Multi-Frame Generation requires an NVIDIA RTX 50-series GPU.

Guide Structure

This guide is organized into the following sections:

1. **Getting Started**: Initial setup and configuration of the NGX SDK.
2. **Initialization and Shutdown**: Detailed steps to initialize and shut down the NGX SDK.
3. **Using NGX Parameter Maps**: Instructions to allocate parameter maps, used to provide parameters to the feature.
4. **Checking Feature Support**: Methods to check if DLSS-FG and DLSS-MFG are supported on the target system.
5. **Feature Lifecycle**: Instructions on creating, managing, and destroying DLSS-FG feature instances.
6. **Evaluating the Feature**: Steps to execute the DLSS-FG algorithm and generate interpolated frames.
7. **Presenting Generated Frames**: Details on timing and pacing generated frames to the display.

By following this guide, you will be able to integrate DLSS Frame Generation into your application, providing higher frame rates and smoother motion for your users.

1. Getting Started

To integrate DLSS Frame Generation (DLSS-FG) using the NGX SDK, you need to set up the necessary files. If you haven't downloaded it already, visit the NVIDIA Developer website and download the latest version of the NGX SDK.

NGX DLSS SDK Download

Adding the NGX SDK to a Project

Project Requirements

The NGX SDK requires the following build dependencies:

- **Windows**: Microsoft Visual Studio 2012 or newer
- **Linux**: glibc 2.11 or newer

In addition, your application must use one of DirectX 11, DirectX 12, Vulkan (version 1.1 or higher), or CUDA.

Header Files

Configure your application to include the `include` directory in your project's include path. The NGX SDK contains four types of headers:

- `nvsdk_ngx.h` contains the main entry-point functions for the SDK.
- `nvsdk_ngx_defs.h` defines common structs, constants, and parameter names.
- `nvsdk_ngx_params.h` defines the `NVSDK_NGX_Parameter` struct, which is used for providing parameters to the feature, as well as common parameter structs used by some helper functions.
- `nvsdk_ngx_helpers.h` contains helper functions to simplify certain common operations, like creating or evaluating a feature. Helpers are API-specific: Vulkan helpers can be found in headers named `*_vk.h`, and Direct3D helpers can be found in headers without `_vk`.

For DLSS-FG, many of the required functions and definitions can be found in headers with the `_dlssg` suffix.

Static Libraries

Link your application against the appropriate static library. Static libraries are found in the `lib` directory, and the correct library to use depends on your project's configuration:

Windows

Static libraries for Windows can be found in `lib/Windows_x86_64/x64` (or in `lib/Windows_x86_64/vs20XX` for projects using older versions of Visual Studio). Use:

- `nvsdk_ngx_s.lib` if the project uses static runtime linking (`/MT`).
- `nvsdk_ngx_d.lib` if the project uses dynamic runtime linking (`/MD`).

Linux

Linux projects must link against `lib/Linux_x86_64/libsdk_nvngx.a`, `libstdc++.so`, and `libdl.so`.

Feature Dynamic Libraries

The actual code for NGX features is contained in dynamic linked libraries. These can also be found in the `lib` directory. Place the appropriate library files in the game directory containing the main game executable. Alternatively, you can place these files in another directory that ships with your game, as long as you specify this path when you initialize the SDK. This ensures the NGX runtime can find and load the libraries.

The NGX SDK provides release binaries (in `rel` directories) to be packaged with released versions of the application, as well as development binaries (`dev` directories) for use during development.

For DLSS-FG, the required library is:

- On Windows: `nvngx_dlssg.dll`
- On Linux: `libnvidia-ngx-dlssg.so.<ver>`

Debugging Utilities

The NGX SDK provides several features to assist with debugging. On Windows, these features are controlled by Windows registry keys, which can be set using the `.reg` files in the `utils` directory. On Linux, these features are controlled using environment variables.

Additional information on debugging tools can be found in **Debugging Tools**

On-Screen Debugging Information

To enable on-screen debugging text:

- On Windows, use `ngx_driver_onscreenindicator.reg`, or set `HKEY_LOCAL_MACHINE\SOFTWARE\NVIDIA Corporation\Global\NGXCore\ShowDlssIndicator` (DWORD) to 1.
- On Linux, set the environment variable `__NGX_SHOW_INDICATOR=1`.

When debugging text is enabled, the NGX feature will draw additional text to the screen, which you can use to verify that the feature has been enabled properly and see additional debugging information.

Note: Setting `ShowDlssIndicator` to 1 enables the indicator *only in development builds* of your application. This helps ensure the indicator doesn't unexpectedly appear in other applications on your system. If you need to verify DLSS-FG behavior in a release build, set `ShowDlssIndicator` to **1024** to force the indicator to show in all builds.

Logging

You can configure NGX features to log errors and information to the console and to a file.

To enable NGX logging, set the desired logging level, which can be 0 (off), 1 (normal), or 2 (verbose):

- On Windows, use `ngx_log_on.reg` or `ngx_log_verbose.reg`, or set `HKEY_LOCAL_MACHINE\SOFTWARE\NVIDIA Corporation\Global\NGXCore\LogLevel` (DWORD) to the desired level.
- On Linux, set the environment variable `__NGX_LOG_LEVEL` to the desired level.

General Usage

- **API-Specific Functions**: The NGX SDK uses a similar set of functions for supported APIs (D3D12, Vulkan, etc.). When possible, this document refers to NGX SDK functions using API-agnostic names (for example, `NVSDK_NGX_Init` instead of `NVSDK_NGX_D3D12_Init`). Ensure that the function used matches the API in use by the game or rendering engine.
- **Thread Safety**: NGX APIs are not thread-safe. Ensure that all NGX API calls are made from a single thread or are properly synchronized to avoid race conditions and undefined behavior.
- **Command Lists and Contexts**: NGX APIs do not guarantee the preservation of the state of command lists or buffers. Manage the state as needed to ensure that the rendering pipeline remains consistent.
- **Error Handling**: Handle any errors that may occur during NGX API calls. Check the return values of the functions and implement appropriate error handling logic. Check the NGX logs for additional information about errors.

2. Initialization and Shutdown

DLSS Frame Generation (DLSS-FG) is a feature of NGX, which is part of the NVIDIA Display Driver. To use NGX and its features, you first need to initialize it.

Initialization

To initialize the NGX SDK, you must provide an identifier for your application. There are two types of identifiers:

- **NVIDIA-Provided App ID**: If NVIDIA has provided an identifier for your application, use **`NVSDK_NGX_Init`** to initialize the NGX SDK.
- **Project ID**: If no identifier has been provided, you can supply your project's identifier or your own custom identifier by using **`NVSDK_NGX_Init_with_ProjectID`** instead.

In addition to the application identifier or project ID, initialization requires:

- An application data path, which is a path to a directory where logs and other temporary data can be stored. Typically, this would be next to the application executable, or in a location like Documents or ProgramData.
- The device used by the engine, either an `ID3D12Device*` for D3D12, or `VkInstance`, `VkPhysicalDevice`, and `VkDevice` pointers for Vulkan.

You may also provide a **`NVSDK_NGX_FeatureCommonInfo`** struct to specify an alternate search path for feature dynamic libraries, or to set logging settings.

Project ID

The Project ID refers to a unique identifier specific to third-party engines such as Unreal Engine or Omniverse. The integration in these engines should handle passing the value to DLSS-FG. Ensure the project ID is set in the engine's editor.

For other engines, you can specify the `NVSDK_NGX_EngineType` as `CUSTOM`. When using a custom engine, the driver will verify that the Project ID is GUID-like, for example, `a0f57b54-1daf-4934-90ae-c4035c19df04`. Make sure that your project ID matches this format.

Example (Application ID)

```
unsigned long long ApplicationID; // Application ID provided by NVIDIA
const wchar_t* AppDataPath = L"."; // Path where NGX can store data
// Device in use by the engine. For Vulkan, provide the engine's VkInstance,
// VkPhysicalDevice, and VkDevice instead
ID3D12Device* Device;

NVSDK_NGX_Result Result = NVSDK_NGX_Init(ApplicationID, AppDataPath, Device);
if (NVSDK_NGX_FAILED(Result))
{
    // Handle initialization failure
}
```

Example (Project ID)

```
const char* ProjectId = "a0f57b54-1daf-4934-90ae-c4035c19df04"; // GUID for your project
const wchar_t* AppDataPath = L"."; // Path where NGX can store data
// Device in use by the engine. For Vulkan, provide the engine's VkInstance,
// VkPhysicalDevice, and VkDevice instead
ID3D12Device* Device;

NVSDK_NGX_Result Result = NVSDK_NGX_Init_with_ProjectID(
    ProjectId, NVSDK_NGX_ENGINE_TYPE_CUSTOM, AppDataPath, Device);
if (NVSDK_NGX_FAILED(Result))
{
    // Handle initialization failure
}
```

Shutdown

When you are done using DLSS-FG (or other NGX features), you should shut down the NGX SDK to ensure that all allocated resources are released.

Use the appropriate **`NVSDK_NGX_Shutdown1`** variant for your API.

Example

```
NVSDK_NGX_Result Result = NVSDK_NGX_Shutdown1(nullptr);
if (NVSDK_NGX_FAILED(Result))
{
    // Handle shutdown failure
}
```

3. Using NGX Parameter Maps

NGX parameter maps are the primary way to configure and pass data to NGX features like DLSS-FG. Think of them as a key-value store that you populate with the parameters needed for a specific SDK function, such as **`NVSDK_NGX_CreateFeature`** or **`NVSDK_NGX_EvaluateFeature`**.

The parameter map is a flexible interface that allows you to set or get different data types, including integers, floats, and graphics API resources, using named parameters.

Lifecycle of a Parameter Map

Allocation

You must first allocate an **`NVSDK_NGX_Parameter`** map before you can use it. The NGX SDK provides two different functions for this, depending on your needs.

- **`NVSDK_NGX_GetCapabilityParameters`**: This function allocates a map that comes pre-populated with system capabilities and information about available features. You should use this function if you need to query the SDK for supported features, as there is some overhead from populating the map. This map may still be used for providing parameters to the SDK, which is a common usage pattern for single-feature applications.
- **`NVSDK_NGX_AllocateParameters`**: This function allocates a new, empty parameter map. It is the recommended function when you only need to provide parameters to the SDK and do not need to query capabilities first.

Note: For applications using multiple features simultaneously (such as DLSS-SR and DLSS-FG), or multiple instances of the same feature, we recommend creating a separate parameter map for each feature or instance. This prevents parameter conflicts and the unintended carryover of data between features.

Setting Parameters

Once a parameter map is allocated, you can populate it using the `Set()` method, which is overloaded for different data types. Each parameter is identified by a unique name (a `const char*`), which are defined as macros in `nvsdk_ngx_defs.h` or `nvsdk_ngx_defs_dlssg.h`. For the full list of parameters supported by DLSS-FG, refer to the **Parameter Reference**.

Note: The parameter map is a stateful object that is not cleared automatically by NGX functions. Once a value is set for a parameter, it persists in the map until it is explicitly overwritten or the map is cleared with `Reset()`.

Getting Parameters

You can also retrieve values from a parameter map using the `Get()` method. This is most often done when querying a map created by `NVSDK_NGX_GetCapabilityParameters`, for example, to check if a feature is supported on the end-user's system. The method returns a **`NVSDK_NGX_Result`** to indicate success or failure.

Destruction

The lifetime of a parameter map must be managed by your application. When you are finished with it, you must destroy it using **`NVSDK_NGX_DestroyParameters`**.

Important: Parameter maps created by `NVSDK_NGX_AllocateParameters` or `NVSDK_NGX_GetCapabilityParameters` must **not** be freed using `free`/`delete`.

Example

Here is a simple example of the full lifecycle of a parameter map:

```
NVSDK_NGX_Result Result;
// A parameter map for setting feature parameters
NVSDK_NGX_Parameter* ngxParameters = nullptr;

// Allocate the parameter map. We use GetCapabilityParameters since we will be
// both getting and setting values, but AllocateParameters could also be used
Result = NVSDK_NGX_GetCapabilityParameters(&ngxParameters);
if (NVSDK_NGX_FAILED(Result))
{
    // Handle error
}

// Example: getting capability parameters
int FeatureSupported = 0;
Result = ngxParameters->Get(NVSDK_NGX_Parameter_FrameGeneration_Available, &FeatureSupported);
if (NVSDK_NGX_SUCCEED(Result) && FeatureSupported)
{
    // Feature supported
}

// Example: setting parameters needed to create or evaluate the feature
ngxParameters->Set(NVSDK_NGX_DLSSG_Parameter_Width, 1920);
ngxParameters->Set(NVSDK_NGX_DLSSG_Parameter_Height, 1080);

ID3D12GraphicsCommandList* commandList,
NVSDK_NGX_Handle* ngxHandle;
Result = NVSDK_NGX_CreateFeature(
    commandList, NVSDK_NGX_Feature_FrameGeneration, ngxParameters, &ngxHandle);

// Be sure to destroy the parameter map when it is no longer needed
NVSDK_NGX_DestroyParameters(ngxParameters);
```

4. Checking Feature Support and Requirements

Before using NGX features, you should check if the feature is supported on the end-user's system. It is easiest to check for support after the NGX SDK is initialized, but the SDK also provides a way to check for support before initialization if you wish to avoid initializing NGX unnecessarily on unsupported systems.

Querying Feature Requirements Before Initializing the NGX SDK

You can use the function **`NVSDK_NGX_GetFeatureRequirements`** to determine if a feature is supported, along with the minimum requirements for the feature.

`NVSDK_NGX_GetFeatureRequirements` requires a **`NVSDK_NGX_FeatureDiscoveryInfo`**, which you must populate with the feature you wish to query requirements for, as well as the parameters that would be provided to **`NVSDK_NGX_Init`**.

The function populates a **`NVSDK_NGX_FeatureRequirement`** struct, which contains `FeatureSupported` to indicate whether or not the feature is supported, as well as information about the minimum requirements for the feature to determine the reasons a feature might not be supported.

Example

```
// Adapter in use by engine. For Vulkan, provide the VkDevice and
// VkPhysicalDevice instead.
IDXGIAdapter* Adapter = ...;
NVSDK_NGX_FeatureDiscoveryInfo DiscoveryInfo = {};
DiscoveryInfo.FeatureID = NVSDK_NGX_Feature_FrameGeneration;
// Set other members of DiscoveryInfo as they would be passed to NVSDK_NGX_Init
NVSDK_NGX_FeatureRequirement OutSupported;
NVSDK_NGX_Result Result = NVSDK_NGX_GetFeatureRequirements(Adapter, &DiscoveryInfo, &OutSupported);
if (NVSDK_NGX_SUCCEED(Result) && OutSupported.FeatureSupported ==
NVSDK_NGX_FeatureSupportResult_Supported)
{
    // Feature is supported
}
else
{
    // Feature not supported, disable feature for end user and check
    // FeatureRequirement to determine why
}
```

Make sure to implement the appropriate logic when DLSS-FG is not supported. For example, the feature should be disabled, and the UI used to enable the feature should be greyed-out.

Querying Feature Support After Initializing the NGX SDK

You can use the function **`NVSDK_NGX_GetCapabilityParameters`** to query the capabilities of the target device, including its supported features.

If DLSS-FG is supported, the parameter **`NVSDK_NGX_Parameter_FrameGeneration_Available`** will be set to 1.

Example

```
NVSDK_NGX_Parameter ngxParameters;
NVSDK_NGX_Result Result = NVSDK_NGX_GetCapabilityParameters(Device, &ngxParameters);
if (NVSDK_NGX_SUCCEED(Result))
{
    int FeatureSupported;
    Result = ngxParameters->Get(NVSDK_NGX_Parameter_FrameGeneration_Available, &FeatureSupported);
    if (NVSDK_NGX_SUCCEED(Result) && FeatureSupported)
    {
        // Feature supported
    }
    else
    {
        // Feature not supported, query NVSDK_NGX_Parameter_FrameGeneration_FeatureInitResult
        // to determine why
    }
}

// ngxParameters may be reused later, but be sure to release the parameter map
// after it's done being used
NVSDK_NGX_DestroyParameters(&ngxParameters);
```

Checking for Multi-Frame Generation Support

Some devices that support DLSS-FG may not support DLSS-MFG. To check if DLSS-MFG is supported on an end-user's system, query **`NVSDK_NGX_DLSSG_Parameter_MultiFrameCountMax`** in the capability parameter map.

This parameter indicates the maximum number of frames that can be generated between two rendered frames. For example, if `MultiFrameCountMax` is 3, then DLSS-MFG is supported with a maximum multiplier of 4x. If `MultiFrameCountMax` is less than or equal to 1, or not set at all, only single-frame generation (2x) is supported.

Example

```
// Populated using NVSDK_NGX_GetCapabilityParameters, see above
NVSDK_NGX_Parameter ngxParameters;
uint32_t MultiFrameCountMax;
NVSDK_NGX_Result Result = ngxParameters->Get(
    NVSDK_NGX_DLSSG_Parameter_MultiFrameCountMax, &MultiFrameCountMax);
if (NVSDK_NGX_SUCCEED(Result) && MultiFrameCountMax > 1)
{
    // DLSS-MFG supported
}
else
{
    // DLSS-MFG not supported, only 2x allowed
}
```

Ensure that your application's UI takes the maximum multi-frame count into account. For example, if the maximum is 3, the UI should provide options to enable 2x, 3x, or 4x.

5. Feature Lifecycle

The NGX SDK operates on the concept of "feature instances." An instance contains all of the state required to execute a particular feature. You can execute multiple instances of a feature without affecting the state of other instances.

Creating the Feature

Creating a feature will return an **`NVSDK_NGX_Handle*`** to that instance, which you must use to evaluate or release that instance.

Most applications will need a single DLSS-FG instance to run frame generation. You can create instances using the **`NVSDK_NGX_CreateFeature`** function, but it is recommended to use the helper function **`NGX_CREATE_DLSSG`** for creating DLSS-FG instances.

`NGX_CREATE_DLSSG` requires a **`NVSDK_NGX_DLSSG_Create_Params`** struct, which you must fill with the following fields:

- **Width/Height**: The resolution of the input/output color resources.
- **NativeBackbufferFormat**: The format of the input backbuffer color resource, as defined by the graphics API in use (e.g., `DXGI_FORMAT_R10G10B10A2_UNORM`).
- **RenderWidth/Height**: The resolution at which the game was initially rendered. If using an upscaler such as DLSS Super Resolution, this is the input resolution to DLSS-SR. In most cases, the depth and motion vector resources should be this resolution.
- **DynamicResolutionScaling**: Boolean `true` if dynamic resolution scaling is enabled, `false` otherwise. If `true`, RenderWidth and RenderHeight are ignored.

Creating a feature also requires a command list that is open and recording. It is the application's responsibility to execute the command list after `CreateFeature` returns.

Example

```
NVSDK_NGX_Handle* ngxHandle;
// Allocate using NVSDK_NGX_AllocateParameters, or use an existing parameter map
NVSDK_NGX_Parameter* ngxParameters;
NVSDK_NGX_DLSSG_Create_Params CreateParams = {};

CreateParams.Width = 3840;
CreateParams.Height = 2160;
CreateParams.NativeBackbufferFormat = DXGI_FORMAT_R10G10B10A2_UNORM;
CreateParams.RenderWidth = 2560;
CreateParams.RenderHeight = 1440;
NVSDK_NGX_Result Result = NGX_D3D12_CREATE_DLSSG(
    CommandList, CreationNodeMask, VisibilityNodeMask, &ngxHandle, ngxParameters, &CreateParams);
if (NVSDK_NGX_FAILED(Result))
{
    // Handle error
}
```

Destroying the Feature

If you are no longer using DLSS-FG, you can destroy feature instances using **`NVSDK_NGX_ReleaseFeature`**. Destroying an instance releases any VRAM allocated for that instance. Do not destroy a feature instance while its resources are in use, such as by a command list that is still in-flight.

Example

```
NVSDK_NGX_Handle* ngxHandle; // Allocated by NVSDK_NGX_CreateFeature
if (NVSDK_NGX_FAILED(NVSDK_NGX_ReleaseFeature(ngxHandle)))
{
    // Handle error
}
```

When to Recreate the Feature

The parameters you provide when creating a feature instance define its internal settings and memory allocation. These parameters are fixed for the lifetime of the instance. If you need to change any of these settings, you must destroy the current instance and create a new one with the updated parameters.

Common reasons to recreate the feature include:

- **Window Resizing**: If the backbuffer width and height change because the window was resized, the feature must be recreated with the new width and height.
- **Internal Resolution Changes**: If the game changes its output or rendering resolution, the feature must be recreated with the new resolution.

See Also

- **Multiple Instances/Viewports**: Details and considerations for creating multiple instances, for applications which use multiple windows, split screen, or similar features.
- **HDR (High Dynamic Range)**: Instructions on how to configure DLSS-FG for display to HDR monitors.

6. Evaluating the Feature

After creating a DLSS-FG feature instance, you can now execute the algorithm. You may use the **`NVSDK_NGX_EvaluateFeature`** function directly, but it is recommended to use the helper function **`NGX_EVALUATE_DLSSG`** for evaluating a DLSS-FG instance.

Each call to the DLSS-FG evaluate function produces a single interpolated frame. Therefore, the number of calls you make per rendered frame determines the final framerate multiplier:

- For single-frame generation (2x), you must call evaluate once per rendered frame.
- For multi-frame generation (3x or higher), you must call evaluate multiple times. The number of calls should be one less than the desired multiplier (e.g., call evaluate 3 times for a 4x multiplier).

Evaluating the feature requires an **`NVSDK_NGX_Handle*`**, assigned when the feature was created, and a command list that is open and recording. It is the application's responsibility to execute the command list after `EvaluateFeature` returns.

`NGX_EVALUATE_DLSSG` also requires two sets of parameters:

- **`NVSDK_NGX_DLSSG_Eval_Params`**, which contains API-specific resources that the algorithm will use.
- **`NVSDK_NGX_DLSSG_Opt_Eval_Params`**, which contains common parameters like camera information.

To evaluate DLSS-FG, you must provide the following input resources, by setting the appropriate fields in `NVSDK_NGX_DLSSG_Eval_Params`. All input resources are assumed to be in a state suitable for reading. For D3D12, resources must be in state `D3D12_RESOURCE_STATE_NON_PIXEL_SHADER_RESOURCE`.

Using Vulkan Resources: For Vulkan, additional metadata about the resource needs to be provided. For each input resource, create a **`NVSDK_NGX_Resource_VK`** struct with the relevant metadata for that resource.

Required Inputs

Final Color (pBackbuffer)

The input color resource. DLSS-FG will interpolate between this resource and the `pBackbuffer` from the previous frame. In most cases, this should be the same resource as would be presented if DLSS-FG were not enabled, and so it should contain any post-processing effects, UI, etc. in use by the game. This resource's resolution must match the width and height used when creating the feature.

Note on using motion blur: if motion blur is enabled, you should reduce the magnitude of the effect by half for single-frame generation, or by a factor equal to the selected multiplier for DLSS-MFG (e.g., 1/4 for 4x).

Depth (`pDepth`)

The depth of the current frame, specified in hardware/non-linear units. The depth buffer used by the engine during rasterization typically meets these requirements. Depth may be inverted/non-inverted; set the `depthInverted` flag appropriately.

The resolution of `pDepth` need not match the resolution of `pBackbuffer`. If an upscaler such as DLSS-SR is in use, this resource should be the same depth resource that was provided to the upscaler.

Motion Vectors (`pMVecs`)

This resource contains motion vectors indicating the per-pixel, screen-space motion from the current frame to the previous frame. The value at each pixel represents the distance the object at that pixel would need to move to reach its position in the previous frame, in pixels *at the resolution of the motion vector resource*.

For example, if `pMVecs` is 1920x1080, and an object moves from the top-right corner of the screen to the center, the motion vector at the center of the screen would have a value of (960, -540). The `mvecScale` property, detailed later, can be used to rescale motion vectors that are provided in a different scale.

Like `pDepth`, the resolution of `pMVecs` need not match the resolution of `pBackbuffer`. If an upscaler such as DLSS-SR is in use, this resource should be the same motion vector resource that was provided to the upscaler.

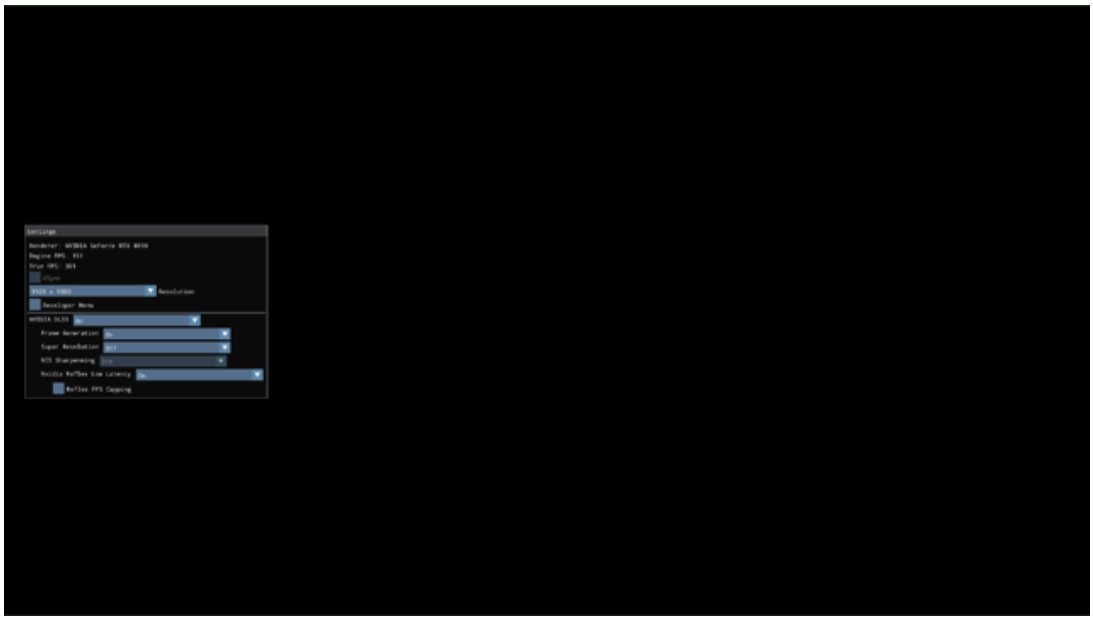
Recommended Inputs

It is strongly recommended that you provide these resources, if possible.

HUDLess (`pHudless`)

The scene color without any UI or HUD elements drawn. The value of `pHudless` must be exactly the same as the value of `pBackbuffer` for any pixel which is not UI. Therefore, any tonemapping or other post-processing effects used to render `pBackbuffer` must also be present in `pHudless`.

UI Color and Alpha (`pUI`)



A four-channel resource containing the color and opacity/alpha of the UI; may also be referred to as "UIAlpha". The color should be pre-multiplied by the alpha value, so that the following formula holds:

$$\text{Backbuffer} = \text{UI Color} + (1 - \alpha) \cdot \text{Hudless}$$

Any pixel which is not UI should have a value of 0. For best results, `pUI` must have sufficient precision in all four channels, including the alpha channel. Formats such as RGB10A2 are not appropriate for use as `pUI`.

Output Resources

You must also provide resources for DLSS-FG to write its output. All output resources are assumed to be in a state suitable for writing. For D3D12, output resources must be in state `D3D12_RESOURCE_STATE_UNORDERED_ACCESS`.

Output Interpolated Frame (`pOutputInterpFrame`)

The output resource for DLSS-FG to write the interpolated frame to. This resource must be the same size and format as `pBackbuffer`.

Output Real Frame (`pOutputRealFrame`)

Optional output resource to write the rendered ("real") frame to. Typically, the algorithm will just make a copy of `pBackbuffer`. However, DLSS-FG may draw debugging information and utilities to the output frame. Providing the real frame allows these debugging aids to be drawn every frame, instead of just to the interpolated frames (which would cause these aids to appear to flicker). You may wish to omit `pOutputRealFrame` for performance reasons in release builds, where these debugging aids would not be used.

Required Input Engine Data

The `NVSDK_NGX_DLSSG_Opt_Eval_Params` structure must be filled with the following fields:

- **reset**: Boolean indicating whether or not to reset the algorithm. Set this to `true` whenever a scene change occurs, that is, whenever the contents of the current frame are unrelated to the contents of the previous frame. When the algorithm is reset, the next interpolated frame will be a copy of the input color.
- **depthInverted**: Boolean indicating whether depth is inverted (1 is near, 0 is far) or not (0 is near, 1 is far).
- **mvecScale**: (x, y) scale factors used to rescale your input motion vector values to the required pixel-units. Common values are 1 (if the values are already in the correct units) or the input motion vector resource's width/height (if the values are in normalized units). Negative values can be used to correct for inverted axes.
- Camera matrices, each as a `float[4][4]` in row-major order (assuming post-multiplication):
  - **clipToPrevClip/prevClipToClip**: Transformation from the current frame's clip space to the previous frame's clip space, and vice-versa.
  - **cameraViewToClip/clipToCameraView**: Transformation from the current frame's camera view space to clip space, and vice-versa.
- Additional camera information:
  - **cameraPos**: Vector representing the camera's current position in world space.
  - **cameraUp/cameraRight/cameraFwd**: Unit vectors defining the orientation of the camera in 3D space, representing the upward, rightward, and forward directions relative to the camera, respectively.
  - **cameraNear/cameraFar**: The camera's near-plane and far-plane distance.
  - **cameraFOV**: The camera's field of view.
  - **cameraAspectRatio**: The camera's aspect ratio.

In addition, the following fields must be set when using DLSS-MFG:

- **multiFrameCount**: The number of intermediate frames you want to generate between each pair of rendered frames. This must be 1 for single-frame generation (2x), and a value greater than 1 for multi-frame generation. The value must not exceed `MultiFrameCountMax` (see **Checking for Multi-Frame Generation Support**).
- **multiFrameIndex**: The index (starting at 1) of the intermediate frame being generated, where 1 represents the first generated frame and `multiFrameCount` represents the last. Must be 1 for single-frame generation.

To generate `multiFrameCount` intermediate frames, you must call `EvaluateFeature` a total of `multiFrameCount` times. For each call, you must increment `multiFrameIndex` from 1 to `multiFrameCount`.

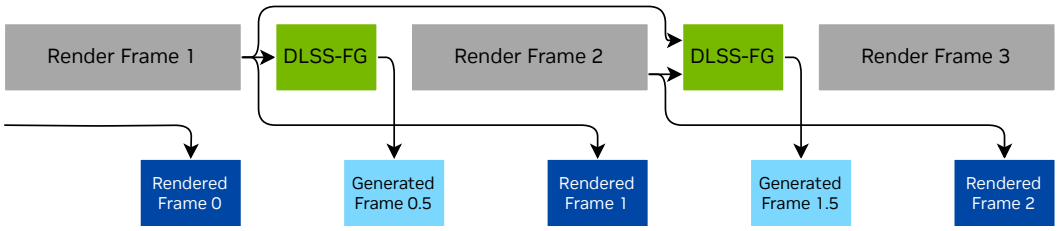
Example

```
// Allocate using NVSDK_NGX_AllocateParameters, or use an existing parameter map
NVSDK_NGX_Parameter* ngxParameters;
NVSDK_NGX_DLSSG_Eval_Params EvalParams = {};
NVSDK_NGX_DLSSG_Opt_Eval_Params OptEvalParams = {};

// Set necessary parameters for DLSS Frame Generation
EvalParams.pBackbuffer = pInColorResource;
EvalParams.pDepth = pInDepthResource;
EvalParams.pMVecs = pInMotionVectorsResource;
EvalParams.pOutputInterpFrame = pOutInterpFrameResource;
// ...

OptEvalParams.reset = reset;
OptEvalParams.depthInverted = depthInverted;
OptEvalParams.mvecScale = mvecScale;
OptEvalParams.clipToPrevClip = clipToPrevClipMatrix;
OptEvalParams.prevClipToClip = prevClipToClipMatrix;
OptEvalParams.multiFrameCount = 1;
OptEvalParams.multiFrameIndex = 1;
// ...

NVSDK_NGX_Result Result = NGX_D3D12_EVALUATE_DLSSG(
    CommandList, ngxHandle, ngxParameters, &EvalParams, &OptEvalParams);
if (NVSDK_NGX_FAILED(Result))
{
    // Handle error
}
```

7. Presenting Generated Frames

For DLSS-FG to deliver maximum smoothness and frame rate, your application must present generated frames at consistent intervals between the rendered frames. Unlike traditional rendering, where frames are presented immediately, DLSS-FG requires you to manage the timing of both rendered and generated frames. The DLSS-FG feature itself does not handle timing or presentation; this is the responsibility of your application.

The Basic Pacing Pattern



The fundamental goal is to alternate between presenting a generated frame and a rendered frame. To achieve this smooth output, you must modify your render loop to generate and then present both frames in the correct sequence and at the appropriate times.

Modifying the Render Loop

Replacing the Present

In a typical render loop, you call your graphics API's present function after a frame is rendered. When using DLSS-FG, instead of presenting the rendered frame, you pass it to the **`NVSDK_NGX_EvaluateFeature`** function to generate an intermediate frame.

Since the generated frame must be presented *before* the rendered frame, you also need to retain the rendered frame until it is ready to be presented. You can do this in two ways:

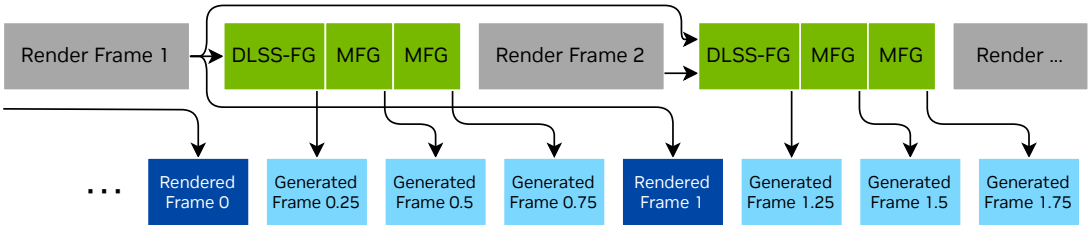
- **Using `NVSDK_NGX_Parameter_OutputReal`**: This is the recommended method. Provide an output texture to `EvaluateFeature` via the `OutputReal` parameter. The algorithm will then copy the input backbuffer to this texture. This also allows DLSS-FG to draw debugging elements in development builds.
- **Manual Retention**: You may manually retain the rendered frame resource if your rendering architecture guarantees it won't be overwritten. This can offer a slight performance advantage if it avoids a copy operation. If a copy is required, using `OutputReal` is generally simpler and incurs the same overhead.

Sequencing and Presenting Frames

After `EvaluateFeature` completes, both the generated frame and the retained rendered frame must be presented to the display using your graphics API's present function (e.g., `Swapchain::Present()`).

- Present the generated frame as soon as EvaluateFeature completes.
- Present the retained rendered frame at the correct time to achieve equal time intervals between frames. This typically requires you to present asynchronously from the main render thread.

Presenting Multiple Generated Frames (DLSS-MFG)



When DLSS Multi-Frame Generation is enabled, the pacing strategy changes slightly. The sequence involves presenting all generated frames followed by the retained rendered frame.

- After the `EvaluateFeature` call for the first generated frame completes, present this frame immediately to minimize latency. You do not need to wait for the additional generated frames to complete before presenting the first frame.
- Call `EvaluateFeature` again for each subsequent generated frame.
- Present the generated frames in the order they are generated, followed by the retained rendered frame. Aim for roughly equal temporal spacing between all presented frames (generated and rendered).

Troubleshooting and Optional Features

Introduction

This section contains guides for troubleshooting common issues that may arise when integrating DLSS Frame Generation, as well as documentation for optional features that can enhance the quality of frame generation in specific scenarios.

The guides cover topics such as:

- Debugging visual artifacts and performance issues
- Handling special cases like UI elements and transparency
- Using advanced features like motion vector hints and custom optical flow
- Optimizing DLSS-FG for different types of content and rendering techniques

Each guide provides detailed steps to identify and resolve specific issues, along with best practices and recommendations for achieving optimal results with DLSS Frame Generation.

Guide Structure

The guides in this section include:

- **Debugging Tools**: Tools and techniques for debugging DLSS-FG integration issues, including visual artifacts and performance problems.
- **Sub-Rectangle Support**: Documentation for using DLSS-FG with sub-rectangles of the frame, useful for picture-in-picture or partial frame updates.
- **Multiple Instances**: Instructions for using multiple DLSS-FG instances simultaneously, such as for split-screen rendering or multiple viewports.
- **Lens Distortion**: Best practices for handling lens distortion effects with DLSS-FG, including motion vector adjustments and artifact mitigation.
- **HDR Support**: Guide for using DLSS Frame Generation with HDR content, including color space handling and tone mapping considerations.

Debugging Tools and Hotkeys

The NGX SDK provides several features to assist with debugging. These features can help you diagnose issues, visualize intermediate results, and fine-tune the performance and quality of the DLSS-FG algorithm.

Most debugging features, with the exception of logging and limited on-screen debug information, are only available in development versions of the dynamic library. The development binaries can be found in the `dev` directories within the `lib` directory of the NGX SDK. Ensure that you are using the development binaries during development to access debugging features.

On Windows, many debugging features are controlled by Windows registry keys. The NGX SDK provides `.reg` files in the `utils` directory for enabling and disabling these features.

Logging

You can configure NGX features to log errors and information to the console and to a file.

To enable NGX logging, set the desired logging level, which can be 0 (off), 1 (normal), or 2 (verbose):

- **Windows**: Use `ngx_log_on.reg` or `ngx_log_verbose.reg`, or set `HKEY_LOCAL_MACHINE\SOFTWARE\NVIDIA Corporation\Global\NGXCore\LogLevel` (DWORD) to the desired level.
- **Linux**: Set the environment variable `__NGX_LOG_LEVEL` to the desired level.

Setting Log Paths

By default, NGX logs are generated in the application data path, or next to the application executable. To specify an alternate path for logs:

- **Windows**: Set `HKEY_LOCAL_MACHINE\SOFTWARE\NVIDIA Corporation\Global\NGXCore\EnableLogPathOverride` (DWORD) to 1, and set `HKEY_LOCAL_MACHINE\SOFTWARE\NVIDIA Corporation\Global\NGXCore\LogPath` (REG_SZ) to the directory where you want logs written.
- **Linux**: Set the environment variables `__NGX_ENABLE_OVERRIDE_LOG_PATH=1` and `__NGX_LOG_PATH_OVERRIDE=/path/to/write/logs`.

Note: The specified directory must exist prior to initialization of the NGX SDK. It will not be created automatically.

Configuring Logs from the Application

In addition to enabling logs via registry keys, you can also configure logging properties from within your application when the NGX SDK is initialized. To do so, fill the `LoggingInfo` field of the input **`NVSDK_NGX_FeatureCommonInfo`** struct when you call **`NVSDK_NGX_Init`**:

- `LoggingCallback`: An application-provided callback function that receives log messages from the SDK. The callback must be thread-safe and handle concurrent calls from multiple threads. See **`NVSDK_NGX_AppLogCallback`** for the required signature for this callback.
- `MinimumLoggingLevel`: The minimum level of logging detail to include. This setting establishes the effective minimum logging level; if it's higher than any other configured logging level (e.g., via registry key), it will override that setting.
- `DisableOtherLoggingSinks`: When set to true, this disables writing log lines to sinks other than the app log callback. This is useful when the application provides its own logging callback and wants to handle all logging itself. The

`LoggingCallback` must be non-null and point to a valid logging callback if this is set to true.

Example

```
void LogCallback(
    const char* message, NVSDK_NGX_Logging_Level loggingLevel, NVSDK_NGX_Feature sourceComponent)
{
    // Simple logging implementation example
    printf("%s\n", message);
}

// ...

unsigned long long ApplicationID;
const wchar_t* AppDataPath = L".";
ID3D12Device* Device; // Or appropriate Vulkan device handles

NVSDK_NGX_FeatureCommonInfo FeatureInfo{};

FeatureInfo.LoggingInfo.LoggingCallback = LogCallback;
FeatureInfo.LoggingInfo.MinimumLoggingLevel = NVSDK_NGX_LOGGING_LEVEL_VERBOSE;
FeatureInfo.LoggingInfo.DisableOtherLoggingSinks = true;

NVSDK_NGX_Result Result = NVSDK_NGX_Init(ApplicationID, AppDataPath, Device, &FeatureInfo);
if (NVSDK_NGX_FAILED(Result))
{
    // Handle initialization failure
}
```

On-Screen Debug Information

To enable on-screen debug information:

- **Windows**: Use `ngx_driver_onscreenindicator.reg`, or set `HKEY_LOCAL_MACHINE\SOFTWARE\NVIDIA Corporation\Global\NGXCore\ShowDlssIndicator` (DWORD) to `1`.
- **Linux**: Set the environment variable `__NGX_SHOW_INDICATOR=1`.

When debugging text is enabled, the NGX feature will draw additional text to the screen, which you can use to verify that the feature has been enabled properly and see additional debugging information.

Note: For on-screen debug information to appear in release builds, set `ShowDlssIndicator` to **1024** instead of `1`. This prevents the indicator from inadvertently appearing in other applications on developer systems.

A typical on-screen debug text printout might look like this:

```
DLSSG SDK - DO NOT DISTRIBUTE - CONTACT NVIDIA TO OBTAIN DLLs FOR YOUR TITLE
NVIDIA DLSSG v310.3.0 CL 35929230 - endpoint - Develop - DD v576.40 - D3D12 - Output 3840x2160 - MVec
1920x1080 - Hudless: Yes - UIAlpha: Yes - Distortion: Yes
USER DEBUG TEXT: <debug string>
Vis Mode (ctl+shift+v) - Stats (ctl+shift+t) - Keybinds (ctl+shift+k)
```

The components of the debug information are as follows:

- **Watermark** (`DLSSG SDK - DO NOT DISTRIBUTE - ...`): This text is rendered by any DLSS-FG dynamic library which is not intended for production. To remove this text, use a production version of the feature dynamic library.
- **Version Info** (`NVIDIA DLSSG v310.3.0 CL 35929230 - endpoint`): The version and related information about the dynamic library in use.
- **Build Type** (`Develop`): The build type of the library, either Develop or Release.
- **Display Driver** (`DD v576.40`): The version of the NVIDIA Display Driver installed on the current system.
- **Graphics API** (`D3D12`): The graphics API in use.
- **Resolution Info** (`Output 3840x2160 - MVec 1920x1080`): The resolutions of the output and motion vectors resources, respectively.
- **Optional Resources**: (`Hudless: Yes - UIAlpha: Yes`): Whether or not optional resources were provided to the algorithm.

The remaining components are shown in develop builds only:

- **User Debug Text**: (`USER DEBUG TEXT`): A string provided by the application via the **`NVSDK_NGX_DLSSG_Parameter_UserDebugText`** parameter.
- **Keybinds** (`Vis Mode (ctl+shift+v)`): Hotkeys to enable/disable various debugging features (see **Hotkeys for Debugging**)

Additional verbose information can be accessed via the Stats hotkey (default `ctrl+shift+t`). This will show detailed information about the input and output resources, including resolutions and subrect extents, as well as the input camera matrices.

Note: To draw on-screen debug information to the rendered frame, the **`NVSDK_NGX_Parameter_OutputReal`** parameter must be provided to the algorithm. If this copy of the rendered frame is not displayed, the debug information may appear to flicker.

Hotkeys for Debugging

The NGX SDK provides hotkeys that can be used to toggle various debugging features on and off during runtime. You can see a list of available keybinds by pressing the Keybinds hotkey (default `ctrl+shift+k`).

The full list of available hotkeys, and their default keybinds, is shown in the table below

Name	Default Keystroke	Description
Cycle Text Indicator	`ctrl+shift+t`	Change the verbosity level of the on-screen debug information.
Debug Visualization Mode	`ctrl+shift+v`	Enable/disable buffer visualization (see **Debug Visualization**).
Full Screen Menu Detection	`ctrl+shift+m`	Enable/disable full-screen menu detection.
Auto Scene Change Detection	`ctrl+shift+s`	Enable/disable automatic scene change detection.
Toggle Ignore Reset	`ctrl+shift+n`	Toggle whether or not the **`NVSDK_NGX_DLSSG_Parameter_Reset`** parameter from the application is ignored.

Name	Default Keystroke	Description
Toggle HUDless	`ctrl+shift+u`	Toggle whether or not the input HUDLess resource is ignored.
Toggle UI Alpha	`ctrl+shift+i`	Toggle whether or not the input UI resource is ignored.
Toggle Text Keybinds	`ctrl+shift+k`	Show/hide the list of keybinds on screen.

JSON Configuration

The NGX SDK allows for customization of certain debugging features, such as hotkey bindings, through a JSON configuration file named `ngxdlssg.json`. This file can be placed either in the same directory as the application executable or in the `%USERPROFILE%/ngx` directory.

The JSON file allows rebinding of hotkeys, which are specified in the `keybinds` object as `"Key": "Name"`, for example `"F2": "NGXDLSSG_ToggleIgnoreReset"`.

Example JSON File

```
{
  "keybinds": {
    "ctrl+shift+t": "NGXDLSSG_CycleTextIndicator",
    "ctrl+shift+m": "NGXDLSSG_ToggleFSMD",
    "ctrl+shift+s": "NGXDLSSG_ToggleSCD",
    "ctrl+shift+n": "NGXDLSSG_ToggleIgnoreReset",
    "ctrl+shift+u": "NGXDLSSG_ToggleHUDless",
    "ctrl+shift+i": "NGXDLSSG_ToggleUIAlpha",
    "ctrl+shift+k": "NGXDLSSG_ToggleTextKeybinds",
    // Debug visualization keybinds
    "ctrl+shift+v": "NGXDLSSG_ToggleVisualizeBufferMode",
    "F6": "NGXDLSSG_DecrementBufferVisualizationKey",
    "F7": "NGXDLSSG_IncrementBufferVisualizationKey",
    "ctrl+shift+f": "NGXDLSSG_ToggleVisualizationFullScreen",
    "ctrl+shift+g": "NGXDLSSG_ToggleVisualizationThumbnail"
  },
  // Ignore input distortion
  "ignore_distortion": false,
  // Ignore input hudless mode
  "ignore_hudless": false,
  // Ignore input UI alpha
  "ignore_ui": false,
  // Ignore reset flag
  "ignore_reset": false,
  // Depth value defining objects that are "near" vs "far"
  "max_depth_for_testing": 600.0,
  // Indicator level (0=None, 1=Minimal, 2=Verbose)
  "text_indicator_level": 0,
  // Enable scene change detection
  "enable_scene_change_detection": true,
  // Enable fullscreen menu detection
  "enable_fullscreen_menu_detection": false,
  // Enable asynchronous resource allocation
  "enable_async_resource_alloc": false
}
```

Debug Visualization

The DLSS-FG algorithm includes debug visualization functionality, which helps you verify that the inputs provided to the SDK are correct and free of artifacts. These visual debugging aids can be enabled and cycled through using keybinds.

Keybinds for Debug Visualization

- `ctrl+shift+v`: Toggle Visualize Buffer Mode
- `F6`: Decrement Buffer Visualization
- `F7`: Increment Buffer Visualization

Subrect Handling

What are Subrects?

A subrect (sub-rectangle) defines a specific rectangular region within a larger texture. When you use a subrect, you are instructing DLSS-FG to operate on only that portion of the texture, rather than the entire resource.

Core Principle: DLSS-FG Sees the Subrect as the Full Resource.

From the perspective of the DLSS-FG algorithm, the specified subrect *is* the full resource. DLSS-FG's internal processing operates identically on the pixels within that subrect, ignoring the rest of the texture.

Consequently, any time DLSS-FG documentation or API parameters mention "resource size" or "resolution" when subrects are in use, they are referring to the subrect's dimensions (width and height), not the dimensions of the larger underlying texture resource. The result is identical to providing a separate texture that is exactly the size of the subrect, containing only the relevant data.

Common Use Cases and Examples

- **Letterboxing or Cinematic Crops:** Use a backbuffer subrect to render generated frames into the center of the backbuffer, creating a letterboxed or cropped scene.
- **Dynamic Resolution Scaling (DRS) and Upscalers:** When implementing DRS, subrects are particularly valuable for depth and motion vector resources. Instead of reallocating these resources whenever the render resolution changes, you can allocate them once at the maximum size and then provide a subrect matching the current resolution. This eliminates expensive reallocations.
- **Split-screen Rendering:** You can use multiple DLSS-FG instances with distinct subrects on a shared backbuffer to render multiple split-screen views. For more details, see the **Multiple Instances** guide.

Defining Subrects

Subrects are defined by a `base` (the top-left coordinate, X and Y) and a `size` (width and height). You specify these parameters per resource type. DLSS-FG uses a top-left origin for coordinates, with positive X extending rightwards and positive Y extending downwards.

The NGX Parameters used to set the subrect parameters for a resource are the parameter for that resource plus one of `SubrectBaseX`, `SubrectBaseY`, `SubrectWidth`, or `SubrectHeight`. For example, the subrect parameters for the motion vector resource are:

- `NVSDK_NGX_DLSSG_Parameter_MVecsSubrectBaseX`
- `NVSDK_NGX_DLSSG_Parameter_MVecsSubrectBaseY`
- `NVSDK_NGX_DLSSG_Parameter_MVecsSubrectWidth`
- `NVSDK_NGX_DLSSG_Parameter_MVecsSubrectHeight`

You can also set subrect parameters using the relevant fields within the `NVSDK_NGX_DLSSG_Opt_Eval_Params` struct:

- `mvecsSubrectBase`, `mvecsSubrectSize`
- `depthSubrectBase`, `depthSubrectSize`
- etc.

If all of the subrect parameters for a particular resource are not provided or set to 0, DLSS-FG assumes that it should use the full size of the resource instead.

Example: Specifying Subrects

```
NVSDK_NGX_DLSSG_Eval_Params EvalParams = {};
NVSDK_NGX_DLSSG_Opt_Eval_Params OptEvalParams = {};

// Set subrect parameters
EvalParams.pBackbuffer = pInColorResource;
EvalParams.pDepth = pInDepthResource;
EvalParams.pMVecs = pInMotionVectorsResource;
EvalParams.pOutputInterpFrame = pOutInterpFrameResource;

OptEvalParams.reset = reset;
OptEvalParams.depthInverted = depthInverted;
OptEvalParams.mvecScale = mvecScale;
OptEvalParams.backbufferSubrectBase = { 0, 257 };     // x, y
OptEvalParams.backbufferSubrectSize = { 3840, 1646 }; // width, height
OptEvalParams.mvecsSubrectBase = { 0, 0 };            // x, y
OptEvalParams.mvecsSubrectSize = { 1920, 823 };       // width, height
// ...

NVSDK_NGX_Result Result = NGX_D3D12_EVALUATE_DLSSG(CommandList, ngxHandle, ngxParameters, &EvalParams,
&OptEvalParams);
if (NVSDK_NGX_FAILED(Result))
{
    // Handle error
}
```

Spatial Alignment of Subrects

All subrects you provide to DLSS-FG—for color, depth, motion vectors, and any other inputs—must be spatially aligned. This means their corners must correspond to the same geometric region in the scene. This ensures DLSS-FG's internal algorithms correctly correlate information across different input buffers.

Backbuffer Subrects

The backbuffer subrect is shared between the input and output backbuffer resources. This means that the `NVSDK_NGX_DLSSG_Parameter_BackbufferSubrectBaseX/BaseY/Width/Height` parameters define the active region for both the input backbuffer color resource **`NVSDK_NGX_DLSSG_Parameter_Backbuffer`** and the color outputs **`NVSDK_NGX_DLSSG_Parameter_OutputInterpolated`** and **`NVSDK_NGX_DLSSG_Parameter_OutputReal`** (if provided).

Remember that the backbuffer subrect width and height must always match the **`NVSDK_NGX_DLSSG_Parameter_Width/Height`** values you provided when creating the DLSS-FG feature instance. The feature's creation size directly corresponds to the *subrect dimensions* that DLSS-FG will operate on, not the full size of the underlying resource.

If the required backbuffer subrect size changes (for example, if a user resizes the application window in a way that alters the effective render area), you must release the existing DLSS-FG feature and re-create it with the new subrect dimensions.

Controlling Writes Outside the Subrect

By default, DLSS-FG fills regions outside the defined backbuffer subrect by copying color from the input backbuffer. This ensures that any parts of the output buffer not covered by the subrect do not display uninitialized or stale data.

However, there are common cases where this default behavior is undesirable. For example:

- **Multiple Subrects on a Shared Backbuffer:** If your application uses multiple DLSS-FG instances to render into a shared backbuffer (e.g., for split-screen rendering), the default fill behavior would cause a subsequent instance to overwrite the output from a previous one.
- **Performance and Cropped Outputs:** Copying pixels outside the subrect might incur a small, but potentially avoidable, performance overhead. This is especially relevant if the final output will be cropped to the subrect, making the data outside the subrect unnecessary.

To disable this default fill behavior, you must set the `NVSDK_NGX_DLSSG_EvalFlags_UpdateOnlyInsideExtents` flag (via the `evalFlags` field in `NVSDK_NGX_DLSSG_Opt_Eval_Params` or the **`NVSDK_NGX_DLSSG_Parameter_EvalFlags`** parameter). When set, this flag instructs DLSS-FG to only write within the specified backbuffer subrect, leaving all areas outside untouched.

Backbuffer Texture Size Requirements

When the `NVSDK_NGX_DLSSG_EvalFlags_UpdateOnlyInsideExtents` flag is *not* set, the underlying resource resolution of the output generated frame **`NVSDK_NGX_Parameter_OutputInterpolated`** and the optional output real frame **`NVSDK_NGX_Parameter_OutputReal`** *must* match the underlying resolution of the input backbuffer color resource **`NVSDK_NGX_DLSSG_Parameter_Backbuffer`**, in addition to the subrect sizes matching the create-time width and height. This is because the "fill outside subrect" operation is the only DLSS-FG action that accesses pixels outside the defined subrects, requiring consistency in the full underlying resource dimensions for this operation. When the flag *is* set, this underlying resource size matching requirement does *not* apply, as DLSS-FG only interacts with the pixels within the subrect.

Performance and Resource Management Considerations

Performance Impact

Processing subrects has minimal performance impact compared to processing full-sized resources of the same dimensions. Any potential performance differences, such as cache locality concerns, are typically negligible.

Memory Management

While DLSS-FG itself doesn't inherently use more memory for subrect processing, your application must be mindful if you allocate underlying resources (like large depth or motion vector textures) at a significantly larger size than their currently active subrect requires. This can lead to increased VRAM usage if not judiciously managed, although the benefit of avoiding frequent resource reallocations (as demonstrated in the DRS example) often outweighs this.

Multiple Instances/Viewports

A DLSS-FG instance is an independent, self-contained unit that holds the state and history needed to run DLSS-FG. Each instance operates without directly interfering with other instances. While most applications will only need a single instance at a time, you may want to run multiple instances simultaneously to generate frames for two or more unrelated scenes.

This can be especially useful in scenarios including:

- **Applications with Multiple Windows**: For example, an editor with several windows, such as a 3D scene view and a material editor, where each can independently benefit from frame generation.
- **Stereoscopic Views (VR)**: Each eye in a VR headset can have its own dedicated DLSS-FG instance.
- **Split-screen Applications**: Multiple instances can control separate viewports on the same backbuffer, which is well-suited for split-screen games.

Creating Multiple Instances

Each call to **`NVSDK_NGX_CreateFeature`** (or the **`NGX_CREATE_DLSSG`** helper) creates a separate instance of DLSS-FG. Each instance is identified by a unique **`NVSDK_NGX_Handle*`**.

Each instance requires its own set of creation parameters, which can be specified using the **`NVSDK_NGX_DLSSG_Create_Params`** struct, or set directly in the **`NVSDK_NGX_Parameter`** map. While not strictly required, it's good practice to use a different `NVSDK_NGX_Parameter` map for each instance. This helps prevent unintentionally passing stale data from other instances.

Remember that creating a feature requires an open and recording command list. You may use the same command list for multiple instances or use a different command list for each instance. Either way, make sure to execute the relevant command lists after your `CreateFeature` calls return.

Example

```
// Instance 1 for Main Window
NVSDK_NGX_Handle* ngxHandleMainWindow;
// Allocate using NVSDK_NGX_AllocateParameters, or use an existing parameter map
NVSDK_NGX_Parameter* ngxParametersMain;
NVSDK_NGX_DLSSG_Create_Params createParamsMain = {};

createParamsMain.Width = 3840;      // Output width for main window
createParamsMain.Height = 2160;     // Output height for main window
createParamsMain.NativeBackbufferFormat = DXGI_FORMAT_R10G10B10A2_UNORM;
createParamsMain.RenderWidth = 2560; // Render resolution for main window
createParamsMain.RenderHeight = 1440;
createParamsMain.DynamicResolutionScaling = false;

NVSDK_NGX_Result result = NGX_D3D12_CREATE_DLSSG(
    CommandList, CreationNodeMask, VisibilityNodeMask, &ngxHandleMainWindow,
    ngxParametersMain, &createParamsMain);
if (NVSDK_NGX_FAILED(result))
{
    // Handle error for main instance
}
```

```
// Instance 2 for secondary view
NVSDK_NGX_Handle* ngxHandleAux;
NVSDK_NGX_Parameter* ngxParametersAux; // Separate parameters for secondary instance
NVSDK_NGX_DLSSG_Create_Params createParamsAux = {};

createParamsAux.Width = 1280;       // Output width for secondary
createParamsAux.Height = 720;       // Output height for secondary
createParamsAux.NativeBackbufferFormat = DXGI_FORMAT_R10G10B10A2_UNORM;
createParamsAux.RenderWidth = 640;  // Render resolution for secondary
createParamsAux.RenderHeight = 360;
createParamsAux.DynamicResolutionScaling = false;

result = NGX_D3D12_CREATE_DLSSG(
    CommandList, CreationNodeMask, VisibilityNodeMask, &ngxHandleAux,
    ngxParametersAux, &createParamsAux);
if (NVSDK_NGX_FAILED(result))
{
    // Handle error for secondary instance
}

// Make sure to execute the command list after all creation calls.
```

Evaluating Multiple Instances

Once you've created your instances, you can evaluate them independently using **`NVSDK_NGX_EvaluateFeature`** or the **`NGX_EVALUATE_DLSSG`** helper function. Each `EvaluateFeature` call produces a single interpolated frame for its specified instance.

When evaluating an instance, you must provide its specific **`NVSDK_NGX_Handle*`** along with its unique **`NVSDK_NGX_Parameter`** map.

As with creation, evaluation also requires an open and recording command list, which you'll need to execute after `EvaluateFeature` returns.

Example

```
// Assuming CommandList is open and recording

// Evaluate Main Viewport Instance
NVSDK_NGX_Parameter* ngxParametersMain; // Use parameters associated with the main viewport instance
NVSDK_NGX_DLSSG_Eval_Params evalParamsMain = {};
NVSDK_NGX_DLSSG_Opt_Eval_Params optEvalParamsMain = {};

evalParamsMain.pBackbuffer = pInColorResourceMain;
evalParamsMain.pDepth = pInDepthResourceMain;
evalParamsMain.pMVecs = pInMotionVectorsResourceMain;
evalParamsMain.pOutputInterpFrame = pOutInterpFrameResourceMain; // Output for main viewport
// ...

optEvalParamsMain.reset = false;
optEvalParamsMain.depthInverted = true;
// ...
```

```
NVSDK_NGX_Result result = NGX_D3D12_EVALUATE_DLSSG(
    CommandList, ngxHandleMainViewport, ngxParametersMain, &evalParamsMain, &optEvalParamsMain);
if (NVSDK_NGX_FAILED(result))
{
    // Handle error for main viewport evaluation
}

// Evaluate secondary Instance
NVSDK_NGX_Parameter* ngxParametersAux; // Use parameters associated with the auxiliary instance
NVSDK_NGX_DLSSG_Eval_Params evalParamsAux = {};
NVSDK_NGX_DLSSG_Opt_Eval_Params optEvalParamsAux = {};

evalParamsAux.pBackbuffer = pInColorResourceAux;
evalParamsAux.pDepth = pInDepthResourceAux;
evalParamsAux.pMVecs = pInMotionVectorsResourceAux;
evalParamsAux.pOutputInterpFrame = pOutInterpFrameResourceAux; // Output for aux
// ...

optEvalParamsAux.reset = false;
optEvalParamsAux.depthInverted = true;
// ...

result = NGX_D3D12_EVALUATE_DLSSG(
    CommandList, ngxHandleAux, ngxParametersAux, &evalParamsAux, &optEvalParamsAux);
if (NVSDK_NGX_FAILED(result))
{
    // Handle error for auxiliary evaluation
}

// Make sure to execute the command list after all evaluation calls.
```

Multiple Instances on the Same Backbuffer (Multiple Viewports)

In addition to supporting separate windows, multiple instances of DLSS-FG can also be used to render to different sub-regions of a single backbuffer. This enables scenarios like split-screen rendering.

To prevent each DLSS-FG instance from overwriting another's output, you must define a specific sub-region for each using the subrect parameters.

Defining and Controlling Viewport Subrects

You can define these subrects using the `backbufferSubrectBase` and `backbufferSubrectSize` fields within the **`NVSDK_NGX_DLSSG_Opt_Eval_Params`** struct for each instance, or via the **`NVSDK_NGX_DLSSG_Parameter_BackbufferSubrectBaseX/BaseY/Width/Height`** parameters.

DLSS-FG's default behavior is to fill regions outside of the defined subrect, preventing uninitialized or stale data from being displayed. However, this behavior can cause a problem in multi-instance scenarios, as a subsequent instance would overwrite the output from a previous one. The **`NVSDK_NGX_DLSSG_EvalFlags_UpdateOnlyInsideExtents`** flag can be used to prevent this behaviour. The recommended usage depends on whether your subrects collectively cover the entire backbuffer:

- If subrects cover the entire backbuffer (e.g., a standard split-screen layout), set the **`NVSDK_NGX_DLSSG_EvalFlags_UpdateOnlyInsideExtents`** flag on all instances. This ensures each instance only writes

to its assigned region and avoids unnecessary fill work.
- If subrects do not cover the entire backbuffer (e.g., a letterboxed cinematic or split-screen with gaps), you may want the first instance to initialize the entire resource. In this case, omit the `UpdateOnlyInsideExtents` flag on that first instance and set it on all subsequent instances.

Other Important Considerations

When using multiple instances on a single backbuffer, you should also consider the following:

- **Synchronization**: DLSS-FG does not provide any synchronization between calls to `EvaluateFeature`. It is your responsibility to implement external synchronization to ensure that only one instance operates on the backbuffer at any given time.
- **Timing**: Since all instances contribute to a single output frame, you must display all generated and rendered frames on the same cadence to preserve equal temporal spacing.
- **Overlapping Regions**: If the regions defined for two or more instances overlap, the contents of the shared area will be overwritten by the last instance to write to it. Because DLSS-FG instances process their regions independently, the last instance may lack the necessary historical and motion context for areas handled by other instances. Therefore, overlapping output regions for multiple instances is generally not recommended.

Other input resources can also be specified as subrects of the same resource, or provided as different resources. For detailed information and best practices on defining and managing subrects, refer to the **Subrect Handling** guide.

Example

```
// Evaluate Player 1's Viewport (writes to left subrect)
evalParamsPlayer1.pBackbuffer = pInColorResource;
evalParamsPlayer1.pOutputInterpFrame = pSharedBackbuffer;
optEvalParamsPlayer1.backbufferSubrectBase = {0, 0};
optEvalParamsPlayer1.backbufferSubrectSize = {1920, 2160}; // Left half
optEvalParamsPlayer1.evalFlags = NVSDK_NGX_DLSSG_EvalFlags_UpdateOnlyInsideExtents;
// ...

NVSDK_NGX_Result result = NGX_D3D12_EVALUATE_DLSSG(
    CommandList, ngxHandlePlayer1, ngxParametersPlayer1, &evalParamsPlayer1, &optEvalParamsPlayer1);
if (NVSDK_NGX_FAILED(result))
{
    // Handle error
}

// Evaluate Player 2's Viewport (writes to right subrect)
evalParamsPlayer2.pBackbuffer = pInColorResource; // Same input resource
evalParamsPlayer2.pOutputInterpFrame = pSharedBackbuffer; // Same output resource
optEvalParamsPlayer2.backbufferSubrectBase = {1920, 0}; // Right half
optEvalParamsPlayer2.backbufferSubrectSize = {1920, 2160};
optEvalParamsPlayer2.evalFlags = NVSDK_NGX_DLSSG_EvalFlags_UpdateOnlyInsideExtents;
result = NGX_D3D12_EVALUATE_DLSSG(
    CommandList, ngxHandlePlayer2, ngxParametersPlayer2, &evalParamsPlayer2, &optEvalParamsPlayer2);
if (NVSDK_NGX_FAILED(result))
{
    // Handle error
}
```

Performance Implications

Each DLSS-FG instance operates independently, as if it were the only instance running. This means that the total runtime and memory usage for multiple instances will be approximately the sum of the resources that each instance would consume individually.

There are no inherent performance benefits or resource sharing efficiencies when running multiple instances. Consequently, the SDK does not provide specific mitigation strategies for the aggregate resource consumption. You should profile your application to determine the best optimization strategies for your particular use case.

Additional Note: Internal Synchronization

DLSS-FG does not provide synchronization between `EvaluateFeature` calls; this responsibility lies with the application. However, calls to `CreateFeature` and `ReleaseFeature` *are* synchronized: During these operations, all other `CreateFeature`, `EvaluateFeature`, and `ReleaseFeature` calls are temporarily blocked until the current operation completes.

Lens Distortion and Related Effects

DLSS-FG generates new frames by leveraging data from the game engine, including motion vectors, depth buffers, and camera parameters. These inputs help the algorithm understand the scene's geometry and movement. For the algorithm to work correctly, it is essential that the rendered color buffer is spatially aligned with these guide buffers.

Post-processing distortion effects, like lens distortion, break this alignment by distorting the final color output. Since geometry buffers, like motion vectors and depth, are typically computed before distortion is applied, pixels in the color buffer no longer match up with the corresponding scene data. This can cause visual artifacts.

To solve this, DLSS-FG accepts an optional bidirectional distortion texture. This texture maps pixels between the coordinates in the distorted color and the undistorted guide buffer coordinates, allowing DLSS-FG to generate accurate frames even when distortion effects are present.

What is the Bidirectional Distortion Field?

The bidirectional distortion texture is a four-channel resource that encodes the spatial transformation between distorted and undistorted image spaces.

This guide defines "distorted space" and "undistorted space" as follows:

- **Undistorted Space:** The coordinate system in which motion vectors, depth buffers, and camera parameters are calculated *before* the post-processing distortion is applied.
- **Distorted Space:** The coordinate system in which the final rendered color buffer, as well as the input `HUDLess` and `UI` textures (if provided), are displayed *after* the post-processing distortion has been applied.

Understanding the Bidirectional Mapping

This texture contains two separate 2D vector fields, each stored in a pair of channels and representing an offset in UV coordinates:

- **Forward Mapping (Red and Green Channels):** These channels store the offset vector that maps a pixel from **distorted space to undistorted space**.
  - If you apply this (x, y) offset (x in red, y in green) to the UV coordinates of a pixel in the *distorted* color buffer, you will get its corresponding UV coordinates in the *undistorted* guide buffers.
- **Reverse Mapping (Blue and Alpha Channels):** These channels store the offset vector that maps a pixel from **undistorted space to distorted space**.
  - If you apply this (x, y) offset (x in blue, y in alpha) to UV coordinates in the *undistorted* guide buffers, you will get their corresponding UV coordinates in the *distorted* color buffer.

Example Scenario

Imagine you have a distortion effect that moves the original top-left pixel $(0, 0)$ of the undistorted scene to the center $(0.5, 0.5)$ in the final distorted output. When generating the bidirectional distortion texture to describe this effect:

- To encode the Forward Mapping (Red and Green channels): At the center pixel $(0.5, 0.5)$ in the distortion texture (which corresponds to a location in your distorted output), you would set its RG values to a vector that points back to the

original top-left $(0, 0)$ position (approximately $(-0.5, -0.5)$ in normalized UV space). This describes where the content at this distorted pixel originated from in undistorted space.
- To encode the Reverse Mapping (Blue and Alpha Channels): At the top-left pixel $(0, 0)$ in the distortion texture (which corresponds to a location in your undistorted input), you would set its BA values to a vector that points to its new location at the center $(0.5, 0.5)$ in the distorted output (approximately $(0.5, 0.5)$ in normalized UV space). This describes where the content from this undistorted pixel moves to in distorted space.

Considerations for Using Bidirectional Distortion

- **Texture Resolution:** The recommended resolution for the bidirectional distortion field is half the backbuffer's width and height. This often provides a good balance between accuracy and performance. While you can use any resolution, ensure it maintains spatial alignment with your relevant distorted and undistorted textures. Keep in mind that a smaller resolution might improve performance but could reduce the accuracy of the distortion mapping.
- **Spatial Alignment:** You must ensure the bidirectional distortion field is spatially aligned with both your distorted color buffer and undistorted guide buffers. This means a pixel at a specific (x, y) coordinate in the distortion texture must correspond to the exact same (x, y) screen-space location in the other textures.
- **Data Precision:** To ensure sufficient precision for the distortion offsets, the bidirectional distortion texture must use a format with at least 8 bits per channel. Formats with lower precision, such as RGB10A2, are not recommended as they can lead to visual artifacts in the generated frames.

Examples

Bidirectional Distortion Field Generation

This example shows how to generate the bidirectional distortion field for a barrel distortion.

```
// Distortion strength parameter: negative values create barrel distortion
// (edges pull inward), positive values create pincushion distortion (edges push
// outward). -0.5 gives a moderate barrel distortion.
const float distortionAlpha = -0.5f;

float2 barrelDistortion(float2 inputUV)
{
    // Convert the input coordinates from UV space (0 to 1) to normalized device
    // coordinates (-1 to 1). This centers the distortion effect around the
    // middle of the image
    float2 inputNDC = (inputUV * 2.0f) - 1.0f;

    // Calculate the squared distance from the center point in NDC space. This
    // determines the distortion strength based on distance
    float distanceSquared = dot(inputNDC, inputNDC);

    // Apply the barrel distortion formula
    float distortionFactor = 1.0f + distortionAlpha * distanceSquared;
    float2 distortedNDC = inputNDC / distortionFactor;

    // Convert back to UV coordinates
    return (distortedNDC + 1.0f) / 2.0f;
}

float2 inverseBarrelDistortion(float2 inputUV)
```

```
{
    // Convert the input coordinates from UV space (0 to 1) to normalized device
    // coordinates (-1 to 1)
    float2 inputNDC = (inputUV * 2.0f) - 1.0f;

    // Calculate the squared distance from the center for undistorted coordinates
    float undistortedDistanceSquared = dot(inputNDC, inputNDC);

    // Solve the quadratic equation to find the distorted radius. This is
    // derived by inverting the barrel distortion formula:
    //   undistorted = distorted / (1 + alpha * distorted^2)
    float discriminant = sqrt(1.0f - 4.0f * distortionAlpha * undistortedDistanceSquared);
    float numerator = discriminant - 1.0f;
    float denominator = 2.0f * distortionAlpha * sqrt(undistortedDistanceSquared);
    float distortedRadius = -numerator / denominator;

    // Scale the coordinates to match the solved distorted radius
    float scaleFactor = distortedRadius / sqrt(undistortedDistanceSquared);
    float2 distortedNDC = inputNDC * scaleFactor;

    // Convert back to UV coordinates
    return (distortedNDC + 1.0f) / 2.0f;
}

float2 generateBidirectionalDistortionField(Texture2D output, float2 UV)
{
    // Displacement from distorted to undistorted space is stored in the red/green channels
    float2 rg = barrelDistortion(UV) - UV;
    // Displacement from undistorted to distorted space is stored in the blue/alpha channels
    float2 ba = inverseBarrelDistortion(UV) - UV;

    // Store the displacements in the output texture. Values may be negative, so
    // output should be in a signed format.
    output[UV] = float4(rg, ba);
}
```

Iterative Solver for Inverse Distortion

If your distortion function does not have an analytical inverse, this example HLSL shader shows how to use an iterative Newton-Raphson method to solve for the inverse distortion field. While this method is effective, convergence is not guaranteed for all distortion functions, so make sure that it is suitable for your particular use case.

```
float2 myDistortion(float2 xy)
{
    // The distortion function
}

float loss(float2 Pxy, float2 ab)
{
    float2 Pab = myDistortion(ab);
    float2 delta = Pxy - Pab;
    return dot(delta, delta);
}
```

```
float2 iterativeInverseDistortion(float2 UV)
{
    const float kTolerance = 1e-6f;
    // The delta used for gradient estimation
    const float kGradDelta = 1e-6f;
    // Select a low number of iterations which minimizes the loss
    const int kMaxIterations = 5;
    // Assume a locally uniform distortion field
    const int kImprovedInitialGuess = 1;

    // initial guess
    float2 ab = UV;

    if (kImprovedInitialGuess)
    {
        ab = UV - (myDistortion(UV) - UV);
    }

    for (int i = 0; i < kMaxIterations; ++i)
    {
        float F = loss(UV, ab);

        // Central difference
        const float Fabx1 = loss(UV, ab + float2(kGradDelta * 0.5f, 0));
        const float Fabx0 = loss(UV, ab - float2(kGradDelta * 0.5f, 0));
        const float Faby1 = loss(UV, ab + float2(0, kGradDelta * 0.5f));
        const float Faby0 = loss(UV, ab - float2(0, kGradDelta * 0.5f));

        float2 grad;
        grad.x = (Fabx1 - Fabx0) / kGradDelta;
        grad.y = (Faby1 - Faby0) / kGradDelta;

        const float norm = grad.x * grad.x + grad.y * grad.y;
        if (abs(norm) < kTolerance)
        {
            break;
        }

        float delta_x = F * grad.x / norm;
        float delta_y = F * grad.y / norm;

        ab.x = ab.x - delta_x;
        ab.y = ab.y - delta_y;
    }

    return ab;
}
```

HDR (High Dynamic Range)

This section specifically focuses on HDR considerations for integrating DLSS Frame Generation into applications presenting to HDR-enabled displays.

This guide assumes some familiarity with HDR processing. For Windows/D3D12 applications, the **Microsoft DirectX and Advanced Color guide** is a good resource to familiarize yourself with the concepts, techniques, and API discussed.

Pipeline Placement

DLSS-FG operates on the final, display-ready color buffer. This means it should be placed at the very end of your rendering pipeline, after all rendering and post-processing steps are complete.

Specifically, the input to DLSS-FG must already have tone mapping and the required HDR electro-optical transfer function (EOTF) applied.

If your application is presenting to a SDR display (even if rendering uses linear or HDR values internally), the input to DLSS-FG must be the final SDR frame.

HDR Format

The input color resources to DLSS-FG must be in a display-ready format, with pixel values correctly encoded in the range $[0, 1]$. The recommended HDR format for DLSS-FG is **HDR10 (BT.2100 with ST.2084/PQ EOTF)**, stored in `RGB10A2` (i.e. `DXGI_FORMAT_R10G10B10A2_UNORM` in D3D12, or `VK_FORMAT_A2B10G10R10_UNORM_PACK32` in Vulkan). For Windows/D3D12 applications, refer to **Option 2 in the Microsoft DirectX and Advanced Color guide**.

Any input data that has not been EOTF-encoded to fit within the display-ready $[0, 1]$ range for the target display is not supported. This includes:

- **scRGB data**: If your application presents scRGB buffers with the expectation that the operating system will perform the final tone mapping and EOTF application, these buffers are not suitable as inputs to DLSS-FG.
- **Scene-referred linear data**: Data containing raw light values from your rendering engine, such as buffers from the middle of your rendering pipeline, are not supported.

Before submitting frames to the DLSS-FG algorithm, make sure that the complete HDR pipeline has been completed, including tone mapping and EOTF application. Failing to provide a display-ready signal may lead to artifacts and incorrect rendering.

UI Resources

If the **`NVSDK_NGX_DLSSG_Parameter_HUDLess`** and/or **`NVSDK_NGX_DLSSG_Parameter_UI`** textures are provided, these resources must also account for HDR colors.

The `HUDLess` resource must be identical to the backbuffer, but without any UI elements. This means that you must apply any post-processing, color conversions, or tonemapping performed on the backbuffer equally to `HUDLess`.

You should prepare the `UI` texture so that when it is blended with `HUDLess`, it correctly recreates the backbuffer color in the final display color space. The blending formula is the same as for a standard (SDR) backbuffer:

$$\text{Backbuffer.RGB} = \text{UI.RGB} + (1 - \text{UI.}\alpha) \cdot \text{Hudless.RGB}$$

API Integration

Integrating HDR support with DLSS Frame Generation involves setting specific parameters during the DLSS API calls.

Creating DLSS-FG Instances with HDR

When creating the DLSS-FG feature, make sure to provide the correct backbuffer format.

```
NVSDK_NGX_Handle* ngxHandle;
NVSDK_NGX_Parameter* ngxParameters; // Allocate using NVSDK_NGX_AllocateParameters, or use an existing
parameter map
NVSDK_NGX_DLSSG_Create_Params CreateParams = {};

// Set parameters for HDR
CreateParams.Width = 3840;
CreateParams.Height = 2160;
CreateParams.NativeBackbufferFormat = DXGI_FORMAT_R10G10B10A2_UNORM; // HDR format
CreateParams.RenderWidth = 2560;
CreateParams.RenderHeight = 1440;
NVSDK_NGX_Result Result = NGX_D3D12_CREATE_DLSSG(
      CommandList, CreationNodeMask, VisibilityNodeMask, &ngxHandle, ngxParameters, &CreateParams);
if (NVSDK_NGX_FAILED(Result))
{
    // Handle error
}
```

Evaluating DLSS-FG with HDR

When evaluating the feature, set the **`NVSDK_NGX_DLSSG_Parameter_ColorBuffersHDR`** parameter to 1, or set `OptEvalParams.colorBuffersHDR` to `true` if using the evaluate helper function.

Example:

```
NVSDK_NGX_DLSSG_Eval_Params EvalParams = {};
NVSDK_NGX_DLSSG_Opt_Eval_Params OptEvalParams = {};

// Set evaluation parameters for HDR
EvalParams.pBackbuffer = pInColorResource;
EvalParams.pDepth = pInDepthResource;
EvalParams.pMVecs = pInMotionVectorsResource;
EvalParams.pOutputInterpFrame = pOutInterpFrameResource;
OptEvalParams.reset = reset;
OptEvalParams.colorBuffersHDR = true;
// ...

NVSDK_NGX_Result Result = NGX_D3D12_EVALUATE_DLSSG(CommandList, ngxHandle, ngxParameters, &EvalParams,
&OptEvalParams);
if (NVSDK_NGX_FAILED(Result))
{
    // Handle error
}
```

API Reference

Introduction

This section provides detailed documentation for the DLSS Frame Generation API, including:

- Function signatures and parameters
- Structure definitions and member descriptions
- Parameter names and usage
- Constants and enumerations
- Return codes and error handling

The API reference is organized by feature area and includes both the core NGX SDK API as well as DLSS-FG specific extensions. Each entry includes complete documentation of parameters, return values, and usage requirements.

This reference is intended for developers who are already familiar with the basic concepts of DLSS Frame Generation and are looking for specific details about API usage. For a guided introduction to using the API, please refer to the **Integration Guide**.

API Reference Structure

The API reference is divided into the following sections:

- **Types**: Documentation of types used in the NGX SDK.
- **SDK Functions**: Detailed documentation of all NGX SDK functions, including initialization, feature management, and evaluation functions.
- **Error Codes**: Documentation of all possible error codes returned by NGX SDK functions, including descriptions and recommended handling.
- **Parameters**: Comprehensive listing of all parameters used to configure DLSS Frame Generation features.
- **Helpers**: Helper functions that wrap common DLSS-FG functionality.

Types

Core Types

Type Name	Description
`NVSDK_NGX_Result`	An enumeration that specifies the result of an NGX operation. These status codes are detailed in the **Error Codes** section.
`NVSDK_NGX_Feature`	Identifies different NGX features available in the SDK.
`NVSDK_NGX_Handle`	Represents a handle to an NGX feature instance.
`NVSDK_NGX_Parameter`	A parameter map interface used to set and get parameters for NGX features.
`NVSDK_NGX_Coordinates`	Represents a 2D coordinate point.
`NVSDK_NGX_Dimensions`	Represents the dimensions (width and height) of a 2D region.
`NVSDK_NGX_PrecisionInfo`	Provides parameters to rescale or shift values in an input resource.

Feature Discovery and Initialization

Type Name	Description
`NVSDK_NGX_FeatureCommonInfo`	Contains common information used by NGX features.
`NVSDK_NGX_PathListInfo`	Specifies a list of additional directories where NGX can locate feature DLLs.
`NVSDK_NGX_FeatureDiscoveryInfo`	Contains information used by NGX to determine feature availability.
`NVSDK_NGX_FeatureRequirement`	Specifies the system requirements for an NGX feature.
`NVSDK_NGX_Feature_Support_Result`	Specifies the result of a feature support check.
`NVSDK_NGX_Application_Identifier`	Used to identify applications for various SDK functionalities and optimizations.
`NVSDK_NGX_ProjectIdDescription`	Provides details about your application's project ID and rendering engine.
`NVSDK_NGX_EngineType`	Specifies the rendering engine type used by the application.

Logging

Type Name	Description
`NVSDK_NGX_LoggingInfo`	Contains configuration information for NGX SDK logging.
`NVSDK_NGX_Logging_Level`	Specifies the verbosity level for NGX SDK logging.
`NVSDK_NGX_AppLogCallback`	A callback function type that allows applications to receive log messages from the NGX SDK.

Allocation Callbacks

Type Name	Description
`PFN_NVSDK_NGX_D3D12_ResourceAllocCallback`	The function pointer type for a callback that allows you to control the allocation of a D3D12 resource requested by NGX.
`PFN_NVSDK_NGX_ResourceReleaseCallback`	The function pointer type for a callback to control the release of a resource previously allocated for NGX.

Vulkan Resource Info

Type Name	Description
`NVSDK_NGX_Resource_VK`	Represents a Vulkan resource in the NGX SDK, which can be either a `VkImageView` or a `VkBuffer`.
`NVSDK_NGX_Resource_VK_Type`	Specifies the type of Vulkan resource in an `NVSDK_NGX_Resource_VK` structure.
`NVSDK_NGX_BufferInfo_VK`	Contains Vulkan buffer-specific metadata.
`NVSDK_NGX_ImageViewInfo_VK`	Contains metadata for a Vulkan image view.

DLSS-FG Types

Type Name	Description
`NVSDK_NGX_DLSSG_EvalFlags`	Specifies evaluation flags for DLSSG (DLSS Frame Generation).

Core Types

NVSDK_NGX_Result

An enumeration that specifies the result of an NGX operation. These status codes are detailed in the **Error Codes** section.

Most NGX operations will return a result indicating success or failure. The value `NVSDK_NGX_Result_Success` indicates that the operation completed successfully. All other values indicate some form of failure, with specific error codes providing more detailed information about the nature of the failure.

The macro `NVSDK_NGX_SUCCEED(value)` should be used to check if a result indicates success, while `NVSDK_NGX_FAILED(value)` should be used to check if a result indicates failure.

NVSDK_NGX_Feature

Identifies different NGX features available in the SDK.

```
typedef enum NVSDK_NGX_Feature
{
    NVSDK_NGX_Feature_Reserved0 = 0,
    NVSDK_NGX_Feature_SuperSampling = 1,
    NVSDK_NGX_Feature_InPainting = 2,
    NVSDK_NGX_Feature_ImageSuperResolution = 3,
    NVSDK_NGX_Feature_SlowMotion = 4,
    NVSDK_NGX_Feature_VideoSuperResolution = 5,
    NVSDK_NGX_Feature_Reserved1 = 6,
    NVSDK_NGX_Feature_Reserved2 = 7,
    NVSDK_NGX_Feature_Reserved3 = 8,
    NVSDK_NGX_Feature_ImageSignalProcessing = 9,
    NVSDK_NGX_Feature_DeepResolve = 10,
    NVSDK_NGX_Feature_FrameGeneration = 11,
    NVSDK_NGX_Feature_DeepDVC = 12,
    NVSDK_NGX_Feature_RayReconstruction = 13,
    NVSDK_NGX_Feature_Reserved14 = 14,
    NVSDK_NGX_Feature_Reserved15 = 15,
    NVSDK_NGX_Feature_Reserved16 = 16,
    NVSDK_NGX_Feature_Count,
    NVSDK_NGX_Feature_Reserved_SDK = 32764,
    NVSDK_NGX_Feature_Reserved_Core = 32765,
    NVSDK_NGX_Feature_Reserved_Unknown = 32766
} NVSDK_NGX_Feature;
```

This enum is used to identify which NGX feature to create or use when calling various NGX functions.

Fields

- `NVSDK_NGX_Feature_SuperSampling`: DLSS Super Resolution.
- `NVSDK_NGX_Feature_InPainting`: Deep-Learning Inpainting (Deprecated).
- `NVSDK_NGX_Feature_ImageSuperResolution`: Deep-Learning Image Super Resolution (DLISR).
- `NVSDK_NGX_Feature_SlowMotion`: Deep-Learning Slow-Motion (Deprecated).
- `NVSDK_NGX_Feature_VideoSuperResolution`: Deep-Learning Video Super Resolution (DLVSR).

- `NVSDK_NGX_Feature_ImageSignalProcessing`: Deep-Learning Image Signal Processing (DLISP).
- `NVSDK_NGX_Feature_DeepResolve`: Deep Resolve (Deprecated).
- `NVSDK_NGX_Feature_FrameGeneration`: DLSS Frame Generation.
- `NVSDK_NGX_Feature_DeepDVC`: RTX Dynamic Vibrance (DeepDVC/Deep Learning Digital Vibrance Control).
- `NVSDK_NGX_Feature_RayReconstruction`: DLSS Ray Reconstruction.
- `NVSDK_NGX_Feature_Count`: Total count of features.

NVSDK_NGX_Handle

Represents a handle to an NGX feature instance.

```
typedef struct NVSDK_NGX_Handle {
    unsigned int Id;
} NVSDK_NGX_Handle;
```

This structure is used to identify and reference specific instances of NGX features. The handle is created when a feature is initialized using **`NVSDK_NGX_CreateFeature`** and is used in subsequent calls to **`NVSDK_NGX_EvaluateFeature`** to identify which feature instance to evaluate. The handle is not reference counted, so after calling **`NVSDK_NGX_ReleaseFeature`** on a handle, it becomes invalid and should not be used again.

Fields

- `Id`: Identifier for the feature instance.

NVSDK_NGX_Parameter

A parameter map interface used to set and get parameters for NGX features.

```
struct NVSDK_NGX_Parameter
{
    virtual void Set(const char* InName, unsigned long long InValue) = 0;
    virtual void Set(const char* InName, float InValue) = 0;
    virtual void Set(const char* InName, double InValue) = 0;
    virtual void Set(const char* InName, unsigned int InValue) = 0;
    virtual void Set(const char* InName, int InValue) = 0;
    virtual void Set(const char* InName, ID3D11Resource* InValue) = 0;
    virtual void Set(const char* InName, ID3D12Resource* InValue) = 0;
    virtual void Set(const char* InName, void* InValue) = 0;
    virtual NVSDK_NGX_Result Get(const char* InName, unsigned long long* OutValue) const = 0;
    virtual NVSDK_NGX_Result Get(const char* InName, float* OutValue) const = 0;
    virtual NVSDK_NGX_Result Get(const char* InName, double* OutValue) const = 0;
    virtual NVSDK_NGX_Result Get(const char* InName, unsigned int* OutValue) const = 0;
    virtual NVSDK_NGX_Result Get(const char* InName, int* OutValue) const = 0;
    virtual NVSDK_NGX_Result Get(const char* InName, ID3D11Resource** OutValue) const = 0;
    virtual NVSDK_NGX_Result Get(const char* InName, ID3D12Resource** OutValue) const = 0;
    virtual NVSDK_NGX_Result Get(const char* InName, void** OutValue) const = 0;
    virtual void Reset() = 0;
};
```

This interface provides a way to configure various aspects of NGX features through named parameters. The parameter map can be created using **`NVSDK_NGX_AllocateParameters`** (creates a new empty map) or **`NVSDK_NGX_GetCapabilityParameters`**

(creates a map pre-populated with NGX capabilities). The parameter map must be destroyed using **`NVSDK_NGX_DestroyParameters`** when no longer needed.

Members

- `Set(InName, InValue)`: Sets the given parameter to a value.
- `Get(InName, OutValue) const`: Retrieves the value of a parameter.
- `Reset()`: Resets all parameters in the map (except those populated by `NVSDK_NGX_GetCapabilityParameters`, if any).

NVSDK_NGX_Coordinates

Represents a 2D coordinate point.

```
typedef struct NVSDK_NGX_Coordinates
{
    unsigned int X;
    unsigned int Y;
} NVSDK_NGX_Coordinates;
```

This structure specifies a 2D coordinate in pixels, typically defining the top-left corner of a rectangular region. It is commonly used in conjunction with **`NVSDK_NGX_Dimensions`** to fully specify an area within a texture or buffer.

Fields

- `X`: The X-coordinate of the 2D point.
- `Y`: The Y-coordinate of the 2D point.

NVSDK_NGX_Dimensions

Represents the dimensions (width and height) of a 2D region.

```
typedef struct NVSDK_NGX_Dimensions
{
    unsigned int Width;
    unsigned int Height;
} NVSDK_NGX_Dimensions;
```

This structure specifies the width and height of a 2D region in pixels. It is commonly used in conjunction with **`NVSDK_NGX_Coordinates`** to fully define an area within a texture or buffer.

Fields

- `Width`: The width of the 2D region.
- `Height`: The height of the 2D region.

NVSDK_NGX_PrecisionInfo

Provides parameters to rescale or shift values in an input resource.

```
typedef struct NVSDK_NGX_PrecisionInfo
{
    unsigned int IsLowPrecision;
    float Bias;
    float Scale;
} NVSDK_NGX_PrecisionInfo;
```

This structure provides parameters necessary to convert or transform the numerical values of a resource. It is commonly used to adjust values for different scales (e.g., to convert between normalized and pixel units), or to remap values to better utilize a data type's range, thereby enhancing effective precision. This allows you to control how the SDK accurately processes your input data, regardless of its original format or value range.

Fields

- `IsLowPrecision`: A flag indicating whether the associated resource buffer uses low precision (1) or high precision (0).
- `Bias`: A value used in the conversion formula, such that high precision (`hi`) is derived from low precision (`lo`) using the formula: $hi = lo \times Scale + Bias$.
- `Scale`: A value used in the conversion formula, such that high precision (`hi`) is derived from low precision (`lo`) using the formula: $hi = lo \times Scale + Bias$.

Feature Discovery and Initialization Types

NVSDK_NGX_FeatureCommonInfo

Contains common information used by NGX features.

```
typedef struct NVSDK_NGX_FeatureCommonInfo
{
    NVSDK_NGX_PathListInfo PathListInfo;
    NVSDK_NGX_FeatureCommonInfo_Internal* InternalData;
    NVSDK_NGX_LoggingInfo LoggingInfo;
} NVSDK_NGX_FeatureCommonInfo;
```

Use this structure to provide common configuration information for NGX features. It is commonly used when initializing the NGX SDK to provide configuration information for feature discovery and initialization.

Fields

- `PathListInfo`: A **`NVSDK_NGX_PathListInfo`** structure that specifies additional search paths for feature DLLs beyond the default application directory. The paths are searched in descending order of priority.
- `InternalData`: A pointer used internally by the NGX SDK. The SDK will allocate and manage this pointer. You do not need to initialize this value.
- `LoggingInfo`: A **`NVSDK_NGX_LoggingInfo`** structure that contains logging configuration information.

NVSDK_NGX_PathListInfo

Specifies a list of additional directories where NGX can locate feature DLLs.

```
typedef struct NVSDK_NGX_PathListInfo
{
    wchar_t const* const* Path;
    unsigned int Length;
} NVSDK_NGX_PathListInfo;
```

This structure allows you to provide NGX with a list of custom paths where it can search for feature DLLs. By default, NGX looks for feature DLLs in the application's executable directory. Use this struct when you need NGX to search in other, additional locations.

Fields

- `Path`: A pointer to an array of wide-character strings (`wchar_t const*`), where each string specifies an absolute or relative path to a directory.
- `Length`: The number of paths in the `Path` array.

NVSDK_NGX_FeatureDiscoveryInfo

Contains information used by NGX to determine feature availability.

```
typedef struct NVSDK_NGX_FeatureDiscoveryInfo
{
    NVSDK_NGX_Version SDKVersion;
    NVSDK_NGX_Feature FeatureID;
    NVSDK_NGX_Application_Identifier Identifier;
    const wchar_t* ApplicationDataPath;
    const NVSDK_NGX_FeatureCommonInfo* FeatureInfo;
} NVSDK_NGX_FeatureDiscoveryInfo;
```

This structure is used to identify system requirements to support a given NGX feature on a system.

Fields

- `SDKVersion`: The version of the NGX SDK being used.
- `FeatureID`: **`NVSDK_NGX_Feature`** ID of the feature being queried.
- `Identifier`: Unique identifier for your application.
- `ApplicationDataPath`: Path to a folder where logs and temporary files can be stored (requires write access).
- `FeatureInfo`: Pointer to common feature information, including paths to locate feature DLLs.

NVSDK_NGX_FeatureRequirement

Specifies the system requirements for an NGX feature.

```
typedef struct NVSDK_NGX_FeatureRequirement
{
    NVSDK_NGX_Feature_Support_Result FeatureSupported;
    unsigned int MinHWArchitecture;
    char MinOSVersion[255];
} NVSDK_NGX_FeatureRequirement;
```

This structure is used to identify system requirements to support a given NGX feature. It is populated by the **`NVSDK_NGX_GetFeatureRequirements`** function and used to determine if a system meets the requirements for a specific NGX feature before attempting to initialize it.

Fields

- `FeatureSupported`: A bitfield indicating why a feature is unsupported, as specified in **`NVSDK_NGX_Feature_Support_Result`**. A value of 0 indicates the feature is supported.
- `MinHWArchitecture`: The minimum hardware architecture version required, corresponding to an `NV_GPU_ARCHITECTURE_ID` value defined in the NvAPI GPU Framework.
- `MinOSVersion`: A string containing the minimum OS version required for the feature.

NVSDK_NGX_Feature_Support_Result

Specifies the result of a feature support check.

```
typedef enum NVSDK_NGX_Feature_Support_Result
{
    NVSDK_NGX_FeatureSupportResult_Supported = 0,
    NVSDK_NGX_FeatureSupportResult_CheckNotPresent = 1,
    NVSDK_NGX_FeatureSupportResult_DriverVersionUnsupported = 2,
    NVSDK_NGX_FeatureSupportResult_AdapterUnsupported = 4,
    NVSDK_NGX_FeatureSupportResult_OSVersionBelowMinimumSupported = 8,
    NVSDK_NGX_FeatureSupportResult_NotImplemented = 16
} NVSDK_NGX_Feature_Support_Result;
```

This enum indicates the result of a feature support check. A value of `NVSDK_NGX_FeatureSupportResult_Supported` (0) confirms the feature is supported. All other values specify a reason why it is not. These values may be combined using a bitwise OR to indicate multiple unsupported conditions.

Fields

- `NVSDK_NGX_FeatureSupportResult_Supported`: The feature is fully supported. This is the default state if no unsupported conditions are identified.
- `NVSDK_NGX_FeatureSupportResult_CheckNotPresent`: The SDK could not determine the feature's support status because the necessary capability check is not available or implemented on the current system/driver.
- `NVSDK_NGX_FeatureSupportResult_DriverVersionUnsupported`: The installed display driver version is too old or otherwise unsupported for this feature.

- `NVSDK_NGX_FeatureSupportResult_AdapterUnsupported`: The installed graphics adapter does not meet the minimum hardware architecture requirements.
- `NVSDK_NGX_FeatureSupportResult_OSVersionBelowMinimumSupported`: The operating system version is older than the minimum required for this feature.
- `NVSDK_NGX_FeatureSupportResult_NotImplemented`: The requested feature or its specific implementation is not present in the current SDK build or is not supported in this context.

NVSDK_NGX_Application_Identifier

Used to identify applications for various SDK functionalities and optimizations.

```
typedef struct NVSDK_NGX_Application_Identifier
{
    NVSDK_NGX_Application_Identifier_Type IdentifierType;
    union {
        NVSDK_NGX_ProjectIdDescription ProjectDesc;
        unsigned long long ApplicationId;
    } v;
} NVSDK_NGX_Application_Identifier;
```

This structure identifies your application to the NGX SDK. This is essential for enabling functionalities such as over-the-air updates (both optional and mandatory) and app-specific customizations for some features, as well as other optimizations and support capabilities. You can identify your application using one of two methods:

- **NVIDIA-Provided App ID**: If NVIDIA has provided a unique identifier for your application, set the `IdentifierType` field to `NVSDK_NGX_Application_Identifier_Type_Application_Id` and populate the `v.ApplicationId` member with the assigned ID.
- **Project ID**: If you have not been provided an NVIDIA App ID, you can supply your project's unique identifier and rendering engine information. To do this, set the `IdentifierType` field to `NVSDK_NGX_Application_Identifier_Type_Project_Id` and populate the `v.ProjectDesc` member with a **`NVSDK_NGX_ProjectIdDescription`** struct.

Fields

- `IdentifierType`: Specifies the method used to identify the application.
- `v`: A union containing the specific identifier data.
  - `ProjectDesc`: A `NVSDK_NGX_ProjectIdDescription` struct providing detailed project and engine information.
  - `ApplicationId`: The unique application ID provided by NVIDIA.

NVSDK_NGX_ProjectIdDescription

Provides details about your application's project ID and rendering engine.

```
typedef struct NVSDK_NGX_ProjectIdDescription
{
    const char* ProjectId;
    NVSDK_NGX_EngineType EngineType;
    const char* EngineVersion;
} NVSDK_NGX_ProjectIdDescription;
```

Use this structure to identify your application by its project and rendering engine details when you do not have a unique application ID provided by NVIDIA. This information helps the SDK provide tailored optimizations and support.

Fields

- `ProjectId`: A unique identifier for your project, either provided by your rendering engine or defined by your development team.
- `EngineType`: The type of rendering engine your application uses (e.g., Unreal Engine, Unity, a custom engine). Refer to the **`NVSDK_NGX_EngineType`** enum for valid values.
- `EngineVersion`: The version string of the rendering engine used by your application (e.g., "5.3.1").

NVSDK_NGX_EngineType

Specifies the rendering engine type used by the application.

```
typedef enum NVSDK_NGX_EngineType
{
    NVSDK_NGX_ENGINE_TYPE_CUSTOM = 0,
    NVSDK_NGX_ENGINE_TYPE_UNREAL,
    NVSDK_NGX_ENGINE_TYPE_UNITY,
    NVSDK_NGX_ENGINE_TYPE_OMNIVERSE,
    NVSDK_NGX_ENGINE_COUNT
} NVSDK_NGX_EngineType;
```

This enum is used alongside a Project ID to identify the rendering engine being used. When using an engine not in this list, use `NVSDK_NGX_ENGINE_TYPE_CUSTOM`.

Fields

- `NVSDK_NGX_ENGINE_TYPE_CUSTOM`: Custom/other engine.
- `NVSDK_NGX_ENGINE_TYPE_UNREAL`: Unreal Engine.
- `NVSDK_NGX_ENGINE_TYPE_UNITY`: Unity Engine.
- `NVSDK_NGX_ENGINE_TYPE_OMNIVERSE`: NVIDIA Omniverse.
- `NVSDK_NGX_ENGINE_COUNT`: The number of supported engine types.

Logging Types

NVSDK_NGX_LoggingInfo

Contains configuration information for NGX SDK logging.

```
typedef struct NVSDK_NGX_LoggingInfo
{
    NVSDK_NGX_AppLogCallback LoggingCallback;
    NVSDK_NGX_Logging_Level MinimumLoggingLevel;
    bool DisableOtherLoggingSinks;
} NVSDK_NGX_LoggingInfo;
```

This structure is used to configure how the NGX SDK handles logging. This structure is commonly used in the `NVSDK_NGX_FeatureCommonInfo` structure to provide logging configuration for NGX features.

Fields

- `LoggingCallback`: An application-provided callback function that receives log messages from the SDK. The callback must be thread-safe and handle concurrent calls from multiple threads.
- `MinimumLoggingLevel`: The minimum level of logging detail to use. If this is higher than the logging level otherwise configured, it will override that logging level. Otherwise, the other logging level will be used.
- `DisableOtherLoggingSinks`: When set to true, this disables writing log lines to sinks other than the app log callback. This is useful when the application provides its own logging callback and wants to handle all logging itself. The `LoggingCallback` must be non-null and point to a valid logging callback if this is set to true.

NVSDK_NGX_Logging_Level

Specifies the verbosity level for NGX SDK logging.

```
typedef enum 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;
```

This enumeration controls the verbosity of logging in the NGX SDK. You set the logging level through the **`NVSDK_NGX_LoggingInfo`** structure when initializing the SDK. NGX determines the actual logging level by taking the higher of your configured level and the logging level currently active within the SDK.

Fields

- `NVSDK_NGX_LOGGING_LEVEL_OFF`: Disables all logging output.
- `NVSDK_NGX_LOGGING_LEVEL_ON`: Enables basic logging for important events and errors.
- `NVSDK_NGX_LOGGING_LEVEL_VERBOSE`: Enables detailed logging including debug information.
- `NVSDK_NGX_LOGGING_LEVEL_NUM`: The total number of logging levels defined.

NVSDK_NGX_AppLogCallback

A callback function type that allows applications to receive log messages from the NGX SDK.

```
typedef void NVSDK_CONV (*NVSDK_NGX_AppLogCallback)(
    const char* message,
    NVSDK_NGX_Logging_Level loggingLevel,
    NVSDK_NGX_Feature sourceComponent
);
```

This callback allows you to integrate NGX SDK log messages into your application's own logging pipeline. You provide this callback via the **`NVSDK_NGX_LoggingInfo`** structure when initializing the SDK, giving you control over how NGX log messages are processed, displayed, or stored.

When implementing this callback, you must ensure it:

- Is thread-safe and can handle concurrent calls from multiple threads.
- Processes each message completely before returning, as the message string is not guaranteed to remain valid or in memory afterwards.
- Is able to handle multibyte characters in the message string.

Fields

- `message`: The log message as a null-terminated string.
- `loggingLevel`: The logging level of the message.
- `sourceComponent`: The NGX feature that generated the message.

Allocation Callback Types

PFN_NVSDK_NGX_D3D12_ResourceAllocCallback

The function pointer type for a callback that allows you to control the allocation of a D3D12 resource requested by NGX.

```
typedef void (NVSDK_CONV *PFN_NVSDK_NGX_D3D12_ResourceAllocCallback)(
    D3D12_RESOURCE_DESC *InDesc, int InState,
    CD3DX12_HEAP_PROPERTIES *InHeap, ID3D12Resource **OutResource
);
```

NGX will call a function of this type to request a D3D12 resource for its internal use. You must create the resource based on the provided resource description, heap properties, and initial state, then populate the `OutResource` pointer. If the allocation fails, you must set `OutResource` to `nullptr`. The SDK takes ownership of the allocated resource and manages its lifetime. While NGX generally calls this function during **`NVSDK_NGX_CreateFeature`**, your callback must be resilient to being called at any point in a feature's lifecycle.

Parameters

- `InDesc`: A pointer to a `D3D12_RESOURCE_DESC` containing the properties of the resource to be allocated.
- `InState`: The initial state that the allocated resource must be in.
- `InHeap`: A pointer to the `CD3DX12_HEAP_PROPERTIES` for the heap where the resource should be allocated.
- `OutResource`: A pointer that you must populate with the allocated resource.

PFN_NVSDK_NGX_ResourceReleaseCallback

The function pointer type for a callback to control the release of a resource previously allocated for NGX.

```
typedef void (NVSDK_CONV *PFN_NVSDK_NGX_ResourceReleaseCallback)(IUnknown *InResource);
```

NGX will call a function of this type to release a resource that it previously requested via the allocation callback. For every resource provided to NGX via the allocation callback, NGX will make a corresponding call to this function when it is no longer needed. You must release the resource and its associated memory. While NGX generally calls this function when a feature is destroyed, your callback must be resilient to being called at any point in a feature's lifecycle.

Parameters

- `InResource`: The resource to release.

Vulkan Resource Info Types

NVSDK_NGX_Resource_VK

Represents a Vulkan resource in the NGX SDK, which can be either a `VkImageView` or a `VkBuffer`.

```
typedef struct NVSDK_NGX_Resource_VK {
    union {
        NVSDK_NGX_ImageViewInfo_VK ImageViewInfo;
        NVSDK_NGX_BufferInfo_VK BufferInfo;
    } Resource;
    NVSDK_NGX_Resource_VK_Type Type;
    bool ReadWrite;
} NVSDK_NGX_Resource_VK;
```

This structure is used to provide Vulkan resources to the NGX SDK. This structure is commonly used when passing resources to NGX features like DLSS, DLSS-G, and other Vulkan-based features.

Fields

- `Resource`: A union holding the specific resource information.
  - `ImageViewInfo`: Information about a `VkImageView` resource, including the image view handle, associated image, subresource range, format, and dimensions.
  - `BufferInfo`: Information about a `VkBuffer` resource, including the buffer handle and size.
- `Type`: An enum indicating whether this is a `VkImageView` or `VkBuffer` resource.
- `ReadWrite`: A boolean indicating if the resource supports read and write access. For `VkBuffer` resources, this is `true` if `VkBufferUsageFlags` includes `VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT` or `VK_BUFFER_USAGE_STORAGE_BUFFER_BIT`. For `VkImage` resources, this is `true` if `VkImageUsageFlags` for the associated `VkImage` includes `VK_IMAGE_USAGE_STORAGE_BIT`.

NVSDK_NGX_Resource_VK_Type

Specifies the type of Vulkan resource in an `NVSDK_NGX_Resource_VK` structure.

```
typedef enum NVSDK_NGX_Resource_VK_Type
{
    NVSDK_NGX_RESOURCE_VK_TYPE_VK_IMAGEVIEW,
    NVSDK_NGX_RESOURCE_VK_TYPE_VK_BUFFER
} NVSDK_NGX_Resource_VK_Type;
```

This enum is used to indicate which type of Vulkan resource is contained in the union field of an `NVSDK_NGX_Resource_VK` structure. It helps the SDK determine how to properly handle the resource and which fields in the union are valid.

Fields

- `NVSDK_NGX_RESOURCE_VK_TYPE_VK_IMAGEVIEW`: The resource is a `VkImageView`.
- `NVSDK_NGX_RESOURCE_VK_TYPE_VK_BUFFER`: The resource is a `VkBuffer`.

NVSDK_NGX_BufferInfo_VK

Contains Vulkan buffer-specific metadata.

```
typedef struct NVSDK_NGX_BufferInfo_VK {
    VkBuffer Buffer;
    unsigned int SizeInBytes;
} NVSDK_NGX_BufferInfo_VK;
```

This structure is used to provide information about a Vulkan buffer resource to the NGX SDK. This structure is typically used as part of the `NVSDK_NGX_Resource_VK` union when working with buffer resources in Vulkan. It provides the necessary metadata for the SDK to properly handle buffer operations.

Fields

- `Buffer`: The Vulkan `VkBuffer` resource handle representing the actual buffer resource.
- `SizeInBytes`: The total size of the resource in bytes.

NVSDK_NGX_ImageViewInfo_VK

Contains metadata for a Vulkan image view.

```
typedef struct NVSDK_NGX_ImageViewInfo_VK {
    VkImageView ImageView;
    VkImage Image;
    VkImageSubresourceRange SubresourceRange;
    VkFormat Format;
    unsigned int Width;
    unsigned int Height;
} NVSDK_NGX_ImageViewInfo_VK;
```

This structure is used to provide image view-specific metadata for Vulkan resources in the NGX SDK. This structure is commonly used as part of `NVSDK_NGX_Resource_VK` to represent image resources in Vulkan-based NGX features.

Fields

- `ImageView`: The Vulkan `VkImageView` resource handle.
- `Image`: The Vulkan `VkImage` associated with this `VkImageView`.
- `SubresourceRange`: The `VkImageSubresourceRange` for this `VkImageView`, defining which parts of the image are accessible through this view.
- `Format`: The format of the resource.
- `Width`: The width of the resource in pixels.
- `Height`: The height of the resource in pixels.

DLSS-FG Types

NVSDK_NGX_DLSSG_EvalFlags

Specifies evaluation flags for DLSSG (DLSS Frame Generation).

```
typedef enum NVSDK_NGX_DLSSG_EvalFlags
{
    NVSDK_NGX_DLSSG_EvalFlags_UpdateOnlyInsideExtents = (1 << 0)
} NVSDK_NGX_DLSSG_EvalFlags;
```

This enumeration controls how DLSSG updates the output buffer during evaluation. The flags can be set using the parameter `NVSDK_NGX_DLSSG_Parameter_EvalFlags`.

Fields

- `NVSDK_NGX_DLSSG_EvalFlags_UpdateOnlyInsideExtents`: When set, DLSS-FG updates pixels only within the backbuffer extent, leaving regions outside unchanged. If this flag is not set, the algorithm fills regions outside the subrect, which may overwrite existing content. This flag is useful when you want to preserve content in regions outside the main rendering area.

SDK Functions

Function	Description
`NVSDK_NGX_Init`	Initializes a new SDK instance.
`NVSDK_NGX_Init_with_ProjectID`	Initializes a new SDK instance using a project ID.
`NVSDK_NGX_Shutdown`	Shuts down the current SDK instance and releases all resources.
`NVSDK_NGX_GetParameters`	Retrieves a parameter map used to set parameters needed by the SDK.
`NVSDK_NGX_AllocateParameters`	Allocates a parameter map used to set parameters needed by the SDK.
`NVSDK_NGX_GetCapabilityParameters`	Allocates a parameter map populated with NGX and feature capabilities.
`NVSDK_NGX_DestroyParameters`	Destroys the specified parameter map.
`NVSDK_NGX_GetScratchBufferSize`	Retrieves the size of the scratch buffer needed for the specified feature.
`NVSDK_NGX_CreateFeature`	Creates and initializes an instance of the specified NGX feature.
`NVSDK_NGX_ReleaseFeature`	Releases the specified NGX feature.
`NVSDK_NGX_GetFeatureRequirements`	Identifies system requirements to support a given NGX feature.
`NVSDK_NGX_VULKAN_GetFeatureInstanceExtensionRequirements`	Identifies Vulkan instance extensions required for NGX feature support.
`NVSDK_NGX_VULKAN_GetFeatureDeviceExtensionRequirements`	Identifies Vulkan device extensions required for NGX feature support.
`NVSDK_NGX_EvaluateFeature`	Evaluates the specified NGX feature using the provided parameters.

Function Documentation

NVSDK_NGX_Init

Initializes a new SDK instance.

```
NVSDK_NGX_Result NVSDK_NGX_D3D11_Init(
    unsigned long long 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_NGX_D3D12_Init(
    unsigned long long InApplicationId, const wchar_t *InApplicationDataPath,
    ID3D12Device *InDevice, const NVSDK_NGX_FeatureCommonInfo *InFeatureInfo = nullptr,
    NVSDK_NGX_Version InSDKVersion = NVSDK_NGX_Version_API);
NVSDK_NGX_Result NVSDK_NGX_CUDA_Init(
    unsigned long long InApplicationId, const wchar_t *InApplicationDataPath,
    const NVSDK_NGX_FeatureCommonInfo *InFeatureInfo = nullptr,
    NVSDK_NGX_Version InSDKVersion = NVSDK_NGX_Version_API);
NVSDK_NGX_Result NVSDK_NGX_VULKAN_Init(
    unsigned long long InApplicationId, const wchar_t *InApplicationDataPath, VkInstance InInstance,
    VkPhysicalDevice InPD, VkDevice InDevice, PFN_vkGetInstanceProcAddr InGIPA = nullptr,
    PFN_vkGetDeviceProcAddr InGDPA = nullptr,
    const NVSDK_NGX_FeatureCommonInfo *InFeatureInfo = nullptr,
    NVSDK_NGX_Version InSDKVersion = NVSDK_NGX_Version_API);
```

Initializes a new SDK instance. Most NGX SDK functions require the NGX SDK to be initialized for the corresponding graphics API and device to function properly. If using the NGX SDK on multiple devices, the SDK needs to be initialized independently for each device.

`NVSDK_NGX_Init` requires an application ID provided by NVIDIA. If an application ID is not available, use **`NVSDK_NGX_Init_with_ProjectID`** to supply your own identifier.

Attempting to call `NVSDK_NGX_Init` with a graphics API and device that is already initialized has no effect.

Be sure to call **`NVSDK_NGX_Shutdown`** to shut down the SDK when it is no longer needed.

Parameters

- `InApplicationId`: Unique ID provided by NVIDIA.
- `InApplicationDataPath`: Directory to store logs and other temporary files. Write access is required. Typically, this would be next to the application executable, or in a location like Documents or ProgramData.
- `InDevice` (D3D11/12 only): DirectX device to use.
- `InInstance/InPD/InDevice` (Vulkan only): Vulkan Instance, PhysicalDevice, and Device to use.
- `InGIPA/InGDPA` (Vulkan only): Optional Vulkan function pointers to `vkGetInstanceProcAddr` and `vkGetDeviceProcAddr`.
- `InFeatureInfo`: Contains information common to all features, including a list of paths where feature DLLs can be located, other than the default path (application directory).
- `InSDKVersion`: Version of the SDK currently in use. Typically, this should be left as the default value, `NVSDK_NGX_Version_API`.

Returns

NVSDK_NGX_Result indicating success or failure. Common values include:

- **NVSDK_NGX_Result_Success** if the function executed successfully.
- **NVSDK_NGX_Result_FAIL_FeatureNotSupported** if the NGX SDK is not supported on the current system.

Check the NGX logs for additional information about any failures.

NVSDK_NGX_Init_with_ProjectID

Initializes a new SDK instance using a project ID.

```
NVSDK_NGX_Result 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_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_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);
NVSDK_NGX_Result 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, PFN_vkGetInstanceProcAddr InGIPA = nullptr,
    PFN_vkGetDeviceProcAddr InGDPA = nullptr,
    const NVSDK_NGX_FeatureCommonInfo *InFeatureInfo = nullptr,
    NVSDK_NGX_Version InSDKVersion = NVSDK_NGX_Version_API);
```

Since: Display Driver Release 460

Initializes a new SDK instance. Most NGX SDK functions require the NGX SDK to be initialized for the corresponding graphics API and device to function properly. If using the NGX SDK on multiple devices, the SDK needs to be initialized independently for each device.

Applications must provide a unique identifier to initialize the NGX SDK. If an identifier was provided by NVIDIA, use **NVSDK_NGX_Init** instead.

For projects using third-party engines such as Unreal Engine or Omniverse, the integration for those engines should handle providing the correct Project ID to NGX. Ensure that the project ID is set in the engine's editor.

For other engines, use **NVSDK_NGX_ENGINE_TYPE_CUSTOM**. The driver will validate that the Project ID is GUID-like, for example, "a0f57b54-1daf-4934-90ae-c4035c19df04". Ensure your project ID matches this format.

Attempting to call `NVSDK_NGX_Init_with_ProjectID` with a graphics API and device that is already initialized has no effect.

Be sure to call **NVSDK_NGX_Shutdown** to shut down the SDK when it is no longer needed.

Parameters

- `InProjectId`: Unique ID provided by the rendering engine used. Must be a GUID-like string.
- `InEngineType`: Rendering engine used by the application/plugin. Use **`NVSDK_NGX_ENGINE_TYPE_CUSTOM`** if the specific engine type is not explicitly supported.
- `InEngineVersion`: Version number of the rendering engine used by the application/plugin.
- `InApplicationDataPath`: Directory to store logs and other temporary files (write access required).
- `InDevice` (D3D11/12 only): DirectX device to use.
- `InInstance/InPD/InDevice` (Vulkan only): Vulkan Instance, PhysicalDevice, and Device to use.
- `InGIPA/InGDPA` (Vulkan only): Optional Vulkan function pointers to `vkGetInstanceProcAddr` and `vkGetDeviceProcAddr`.
- `InFeatureInfo`: Contains information common to all features, including a list of paths where feature DLLs can be located in addition to the default path (the application directory).
- `InSDKVersion`: Version of the SDK currently in use. Typically, this should be left as the default value, `NVSDK_NGX_Version_API`.

Returns

`NVSDK_NGX_Result` indicating success or failure. Common values include:

- **`NVSDK_NGX_Result_Success`** if the function executed successfully.
- **`NVSDK_NGX_Result_InvalidParameter`** if the provided project ID is in an invalid format.
- **`NVSDK_NGX_Result_FAIL_FeatureNotSupported`** if the NGX SDK is not supported on the current system.

Check the NGX logs for additional information about any failures.

NVSDK_NGX_Shutdown

Shuts down the current SDK instance and releases all resources.

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

NVSDK_NGX_Result NVSDK_NGX_D3D11_Shutdown1(ID3D11Device *InDevice);
NVSDK_NGX_Result NVSDK_NGX_D3D12_Shutdown1(ID3D12Device *InDevice);
NVSDK_NGX_Result NVSDK_NGX_CUDA_Shutdown(void);
NVSDK_NGX_Result NVSDK_NGX_VULKAN_Shutdown1(VkDevice InDevice);
```

Since: initial release (Shutdown); Display Driver Release 470 (Shutdown1)

Shuts down the current SDK instance and releases all resources. If a device is provided, `NVSDK_NGX_Shutdown1` shuts down the instance corresponding to the specified device. If `nullptr` is provided, all instances are shut down.

Attempting to shut down a device which is not initialized has no effect.

Deprecation Notice: The use of `Shutdown()` is deprecated. Please use `Shutdown1(nullptr)` instead.

Parameters

- `InDevice`: Device to shut down. If `nullptr`, the SDK is shut down for all devices.

Returns

NVSDK_NGX_Result indicating success or failure. Common values include:

- **NVSDK_NGX_Result_Success** if the function executed successfully.
- **NVSDK_NGX_Result_FAIL_NotInitialized** if the SDK was not initialized.

NVSDK_NGX_GetParameters (Deprecated)

Retrieves a parameter map used to set parameters needed by the SDK.

```
NVSDK_NGX_Result NVSDK_NGX_D3D11_GetParameters(NVSDK_NGX_Parameter **OutParameters);
NVSDK_NGX_Result NVSDK_NGX_D3D12_GetParameters(NVSDK_NGX_Parameter **OutParameters);
NVSDK_NGX_Result NVSDK_NGX_CUDA_GetParameters(NVSDK_NGX_Parameter **OutParameters);
NVSDK_NGX_Result NVSDK_NGX_VULKAN_GetParameters(NVSDK_NGX_Parameter **OutParameters);
```

Deprecation Notice: Use **NVSDK_NGX_AllocateParameters** or **NVSDK_NGX_GetCapabilityParameters** instead.

Retrieves the common **NVSDK_NGX_Parameter** map for providing parameters to the SDK. The `NVSDK_NGX_Parameter` interface allows simple parameter setup using named fields. For example, set the width by calling `Parameters->Set(NVSDK_NGX_Parameter_Width, 100)` or provide a resource pointer by calling `Parameters->Set(NVSDK_NGX_Parameter_Color, resource)`. For more details, see the sample code.

Parameter maps returned by `NVSDK_NGX_GetParameters` are pre-populated with NGX capabilities and available features. Unlike **NVSDK_NGX_AllocateParameters**, parameter maps returned by `NVSDK_NGX_GetParameters` have their lifetimes managed by NGX and must not be destroyed by the application using **NVSDK_NGX_DestroyParameters**.

This function may only be called after a successful call to **NVSDK_NGX_Init**.

Note: Allocated memory will be freed by NGX, so do not use the `free`/`delete` operator.

Parameters

- `OutParameters`: Output pointer that will be populated with an `NVSDK_NGX_Parameter` interface.

Returns

NVSDK_NGX_Result indicating success or failure.

NVSDK_NGX_AllocateParameters

Allocates a parameter map used to set parameters needed by the SDK.

```
NVSDK_NGX_Result NVSDK_NGX_D3D11_AllocateParameters(NVSDK_NGX_Parameter** OutParameters);
NVSDK_NGX_Result NVSDK_NGX_D3D12_AllocateParameters(NVSDK_NGX_Parameter** OutParameters);
NVSDK_NGX_Result NVSDK_NGX_CUDA_AllocateParameters(NVSDK_NGX_Parameter** OutParameters);
NVSDK_NGX_Result NVSDK_NGX_VULKAN_AllocateParameters(NVSDK_NGX_Parameter** OutParameters);
```

Since: Display Driver Release 455

Allocates a new **NVSDK_NGX_Parameter** map for providing parameters to the SDK. The lifetime of this parameter map must be managed by the application. The `NVSDK_NGX_Parameter` interface allows simple parameter setup using named fields. For example, set the width by calling `Parameters->Set(NVSDK_NGX_Parameter_Width, 100)` or provide a resource pointer by calling `Parameters->Set(NVSDK_NGX_Parameter_Color, resource)`. For more details, see the sample code.

Use **NVSDK_NGX_DestroyParameters** to free a parameter map created by `NVSDK_NGX_AllocateParameters`. Parameter maps created by `NVSDK_NGX_AllocateParameters` must NOT be freed using the `free/delete` operator.

Parameter maps created by `NVSDK_NGX_AllocateParameters` do not come pre-populated with NGX capabilities and available features. To create a new parameter map pre-populated with such information, use **NVSDK_NGX_GetCapabilityParameters** instead.

This function may return **NVSDK_NGX_Result_FAIL_OutOfDate** if using an older driver that does not support this API call. In such a case, **NVSDK_NGX_GetParameters** may be used as a fallback.

This function may only be called after a successful call to **NVSDK_NGX_Init**.

Parameters

- `OutParameters`: Output pointer that will be populated with the newly-allocated `NVSDK_NGX_Parameter` interface.

Returns

NVSDK_NGX_Result indicating success or failure. Common values include:

- **NVSDK_NGX_Result_Success** if the function executed successfully.
- **NVSDK_NGX_Result_FAIL_OutOfDate** if this function is not supported by the current display driver.

Check the NGX logs for additional information about any failures.

NVSDK_NGX_GetCapabilityParameters

Allocates a parameter map populated with NGX and feature capabilities.

```
NVSDK_NGX_Result NVSDK_NGX_D3D11_GetCapabilityParameters(NVSDK_NGX_Parameter** OutParameters);
NVSDK_NGX_Result NVSDK_NGX_D3D12_GetCapabilityParameters(NVSDK_NGX_Parameter** OutParameters);
NVSDK_NGX_Result NVSDK_NGX_CUDA_GetCapabilityParameters(NVSDK_NGX_Parameter** OutParameters);
NVSDK_NGX_Result NVSDK_NGX_VULKAN_GetCapabilityParameters(NVSDK_NGX_Parameter** OutParameters);
```

Since: Display Driver Release 455

Allocates a new **NVSDK_NGX_Parameter** map pre-populated with NGX capabilities and information about available features. The output parameter map can also be used in the same ways as a parameter map allocated with **NVSDK_NGX_AllocateParameters**. However, it is not recommended to use `NVSDK_NGX_GetCapabilityParameters` unless querying NGX capabilities due to the overhead associated with pre-populating the parameter map.

Use **NVSDK_NGX_DestroyParameters** to free a parameter map created by `NVSDK_NGX_GetCapabilityParameters`. Parameter maps created by `NVSDK_NGX_GetCapabilityParameters` must NOT be freed using the `free`/`delete` operator.

This function may return **NVSDK_NGX_Result_FAIL_OutOfDate** if using an older driver that does not support this API call. In such a case, **NVSDK_NGX_GetParameters** may be used as a fallback.

This function may only be called after a successful call to **NVSDK_NGX_Init**.

Parameters

- `OutParameters`: Output pointer that will be populated with the newly-allocated `NVSDK_NGX_Parameter` interface.

Returns

`NVSDK_NGX_Result` indicating success or failure. Common values include:

- **`NVSDK_NGX_Result_Success`** if the function executed successfully.
- **`NVSDK_NGX_Result_FAIL_OutOfDate`** if this function is not supported by the current display driver.

Check the NGX logs for additional information about any failures.

NVSDK_NGX_DestroyParameters

Destroys the specified parameter map.

```
NVSDK_NGX_Result NVSDK_NGX_D3D11_DestroyParameters(NVSDK_NGX_Parameter* InParameters);
NVSDK_NGX_Result NVSDK_NGX_D3D12_DestroyParameters(NVSDK_NGX_Parameter* InParameters);
NVSDK_NGX_Result NVSDK_NGX_CUDA_DestroyParameters(NVSDK_NGX_Parameter* InParameters);
NVSDK_NGX_Result NVSDK_NGX_VULKAN_DestroyParameters(NVSDK_NGX_Parameter* InParameters);
```

Since: Display Driver Release 455

Destroys the input parameter map. Once `NVSDK_NGX_DestroyParameters` is called on a parameter map, it must not be used again.

`NVSDK_NGX_DestroyParameters` must not be called on any parameter map returned by **`NVSDK_NGX_GetParameters`**; NGX will manage the lifetime of those parameter maps.

This function may only be called after a successful call to **`NVSDK_NGX_Init`**.

Parameters

- `InParameters`: The parameter map to be destroyed.

Returns

`NVSDK_NGX_Result` indicating success or failure.

NVSDK_NGX_GetScratchBufferSize

Retrieves the size of the scratch buffer needed for the specified feature.

```
NVSDK_NGX_Result NVSDK_NGX_D3D11_GetScratchBufferSize(
    NVSDK_NGX_Feature InFeatureId, const NVSDK_NGX_Parameter *InParameters, size_t *OutSizeInBytes);
NVSDK_NGX_Result NVSDK_NGX_D3D12_GetScratchBufferSize(
    NVSDK_NGX_Feature InFeatureId, const NVSDK_NGX_Parameter *InParameters, size_t *OutSizeInBytes);
NVSDK_NGX_Result NVSDK_NGX_CUDA_GetScratchBufferSize(
    NVSDK_NGX_Feature InFeatureId, const NVSDK_NGX_Parameter *InParameters, size_t *OutSizeInBytes);
NVSDK_NGX_Result NVSDK_NGX_VULKAN_GetScratchBufferSize(
    NVSDK_NGX_Feature InFeatureId, const NVSDK_NGX_Parameter *InParameters, size_t *OutSizeInBytes);
```

The SDK requires a buffer of a certain size provided by the client to initialize some NGX features. Once the feature is no longer needed, the buffer can be released. It is safe to reuse the same scratch buffer for different features as long as the minimum size requirement is met for all features. Some features might not need a scratch buffer, so a return size of 0 is completely valid.

Note: Most current NGX features do not use scratch buffers. Please check the documentation for the particular feature in use to determine whether a scratch buffer needs to be created.

Parameters

- `InFeatureId`: Identifier of the feature to query.
- `InParameters`: Parameters used by the feature to determine scratch buffer size.
- `OutSizeInBytes`: Number of bytes needed for the scratch buffer for the specified feature.

Returns

`NVSDK_NGX_Result` indicating success or failure.

NVSDK_NGX_CreateFeature

Creates and initializes an instance of the specified NGX feature.

```
NVSDK_NGX_Result NVSDK_NGX_D3D11_CreateFeature(
    ID3D11DeviceContext *InDevCtx, NVSDK_NGX_Feature InFeatureID,
    const NVSDK_NGX_Parameter *InParameters, NVSDK_NGX_Handle **OutHandle);
NVSDK_NGX_Result NVSDK_NGX_D3D12_CreateFeature(
    ID3D12GraphicsCommandList *InCmdList, NVSDK_NGX_Feature InFeatureID,
    const NVSDK_NGX_Parameter *InParameters, NVSDK_NGX_Handle **OutHandle);
NVSDK_NGX_Result NVSDK_NGX_CUDA_CreateFeature(
    NVSDK_NGX_Feature InFeatureID, const NVSDK_NGX_Parameter *InParameters,
    NVSDK_NGX_Handle **OutHandle);
NVSDK_NGX_Result NVSDK_NGX_VULKAN_CreateFeature(
    VkCommandBuffer InCmdBuffer, NVSDK_NGX_Feature InFeatureID,
    const NVSDK_NGX_Parameter *InParameters, NVSDK_NGX_Handle **OutHandle);
NVSDK_NGX_Result NVSDK_NGX_VULKAN_CreateFeature1(
    VkDevice InDevice, VkCommandBuffer InCmdList, NVSDK_NGX_Feature InFeatureID,
    const NVSDK_NGX_Parameter *InParameters, NVSDK_NGX_Handle **OutHandle);
```

Since: initial release (CreateFeature); Display Driver Release 470 (CreateFeature1)

Each feature needs to be created before it can be used. The parameters required to create the feature vary. Refer to the feature's documentation and/or sample code to find out which input parameters are needed to create a specific feature.

`NVSDK_NGX_CreateFeature` creates an **`NVSDK_NGX_Handle*`** to the feature, which is needed to reference the feature in later calls to **`NVSDK_NGX_EvaluateFeature`**. Multiple instances of the same feature may be created, and evaluating one instance of a feature should not change the state of other instances.

When a feature instance is no longer needed, it should be freed using **`NVSDK_NGX_ReleaseFeature`**.

(Vulkan only) If multiple devices are in use, the device to create the feature on must be specified using `CreateFeature1()`.

Parameters

- `InCmdList` (D3D12 only): Command list to use to execute GPU commands. Must:
  - Be open and recording.
  - Have a node mask that includes the device provided in **`NVSDK_NGX_D3D12_Init`**.
  - Executed on a non-copy command queue.
- `InDevCtx` (D3D11 only): Device context to use to execute GPU commands. Must be an immediate context. Passing a deferred context is not supported and will cause the API to return an error.
- `InCmdBuffer` (Vulkan only): Command buffer to use to execute GPU commands. Must be open and recording.
- `InDevice` (Vulkan only, with `CreateFeature1`): Vulkan device on which to create the feature.
- `InFeatureID`: Identifier of the feature to initialize.
- `InParameters`: Parameters used to create the feature.
- `OutHandle`: Handle which uniquely identifies the feature.

Returns

`NVSDK_NGX_Result` indicating success or failure. Common values include:

- **`NVSDK_NGX_Result_Success`** if the function executed successfully.
- **`NVSDK_NGX_Result_FAIL_UnableToInitializeFeature`** if the feature cannot be created, for example, because the feature is not supported on the current hardware, or because the DLL could not be found.
- **`NVSDK_NGX_Result_FAIL_InvalidParameter`** if any of the input parameters are invalid.
- **`NVSDK_NGX_Result_FAIL_FeatureAlreadyExists`** if a feature instance with the same parameters has already been created, and the feature doesn't allow multiple instances to be created with the same parameters.

Check the NGX logs for additional information about any failures.

NVSDK_NGX_ReleaseFeature

Releases the specified NGX feature.

```
NVSDK_NGX_Result NVSDK_NGX_D3D11_ReleaseFeature(NVSDK_NGX_Handle *InHandle);
NVSDK_NGX_Result NVSDK_NGX_D3D12_ReleaseFeature(NVSDK_NGX_Handle *InHandle);
NVSDK_NGX_Result NVSDK_NGX_CUDA_ReleaseFeature(NVSDK_NGX_Handle *InHandle);
NVSDK_NGX_Result NVSDK_NGX_VULKAN_ReleaseFeature(NVSDK_NGX_Handle *InHandle);
```

Releases the feature associated with the given handle and frees its memory. Handles are not reference counted, so after this call, it is invalid to use the provided handle.

Parameters

- `InHandle`: Handle to the feature to be released.

Returns

`NVSDK_NGX_Result` indicating success or failure.

NVSDK_NGX_GetFeatureRequirements

Identifies system requirements to support a given NGX feature.

```
NVSDK_NGX_Result NVSDK_NGX_D3D11_GetFeatureRequirements(
    IDXGIAdapter *Adapter, const NVSDK_NGX_FeatureDiscoveryInfo *FeatureDiscoveryInfo,
    NVSDK_NGX_FeatureRequirement *OutSupported);
NVSDK_NGX_Result NVSDK_NGX_D3D12_GetFeatureRequirements(
    IDXGIAdapter *Adapter, const NVSDK_NGX_FeatureDiscoveryInfo *FeatureDiscoveryInfo,
    NVSDK_NGX_FeatureRequirement *OutSupported);
NVSDK_NGX_Result NVSDK_NGX_VULKAN_GetFeatureRequirements(
    const VkInstance Instance, const VkPhysicalDevice PhysicalDevice,
    const NVSDK_NGX_FeatureDiscoveryInfo *FeatureDiscoveryInfo,
    NVSDK_NGX_FeatureRequirement *OutSupported);
```

Since: Display Driver Release 525

Utility function used to identify system requirements to support a given NGX feature on a system, given its display device subsystem adapter information that will be subsequently used for creating the graphics device. The output parameter `OutSupported` will be populated with requirements and is valid if and only if the function returns **`NVSDK_NGX_Result_Success`**. The result includes:

- `FeatureSupported`: Bitfield of reasons why the feature is unsupported, as specified in `NVSDK_NGX_Feature_Support_Result`. 0 if the feature is supported.
- `MinHWArchitecture`: The hardware architecture version corresponding to an `NV_GPU_ARCHITECTURE_ID` value defined in the NvAPI GPU Framework.
- `MinOSVersion`: The minimum OS version required for the provided feature.

`NVSDK_NGX_Init` does NOT need to be called before calling this function. Applications may wish to use this function to determine whether a desired feature is supported before initializing the complete SDK.

Parameters

- `Adapter` (D3D11/12 only): Physical adapter which will be used to create the graphics device.
- `Instance` (Vulkan only): `VkInstance` in use by the application.
- `InPhysicalDevice` (Vulkan only): `VkPhysicalDevice` in use by the application.
- `FeatureDiscoveryInfo`: Struct containing information about the feature to query and how it is expected to be initialized, including the feature ID and information that would be provided to **`NVSDK_NGX_Init`**.
- `OutSupported`: Populated with the requirements for the specified feature.

Returns

`NVSDK_NGX_Result` indicating success or failure.

NVSDK_NGX_VULKAN_GetFeatureInstanceExtensionRequirements

Identifies Vulkan instance extensions required for NGX feature support.

```
NVSDK_NGX_Result NVSDK_NGX_VULKAN_GetFeatureInstanceExtensionRequirements(
    const NVSDK_NGX_FeatureDiscoveryInfo *FeatureDiscoveryInfo, uint32_t *OutExtensionCount,
    VkExtensionProperties **OutExtensionProperties);
```

Since: Display Driver Release 525

This utility function identifies the Vulkan instance extensions required for a specific NGX feature, specified by its feature ID. The function populates `OutExtensionCount` with the number of extensions required for the NGX feature. It also populates `OutExtensionProperties` with a pointer to an array of `VkExtensionProperties` structures, which contains details about each required extension. The size of this array is equal to `OutExtensionCount`.

`NVSDK_NGX_Init` does NOT need to be called before calling this function. This function should be called before the VkInstance for the application is created so that the correct extensions can be enabled.

Parameters

- `FeatureDiscoveryInfo`: Struct containing information about the feature to query and how it is expected to be initialized, including the feature ID and information that would be provided to **`NVSDK_NGX_Init`**.
- `OutExtensionCount`: Output pointer that will be populated with the number of required instance extensions.
- `OutExtensionProperties`: Output pointer that will be populated with an array of `VkExtensionProperties` structures required by the given feature.

Returns

`NVSDK_NGX_Result` indicating success or failure.

NVSDK_NGX_VULKAN_GetFeatureDeviceExtensionRequirements

Identifies Vulkan device extensions required for NGX feature support.

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

Since: Display Driver Release 525

This utility function identifies the Vulkan device extensions required for a specific NGX feature, specified by its feature ID, and a given `VkInstance` and `VkPhysicalDevice`. The function populates `OutExtensionCount` with the number of extensions required for the NGX feature. It also populates `OutExtensionProperties` with a pointer to an array of `VkExtensionProperties` structures, which contains details about each required extension. The size of this array is equal to `OutExtensionCount`.

NVSDK_NGX_Init does NOT need to be called before calling this function. This function should be called before the VkDevice for the application is created so that the correct extensions can be enabled.

Parameters

- `Instance`: `VkInstance` in use by the application.
- `InPhysicalDevice`: `VkPhysicalDevice` in use by the application.
- `FeatureDiscoveryInfo`: Struct containing information about the feature to query and how it is expected to be initialized, including the feature ID and information that would be provided to **`NVSDK_NGX_Init`**.
- `OutExtensionCount`: Output pointer that will be populated with the number of required device extensions.
- `OutExtensionProperties`: Output pointer that will be populated with an array of `VkExtensionProperties` structures required by the given feature.

Returns

`NVSDK_NGX_Result` indicating success or failure.

NVSDK_NGX_EvaluateFeature

Evaluates the specified NGX feature using the provided parameters.

```
NVSDK_NGX_Result NVSDK_NGX_D3D11_EvaluateFeature(
    ID3D11DeviceContext *InDevCtx, const NVSDK_NGX_Handle *InFeatureHandle,
    const NVSDK_NGX_Parameter *InParameters, PFN_NVSDK_NGX_ProgressCallback InCallback = NULL);
NVSDK_NGX_Result NVSDK_NGX_D3D12_EvaluateFeature(
    ID3D12GraphicsCommandList *InCmdList, const NVSDK_NGX_Handle *InFeatureHandle,
    const NVSDK_NGX_Parameter *InParameters, PFN_NVSDK_NGX_ProgressCallback InCallback = NULL);
NVSDK_NGX_Result NVSDK_NGX_CUDA_EvaluateFeature(
    const NVSDK_NGX_Handle *InFeatureHandle, const NVSDK_NGX_Parameter *InParameters,
    PFN_NVSDK_NGX_ProgressCallback InCallback = NULL);
NVSDK_NGX_Result NVSDK_NGX_VULKAN_EvaluateFeature(
    VkCommandBuffer InCmdList, const NVSDK_NGX_Handle *InFeatureHandle,
    const NVSDK_NGX_Parameter *InParameters, PFN_NVSDK_NGX_ProgressCallback InCallback = NULL);
```

Evaluates the given feature using the provided parameters. The parameters required to evaluate a feature vary. Refer to the feature's documentation and sample code to determine which input parameters are needed.

To evaluate a feature, an `NVSDK_NGX_Handle` to an instance of that feature, created with **`NVSDK_NGX_CreateFeature`**, must be provided.

For most features, it is beneficial to pass as many input buffers and parameters as possible (e.g., provide all render targets like color, albedo, normals, depth, etc.). This will ensure maximum future compatibility with new updates.

Parameters

- `InCmdList` (D3D12 only): Command list to use to execute GPU commands. Must be:
  - Open and recording.
  - With a node mask including the device provided in **`NVSDK_NGX_D3D12_Init`**.
  - Executed on a non-copy command queue.
- `InDevCtx` (D3D11 only): Device context to use to execute GPU commands. Must be an immediate context. Passing a deferred context is not supported and will cause the API to return an error.
- `InCmdBuffer` (Vulkan only): Command buffer to use to execute GPU commands. Must be open and recording.
- `InFeatureHandle`: Handle representing the feature to be evaluated.
- `InParameters`: List of parameters required to evaluate the feature.

- `InCallback`: Optional callback for features that might take longer to execute. If specified, the SDK will call it with progress values in the range 0.0f - 1.0f. Not all features support progress callbacks. Refer to the documentation for the specific feature to determine whether or not progress callbacks are supported.

Returns

`NVSDK_NGX_Result` indicating success or failure. Common values include:

- **`NVSDK_NGX_Result_Success`** if the function executed successfully.
- **`NVSDK_NGX_Result_FAIL_InvalidParameter`** if any of the input parameters are missing or invalid.
- **`NVSDK_NGX_Result_FAIL_FeatureNotFound`** if a feature with the provided handle could not be found.

Check the NGX logs for additional information about any failures.

Error Codes

Functions in the NGX SDK will return a `NVSDK_NGX_Result` to report the result of the function. The NGX SDK defines the following error codes:

Status Code	Description
`NVSDK_NGX_Result_Success`	The operation completed successfully.
`NVSDK_NGX_Result_Fail`	Generic failure. Check the NGX logs for detailed information.
`NVSDK_NGX_Result_FAIL_FeatureNotSupported`	The NGX SDK or a specific feature is not supported by the current system, hardware, and/or graphics API.
`NVSDK_NGX_Result_FAIL_PlatformError`	An error occurred within the underlying platform, which includes the graphics API in use, the operating system, or other system libraries and dependencies that are not part of the NGX SDK, such as NvAPI. Consult the NGX logs and the graphics API's validation layers for detailed information.
`NVSDK_NGX_Result_FAIL_FeatureAlreadyExists`	The NGX feature could not be created because a feature with identical parameters already exists, and the feature does not support multiple identical instances.
`NVSDK_NGX_Result_FAIL_FeatureNotFound`	A feature associated with the provided handle could not be found.
`NVSDK_NGX_Result_FAIL_InvalidParameter`	One or more provided parameters had an incorrect value or type, or a required parameter was not provided.
`NVSDK_NGX_Result_FAIL_ScratchBufferTooSmall`	The feature requires a scratch buffer, but none was provided or the provided buffer is too small. Use **`NVSDK_NGX_GetScratchBufferSize`** to determine the necessary size.
`NVSDK_NGX_Result_FAIL_NotInitialized`	A function that requires the NGX SDK to be initialized was called before the SDK was properly initialized.
`NVSDK_NGX_Result_FAIL_UnsupportedInputFormat`	One or more input buffers supplied to the feature had an unsupported format.
`NVSDK_NGX_Result_FAIL_RWFlagMissing`	The feature requires read/write access to output buffers, but one or more provided buffers did not have the correct access flags (UAV in D3D11/D3D12).
`NVSDK_NGX_Result_FAIL_MissingInput`	A required input parameter was not provided.

Status Code	Description
NVSDK_NGX_Result_FAIL_UnableToInitializeFeature	The requested feature could not be initialized, likely because the library for that feature could not be found.
NVSDK_NGX_Result_FAIL_OutOfDate	A function was used which requires a newer version of the NVIDIA Display Driver or feature library than is currently installed.
NVSDK_NGX_Result_FAIL_OutOfGPUMemory	An operation could not be completed because the system lacked sufficient GPU memory.
NVSDK_NGX_Result_FAIL_UnsupportedFormat	One or more buffers provided to the feature had an unsupported format.
NVSDK_NGX_Result_FAIL_UnableToWriteToAppDataPath	The SDK does not have the necessary write permissions for the path specified in `InApplicationDataPath`.
NVSDK_NGX_Result_FAIL_UnsupportedParameter	A parameter supplied to the feature is either unsupported by the current version or has an unsupported value.
NVSDK_NGX_Result_FAIL_Denied	NVIDIA has restricted the use of this feature in the current application. Contact NVIDIA for further information.
NVSDK_NGX_Result_FAIL_NotImplemented	The requested feature or functionality has not been implemented in the current version of the NGX SDK, display driver, or feature library.

Error Checking Macros

For convenience, the following macros are provided to check status codes:

Macro Name	Description
NVSDK_NGX_SUCCEED	Boolean `true` if the input value is a `NVSDK_NGX_Result` code indicating success, `false` otherwise.
NVSDK_NGX_FAILED	Boolean `true` if the input value is a `NVSDK_NGX_Result` code indicating failure, `false` otherwise.

NVSDK_NGX_Result Codes

NVSDK_NGX_Result_Success

```
enum NVSDK_NGX_Result::NVSDK_NGX_Result_Success = 0x1
```

The operation completed successfully.

NVSDK_NGX_Result_Fail

```
enum NVSDK_NGX_Result::NVSDK_NGX_Result_Fail = 0xBAD00000
```

Generic failure. Check the NGX logs for detailed information.

NVSDK_NGX_Result_FAIL_FeatureNotSupported

```
enum NVSDK_NGX_Result::NVSDK_NGX_Result_FAIL_FeatureNotSupported = 0xBAD00001
```

The NGX SDK or a specific feature is not supported by the current system, hardware, and/or graphics API.

NVSDK_NGX_Result_FAIL_PlatformError

```
enum NVSDK_NGX_Result::NVSDK_NGX_Result_FAIL_PlatformError = 0xBAD00002
```

An error occurred within the underlying platform, which includes the graphics API in use, the operating system, or other system libraries and dependencies that are not part of the NGX SDK, such as NvAPI. Consult the NGX logs and the graphics API's validation layers for detailed information.

NVSDK_NGX_Result_FAIL_FeatureAlreadyExists

```
enum NVSDK_NGX_Result::NVSDK_NGX_Result_FAIL_FeatureAlreadyExists = 0xBAD00003
```

The NGX feature could not be created because a feature with identical parameters already exists, and the feature does not support multiple identical instances.

NVSDK_NGX_Result_FAIL_FeatureNotFound

```
enum NVSDK_NGX_Result::NVSDK_NGX_Result_FAIL_FeatureNotFound = 0xBAD00004
```

A feature associated with the provided handle could not be found.

NVSDK_NGX_Result_FAIL_InvalidParameter

```
enum NVSDK_NGX_Result::NVSDK_NGX_Result_FAIL_InvalidParameter = 0xBAD00005
```

One or more provided parameters had an incorrect value or type, or a required parameter was not provided.

NVSDK_NGX_Result_FAIL_ScratchBufferTooSmall

```
enum NVSDK_NGX_Result::NVSDK_NGX_Result_FAIL_ScratchBufferTooSmall = 0xBAD00006
```

The feature requires a scratch buffer, but none was provided or the provided buffer is too small. Use **`NVSDK_NGX_GetScratchBufferSize`** to determine the necessary size.

NVSDK_NGX_Result_FAIL_NotInitialized

```
enum NVSDK_NGX_Result::NVSDK_NGX_Result_FAIL_NotInitialized = 0xBAD00007
```

A function that requires the NGX SDK to be initialized was called before the SDK was properly initialized.

NVSDK_NGX_Result_FAIL_UnsupportedInputFormat

```
enum NVSDK_NGX_Result::NVSDK_NGX_Result_FAIL_UnsupportedInputFormat = 0xBAD00008
```

One or more input buffers supplied to the feature had an unsupported format.

NVSDK_NGX_Result_FAIL_RWFlagMissing

```
enum NVSDK_NGX_Result::NVSDK_NGX_Result_FAIL_RWFlagMissing = 0xBAD00009
```

The feature requires read/write access to output buffers, but one or more provided buffers did not have the correct access flags (UAV in D3D11/D3D12).

NVSDK_NGX_Result_FAIL_MissingInput

```
enum NVSDK_NGX_Result::NVSDK_NGX_Result_FAIL_MissingInput = 0xBAD0000A
```

A required input parameter was not provided.

NVSDK_NGX_Result_FAIL_UnableToInitializeFeature

```
enum NVSDK_NGX_Result::NVSDK_NGX_Result_FAIL_UnableToInitializeFeature = 0xBAD0000B
```

The requested feature could not be initialized, likely because the library for that feature could not be found.

NVSDK_NGX_Result_FAIL_OutOfDate

```
enum NVSDK_NGX_Result::NVSDK_NGX_Result_FAIL_OutOfDate = 0xBAD0000C
```

A function was used which requires a newer version of the NVIDIA Display Driver or feature library than is currently installed.

NVSDK_NGX_Result_FAIL_OutOfGPUMemory

```
enum NVSDK_NGX_Result::NVSDK_NGX_Result_FAIL_OutOfGPUMemory = 0xBAD0000D
```

An operation could not be completed because the system lacked sufficient GPU memory.

NVSDK_NGX_Result_FAIL_UnsupportedFormat

```
enum NVSDK_NGX_Result::NVSDK_NGX_Result_FAIL_UnsupportedFormat = 0xBAD0000E
```

One or more buffers provided to the feature had an unsupported format.

NVSDK_NGX_Result_FAIL_UnableToWriteToAppDataPath

```
enum NVSDK_NGX_Result::NVSDK_NGX_Result_FAIL_UnableToWriteToAppDataPath = 0xBAD0000F
```

The SDK does not have the necessary write permissions for the path specified in InApplicationDataPath.

NVSDK_NGX_Result_FAIL_UnsupportedParameter

```
enum NVSDK_NGX_Result::NVSDK_NGX_Result_FAIL_UnsupportedParameter = 0xBAD00010
```

A parameter supplied to the feature is either unsupported by the current version or has an unsupported value.

NVSDK_NGX_Result_FAIL_Denied

```
enum NVSDK_NGX_Result::NVSDK_NGX_Result_FAIL_Denied = 0xBAD00011
```

NVIDIA has restricted the use of this feature in the current application. Contact NVIDIA for further information.

NVSDK_NGX_Result_FAIL_NotImplemented

```
enum NVSDK_NGX_Result::NVSDK_NGX_Result_FAIL_NotImplemented = 0xBAD00012
```

The requested feature or functionality has not been implemented in the current version of the NGX SDK, display driver, or feature library.

Result Checking Macros

NVSDK_NGX_SUCCEED

```
#define NVSDK_NGX_SUCCEED(value) (((value) & 0xFFF00000) != NVSDK_NGX_Result_Fail)
```

Boolean `true` if `value` is a `NVSDK_NGX_Result` code indicating success, `false` otherwise.

NVSDK_NGX_FAILED

```
#define NVSDK_NGX_FAILED(value) (((value) & 0xFFF00000) == NVSDK_NGX_Result_Fail)
```

Boolean `true` if `value` is a `NVSDK_NGX_Result` code indicating failure, `false` otherwise.

DLSS-FG Parameters

The following parameters are used for DLSS-FG:

Creation Parameters

These parameters can be provided to DLSS-FG when creating the feature using **`NVSDK_NGX_CreateFeature`**.

Parameter	Description
`NVSDK_NGX_DLSSG_Parameter_Width`	The width of the input and output color resources.
`NVSDK_NGX_DLSSG_Parameter_Height`	The height of the input and output color resources.
`NVSDK_NGX_DLSSG_Parameter_BackbufferFormat`	The format of the backbuffer.
`NVSDK_NGX_DLSSG_Parameter_InternalHeight`	The internal height resolution used for processing.
`NVSDK_NGX_DLSSG_Parameter_InternalWidth`	The internal width resolution used for processing.
`NVSDK_NGX_DLSSG_Parameter_DynamicResolution`	Whether or not dynamic resolution scaling is enabled.
`NVSDK_NGX_DLSSG_Parameter_AsyncCreateEnabled`	Whether or not the feature is allowed to asynchronously allocate resources.
`NVSDK_NGX_Parameter_CreationNodeMask`	The node mask used for resource creation. Specifies which GPU nodes can create resources.
`NVSDK_NGX_Parameter_VisibilityNodeMask`	The node mask used for resource visibility. Specifies which GPU nodes can access resources.
`NVSDK_NGX_Parameter_ResourceAllocCallback`	Callback function for resource allocation.
`NVSDK_NGX_Parameter_ResourceReleaseCallback`	Callback function for resource release.

Evaluate Parameters

These parameters can be provided to DLSS-FG when evaluating the feature using **`NVSDK_NGX_EvaluateFeature`**.

Resource Parameters

Parameter	Description
NVSDK_NGX_DLSSG_Parameter_Backbuffer	Input color resource.
NVSDK_NGX_DLSSG_Parameter_MVecs	Input motion vectors resource.
NVSDK_NGX_DLSSG_Parameter_Depth	Input depth buffer resource.
NVSDK_NGX_DLSSG_Parameter_HUDLess	Input HUDLess color resource.
NVSDK_NGX_DLSSG_Parameter_UI	Input resource containing UI color and alpha.
NVSDK_NGX_DLSSG_Parameter_Bidirectional_Distortion_Field	Input distortion field.
NVSDK_NGX_DLSSG_Parameter_NoPostProcessingColor	Input resource for color before post-processing.
NVSDK_NGX_Parameter_OutputInterpolated	Output resource to write the interpolated frame to.
NVSDK_NGX_Parameter_OutputReal	Output resource to write the real frame to.
NVSDK_NGX_DLSSG_Parameter_OutputDisableInterpolation	Output resource indicating whether or not interpolation was disabled for the current frame.

Engine Data and Settings

Parameter	Description
NVSDK_NGX_DLSSG_Parameter_Reset	If set, resets the internal state of the feature.
NVSDK_NGX_DLSSG_Parameter_ClipToPrevClip	Transformation matrix from the current frame's clip space to the previous frame's clip space.
NVSDK_NGX_DLSSG_Parameter_PrevClipToClip	Transformation matrix from the previous frame's clip space to the current frame's clip space.
NVSDK_NGX_DLSSG_Parameter_MvecScaleX	Scale factor applied to motion vectors in the X direction.
NVSDK_NGX_DLSSG_Parameter_MvecScaleY	Scale factor applied to motion vectors in the Y direction.

Parameter	Description
NVSDK_NGX_DLSSG_Parameter_CameraPosX	X-coordinate of the camera position.
NVSDK_NGX_DLSSG_Parameter_CameraPosY	Y-coordinate of the camera position.
NVSDK_NGX_DLSSG_Parameter_CameraPosZ	Z-coordinate of the camera position.
NVSDK_NGX_DLSSG_Parameter_CameraUpX	X-coordinate of the camera's up vector.
NVSDK_NGX_DLSSG_Parameter_CameraUpY	Y-coordinate of the camera's up vector.
NVSDK_NGX_DLSSG_Parameter_CameraUpZ	Z-coordinate of the camera's up vector.
NVSDK_NGX_DLSSG_Parameter_CameraRightX	X-coordinate of the camera's right vector.
NVSDK_NGX_DLSSG_Parameter_CameraRightY	Y-coordinate of the camera's right vector.
NVSDK_NGX_DLSSG_Parameter_CameraRightZ	Z-coordinate of the camera's right vector.
NVSDK_NGX_DLSSG_Parameter_CameraFwdX	X-coordinate of the camera's forward vector.
NVSDK_NGX_DLSSG_Parameter_CameraFwdY	Y-coordinate of the camera's forward vector.
NVSDK_NGX_DLSSG_Parameter_CameraFwdZ	Z-coordinate of the camera's forward vector.
NVSDK_NGX_DLSSG_Parameter_CameraNear	Near clipping plane distance.
NVSDK_NGX_DLSSG_Parameter_CameraFar	Far clipping plane distance.
NVSDK_NGX_DLSSG_Parameter_DepthInverted	Whether or not the depth buffer is inverted.
NVSDK_NGX_DLSSG_Parameter_LinearizedDepth_Scale	Optional scalar on the linearized depth before algorithm processing.

Parameter	Description
NVSDK_NGX_DLSSG_Parameter_LinearizedDepth_NearFarPartition	Optional heuristic that defines the Z-distance to the depth plane that artificially partitions near from far objects.
NVSDK_NGX_DLSSG_Parameter_MinRelativeLinearDepthObjectSeparation	Minimum relative linear depth object separation.
NVSDK_NGX_DLSSG_Parameter_Bidirectional_Distortion_Field_LowPrecision_IsLowPrecision	Indicates if the bidirectional distortion field is in low precision.
NVSDK_NGX_DLSSG_Parameter_Bidirectional_Distortion_Field_LowPrecision_Bias	Bias for low precision bidirectional distortion field.
NVSDK_NGX_DLSSG_Parameter_Bidirectional_Distortion_Field_LowPrecision_Scale	Scale for low precision bidirectional distortion field.
NVSDK_NGX_DLSSG_Parameter_EvalFlags	Bitfield containing flags that control various properties of the algorithm.
NVSDK_NGX_DLSSG_Parameter_MultiFrameIndex	Index (starting at 1) of the frame to generate for multi-frame generation.
NVSDK_NGX_DLSSG_Parameter_MultiFrameCount	The number of intermediate frames to generate between each pair of rendered input frames.
NVSDK_NGX_DLSSG_Parameter_UserDebugText	User-provided debug text which will be displayed on screen in non-production builds.
NVSDK_NGX_DLSSG_Parameter_CameraViewToClip	Transformation matrix from camera view space to clip space.
NVSDK_NGX_DLSSG_Parameter_ClipToCameraView	Transformation matrix from clip space to view space.
NVSDK_NGX_DLSSG_Parameter_ClipToLensClip	Transformation matrix describing lens distortion in clip space.
NVSDK_NGX_DLSSG_Parameter_JitterOffsetX	Jitter offset in the X direction.
NVSDK_NGX_DLSSG_Parameter_JitterOffsetY	Jitter offset in the Y direction.
NVSDK_NGX_DLSSG_Parameter_CameraPinholeOffsetX	Pinhole offset in the X direction for the camera.

Parameter	Description
NVSDK_NGX_DLSSG_Parameter_CameraPinholeOffsetY	Pinhole offset in the Y direction for the camera.
NVSDK_NGX_DLSSG_Parameter_CameraFOV	Field of view of the camera, in degrees or radians.
NVSDK_NGX_DLSSG_Parameter_CameraAspectRatio	Aspect ratio of the camera.
NVSDK_NGX_DLSSG_Parameter_ColorBuffersHDR	Indicates if the color buffers use HDR.
NVSDK_NGX_DLSSG_Parameter_CameraMotionIncluded	Indicates if camera motion is included in the input motion vector field.
NVSDK_NGX_DLSSG_Parameter_NotRenderingGameFrames	Indicates if the frames currently being rendered by the game are not "game frames".
NVSDK_NGX_DLSSG_Parameter_OrthoProjection	Indicates if orthographic projection is used by the camera.
NVSDK_NGX_DLSSG_Parameter_MvecInvalidValue	Invalid value for motion vectors.
NVSDK_NGX_DLSSG_Parameter_MvecDilated	Indicates if motion vectors are dilated.
NVSDK_NGX_DLSSG_Parameter_MvecJittered	Indicates if motion vectors contain jitter information.
NVSDK_NGX_DLSSG_Parameter_MenuDetectionEnabled	Enables full-screen menu detection.

Resource Subrect Parameters

Parameter	Description
NVSDK_NGX_DLSSG_Parameter_BackbufferSubrectBaseX	X-coordinate of the base of the backbuffer subrect.
NVSDK_NGX_DLSSG_Parameter_BackbufferSubrectBaseY	Y-coordinate of the base of the backbuffer subrect.
NVSDK_NGX_DLSSG_Parameter_BackbufferSubrectWidth	Width of the backbuffer subrect.
NVSDK_NGX_DLSSG_Parameter_BackbufferSubrectHeight	Height of the backbuffer subrect.

Parameter	Description
NVSDK_NGX_DLSSG_Parameter_MVecsSubrectBaseX	X-coordinate of the base of the motion vectors subrect.
NVSDK_NGX_DLSSG_Parameter_MVecsSubrectBaseY	Y-coordinate of the base of the motion vectors subrect.
NVSDK_NGX_DLSSG_Parameter_MVecsSubrectWidth	Width of the motion vectors subrect.
NVSDK_NGX_DLSSG_Parameter_MVecsSubrectHeight	Height of the motion vectors subrect.
NVSDK_NGX_DLSSG_Parameter_DepthSubrectBaseX	X-coordinate of the base of the depth subrect.
NVSDK_NGX_DLSSG_Parameter_DepthSubrectBaseY	Y-coordinate of the base of the depth subrect.
NVSDK_NGX_DLSSG_Parameter_DepthSubrectWidth	Width of the depth subrect.
NVSDK_NGX_DLSSG_Parameter_DepthSubrectHeight	Height of the depth subrect.
NVSDK_NGX_DLSSG_Parameter_HUDLessSubrectBaseX	X-coordinate of the base of the HUDLess subrect.
NVSDK_NGX_DLSSG_Parameter_HUDLessSubrectBaseY	Y-coordinate of the base of the HUDLess subrect.
NVSDK_NGX_DLSSG_Parameter_HUDLessSubrectWidth	Width of the HUDLess subrect.
NVSDK_NGX_DLSSG_Parameter_HUDLessSubrectHeight	Height of the HUDLess subrect.
NVSDK_NGX_DLSSG_Parameter_UISubrectBaseX	X-coordinate of the base of the UI subrect.
NVSDK_NGX_DLSSG_Parameter_UISubrectBaseY	Y-coordinate of the base of the UI subrect.
NVSDK_NGX_DLSSG_Parameter_UISubrectWidth	Width of the UI subrect.
NVSDK_NGX_DLSSG_Parameter_UISubrectHeight	Height of the UI subrect.
NVSDK_NGX_DLSSG_Parameter_BidirectionalDistortionFieldSubrectBaseX	X-coordinate of the base of the bidirectional distortion field subrect.
NVSDK_NGX_DLSSG_Parameter_BidirectionalDistortionFieldSubrectBaseY	Y-coordinate of the base of the bidirectional distortion field subrect.

Parameter	Description
NVSDK_NGX_DLSSG_Parameter_BidirectionalDistortionFieldSubrectWidth	Width of the bidirectional distortion field subrect.
NVSDK_NGX_DLSSG_Parameter_BidirectionalDistortionFieldSubrectHeight	Height of the bidirectional distortion field subrect.

Capability Parameters

The following parameters will be pre-populated in parameter maps returned by `NVSDK_NGX_GetCapabilityParameters`.

Parameter	Description
NVSDK_NGX_Parameter_FrameGeneration_Available	Whether or not frame generation is available on the current system.
NVSDK_NGX_Parameter_FrameGeneration_FeatureInitResult	If frame generation could not be initialized successfully, the `NVSDK_NGX_Result` indicating the reason for failure.
NVSDK_NGX_Parameter_FrameGeneration_MinDriverVersionMajor	Major version number of the minimum display driver required for frame generation.
NVSDK_NGX_Parameter_FrameGeneration_MinDriverVersionMinor	Minor version number of the minimum display driver required for frame generation.
NVSDK_NGX_Parameter_FrameGeneration_NeedsUpdatedDriver	If frame generation is not available, whether or not an updated driver is needed for frame generation.
NVSDK_NGX_DLSSG_Parameter_MultiFrameCountMax	Maximum number of interpolated frames supported for multi-frame generation.
NVSDK_NGX_Parameter_DLSSGGetCurrentSettingsCallback	Callback for getting current DLSSG settings.
NVSDK_NGX_Parameter_DLSSGEstimateVRAMCallback	Callback for estimating VRAM usage.

Parameter Documentation

NVSDK_NGX_DLSSG_Parameter_Width

The width of the input and output color resources.

Type: `unsigned int`

Specifies the width used when creating the feature, which determines the dimensions of both the input and output color resources. When you call Evaluate, the sizes of these resources (including OutputInterpolated and OutputReal if used) must match the width specified when you created the feature. You can't change the width after creating the feature; to use a different resolution, you must first destroy the feature using **`NVSDK_NGX_ReleaseFeature`** and then recreate it with the new dimensions.

This value must be between 4 and 8192 px (inclusive).

NVSDK_NGX_DLSSG_Parameter_Height

The height of the input and output color resources.

Type: `unsigned int`

Specifies the height used when creating the feature, which determines the dimensions of both the input and output color resources. When you call Evaluate, the sizes of these resources (including OutputInterpolated and OutputReal if used) must match the height specified when you created the feature. You can't change the height after creating the feature; to use a different resolution, you must first destroy the feature using **`NVSDK_NGX_ReleaseFeature`** and then recreate it with the new dimensions.

This value must be between 4 and 7680 px (inclusive).

NVSDK_NGX_DLSSG_Parameter_BackbufferFormat

The format of the backbuffer.

Type: `unsigned int`

Defines the format of your input and output color buffers, as a value from your graphics API's format enumeration (e.g., DXGI_FORMAT for D3D12 or VkFormat for Vulkan) expressed as an unsigned int. The format must have at least three channels that contain floating-point data (FLOAT, UNORM, or SNORM). For example, you might use `DXGI_FORMAT_R8G8B8A8_UNORM` in D3D12, or `VK_FORMAT_R8G8B8A8_UNORM` in Vulkan.

NVSDK_NGX_DLSSG_Parameter_DynamicResolution

Whether or not dynamic resolution scaling is enabled.

Type: Boolean as `unsigned int` (0 or 1)

Set this to true if your application uses dynamic resolution scaling. When enabled, the dimensions of your depth and motion vector resources may change between EvaluateFeature calls. In this mode, NGX ignores the **`NVSDK_NGX_DLSSG_Parameter_InternalWidth/Height`** parameters.

NVSDK_NGX_DLSSG_Parameter_InternalWidth

The internal width resolution used for processing.

Type: `unsigned int`

Specifies the width of your internal rendering resolution, also known as the Render Width. This is the expected width for the depth and motion vectors resources. If you are using an upscaler such as DLSS Super Resolution, this value should be the

input resolution to DLSS Super Resolution. When dynamic resolution scaling is not enabled, your depth and motion vector resources must match the Render Width and Height specified during feature creation.

NVSDK_NGX_DLSSG_Parameter_InternalHeight

The internal height resolution used for processing.

Type: `unsigned int`

Specifies the height of your internal rendering resolution, also known as the Render Height. This is the expected height for the depth and motion vectors resources. If you are using an upscaler such as DLSS Super Resolution, this value should be the input resolution to DLSS Super Resolution. When dynamic resolution scaling is not enabled, your depth and motion vector resources must match the Render Width and Height specified during feature creation.

NVSDK_NGX_DLSSG_Parameter_AsyncCreateEnabled

EXPERIMENTAL: Whether or not the feature is allowed to asynchronously allocate resources.

Type: Boolean as `unsigned int` (0 or 1)

Optional. Default: false (0).

When enabled, the feature can allocate its internal resources asynchronously on a separate thread. This helps eliminate stuttering that may occur when you first enable the feature. While resources are being allocated, any calls to **`NVSDK_NGX_EvaluateFeature`** will copy the input color to the output. This feature is disabled by default.

NVSDK_NGX_Parameter_CreationNodeMask

The node mask used for resource creation. Specifies which GPU nodes can create resources.

Type: `unsigned int`

The node mask used to specify which GPU nodes can create resources. For multi-GPU systems, this mask determines which GPUs are allowed to allocate and create the internal resources needed by DLSS-FG.

NVSDK_NGX_Parameter_VisibilityNodeMask

The node mask used for resource visibility. Specifies which GPU nodes can access resources.

Type: `unsigned int`

The node mask used to specify which GPU nodes can access resources. For multi-GPU systems, this mask determines which GPUs are allowed to access and use the internal resources allocated by DLSS-FG.

NVSDK_NGX_Parameter_ResourceAllocCallback

A callback function to allocate internal resources.

Type: **`PFN_NVSDK_NGX_D3D12_ResourceAllocCallback`** as `void*`

Optional. D3D12 Only

Pointer to a callback function which will be called by the feature to request a resource for its internal use. Providing a callback function allows your application to manage the creation of resources that would otherwise be managed internally by the feature. If provided, your implementation must allocate and return a new D3D12 resource based on the provided parameters.

If you provide an allocation callback, you must provide a corresponding release callback via the **NVSDK_NGX_Parameter_ResourceReleaseCallback** parameter.

See **PFN_NVSDK_NGX_D3D12_ResourceAllocCallback** for details on the parameters to this function.

NVSDK_NGX_Parameter_ResourceReleaseCallback

A callback function to release resources.

Type: **PFN_NVSDK_NGX_ResourceReleaseCallback** as `void*`

Optional. D3D12 Only

Pointer to a callback function which will be called by the feature to release a resource that was previously allocated via the **NVSDK_NGX_Parameter_ResourceAllocCallback** callback. If provided, your callback implementation must release the provided resource.

If you provide a release callback, you must also provide an allocation callback via the **NVSDK_NGX_Parameter_ResourceAllocCallback** parameter.

See **PFN_NVSDK_NGX_ResourceReleaseCallback** for details on the parameters to this function.

NVSDK_NGX_DLSSG_Parameter_Backbuffer

The input color resource.

Type: Resource

The input color resource. DLSS-FG will interpolate between this resource and the backbuffer from the previous frame. In most cases, you should provide the same resource that you would present if DLSS-FG were not enabled, including any post-processing effects, UI, etc. in use by your game.

The format of this resource must match the format of **`NVSDK_NGX_Parameter_OutputInterpolated`**. It must have at least three channels containing floating-point data (FLOAT, UNORM, or SNORM), such as `DXGI_FORMAT_R8G8B8A8_UNORM` in D3D12, or `VK_FORMAT_R8G8B8A8_UNORM` using Vulkan.

You may specify the backbuffer color as a subrect of a larger resource. See `NVSDK_NGX_DLSSG_Parameter_BackbufferSubrectBaseX/BaseY/Width/Height`.

This resource's resolution, or the resolution of its subrect if used, must match the width and height used when you created the feature.

The provided resource is assumed to be in a state suitable for reading. For D3D12, the resource must be in state `D3D12_RESOURCE_STATE_NON_PIXEL_SHADER_RESOURCE` before calling **`NVSDK_NGX_EvaluateFeature`**.

Note on using motion blur: if motion blur is enabled, you should reduce the magnitude of the effect by half for single-frame generation, or by a factor equal to the selected multiplier for DLSS-MFG (e.g., 1/4 for 4x).

NVSDK_NGX_DLSSG_Parameter_MVecs

The input motion vectors resource.

Type: Resource

This resource contains motion vectors indicating the per-pixel, screen-space motion from the current frame to the previous frame. Provide this resource in a two-channel floating-point format such as `RG32_FLOAT`, where the red channel represents motion in the X-axis, and the green channel represents motion on the Y-axis. Though it is not recommended, you may use a format with additional channels, which will be ignored.

Motion vectors map a pixel from the current frame to its position in the previous frame. In other words, when the motion vector for a particular pixel is added to the pixel's position, the result is the location the pixel occupied in the previous frame.

The input motion vectors must satisfy the following properties:

- Motion is expressed in units of screen-space pixels *at the resolution of the motion vector resource*.
    - Screen space pixels define (0, 0) as the upper-left corner of the screen, with positive X extending to the right of the screen, and positve Y extending down. For example, if the motion vector resource is 1080p, the bottom-right pixel would be (1919, 1079).

    - For example, for a 1080p motion vector resource, if an object moves from the top-right corner of the screen to the center, the motion vector at the center of the screen would have a value of (960, -540).
    - The parameters **NVSDK_NGX_DLSSG_Parameter_MvecScaleX/Y** can be used to rescale motion vectors which are provided at a different scale, and/or to invert the motion on one or both axes.
- Motion vectors may be positive or negative
- Motion vectors may include partial pixel movements. For example, a motion vector of (-0.65, 10.1) is valid.

Motion vectors may be specified as a subrect of a larger resource. See `NVSDK_NGX_DLSSG_Parameter_MVecsSubrectBaseX/BaseY/Width/Height`.

The resolution of your motion vector resource (or its subrect, if used) should match the internal width and height you provided when creating the feature, unless dynamic resolution scaling is enabled. While its resolution doesn't need to match the backbuffer's, ensure both are spatially aligned: corresponding pixels in UV space should represent the same point in the image.

If an upscaler such as DLSS-SR is in use, this resource should be the same motion vector resource that was provided to the upscaler.

The provided resource is assumed to be in a state suitable for reading. For D3D12, the resource must be in state `D3D12_RESOURCE_STATE_NON_PIXEL_SHADER_RESOURCE` before calling **NVSDK_NGX_EvaluateFeature**.

NVSDK_NGX_DLSSG_Parameter_Depth

The input depth buffer resource.

Type: Resource

The depth buffer for the current frame, specified in hardware units (linear depth units are not supported). Depth may be inverted (1 is near, 0 is far) or non-inverted (0 is near, 1 is far); set the parameter **NVSDK_NGX_DLSSG_Parameter_DepthInverted** accordingly.

You may specify the depth buffer as a subrect of a larger resource. See `NVSDK_NGX_DLSSG_Parameter_DepthSubrectBaseX/BaseY/Width/Height`.

The resolution of your depth buffer (or its subrect, if used) should match the internal width and height you provided when creating the feature, unless dynamic resolution scaling is enabled. While its resolution doesn't need to match the

backbuffer's, ensure both are spatially aligned: corresponding pixels in UV space should represent the same point in the image.

The provided resource is assumed to be in a state suitable for reading. For D3D12, the resource must be in state `D3D12_RESOURCE_STATE_NON_PIXEL_SHADER_RESOURCE` before calling **`NVSDK_NGX_EvaluateFeature`**.

NVSDK_NGX_DLSSG_Parameter_HUDLess

The input HUDLess color resource.

Type: Resource

Optional.

The HUDLess resource contains the scene color before any UI or HUD elements are drawn. Its content for non-UI pixels should be identical to the backbuffer. Therefore, you must apply any tonemapping or other post-processing effects used to render the backbuffer equally to HUDLess.

The HUDLess texture is used as a guide for the DLSS-FG algorithm to improve the interpolation of UI and nearby regions. Providing HUDLess is optional, but highly recommended. If you do not provide a HUDLess texture, you may see additional artifacts in UI regions.

The HUDLess resource resolution should match the backbuffer resolution. While the algorithm supports HUDLess textures with different resolutions by resampling them to match the backbuffer, this is not recommended since the resampling process can degrade quality and negatively impact the algorithm's output.

HUDLess may be specified as a subrect of a larger resource. See `NVSDK_NGX_DLSSG_Parameter_HUDLessSubrectBaseX/BaseY/Width/Height`.

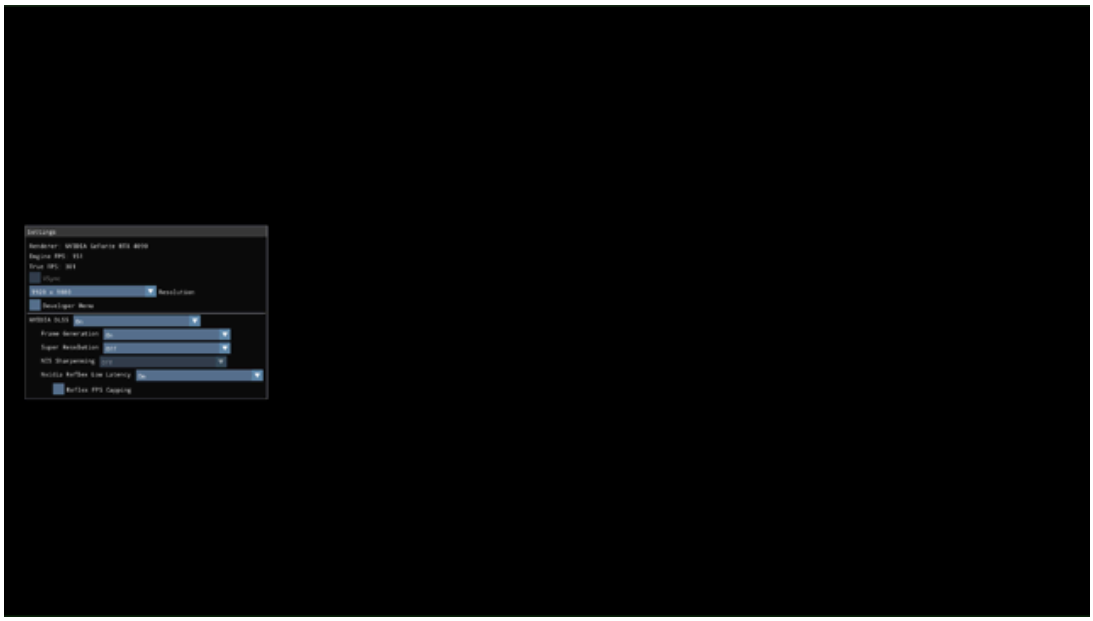
The provided resource is assumed to be in a state suitable for reading. For D3D12, the resource must be in state `D3D12_RESOURCE_STATE_NON_PIXEL_SHADER_RESOURCE` before calling **`NVSDK_NGX_EvaluateFeature`**.

NVSDK_NGX_DLSSG_Parameter_UI

The input resource containing UI color and alpha.

Type: Resource

Optional.



A four-channel resource containing the color and opacity/alpha of the UI; may also be referred to as "UIAlpha". The color should be pre-multiplied by the alpha value, so that the following formula holds as closely as possible for any particular pixel:

$$\text{Backbuffer.RGB} = \text{UI.RGB} + (1 - \text{UI.}\alpha) \cdot \text{Hudless.RGB}$$

Any pixel which is not UI should have a value of 0.

The UI texture is used as a guide for the DLSS-FG algorithm to improve the interpolation of UI and nearby regions. Providing a UI texture is optional, but highly recommended. If you do not provide UI information, you may see additional artifacts in UI regions.

For best results, provide a UI texture which has sufficient precision in all four channels, including the alpha channel. Formats such as RGB10A2 are not appropriate for UI.

The UI resource's resolution should match the backbuffer resolution. While the algorithm supports UI textures with different resolutions by resampling them to match the backbuffer, this is not recommended since the resampling process can degrade quality and negatively impact the algorithm's output.

HUDLess may be specified as a subrect of a larger resource. See `NVSDK_NGX_DLSSG_Parameter_UISubrectBaseX/BaseY/Width/Height`.

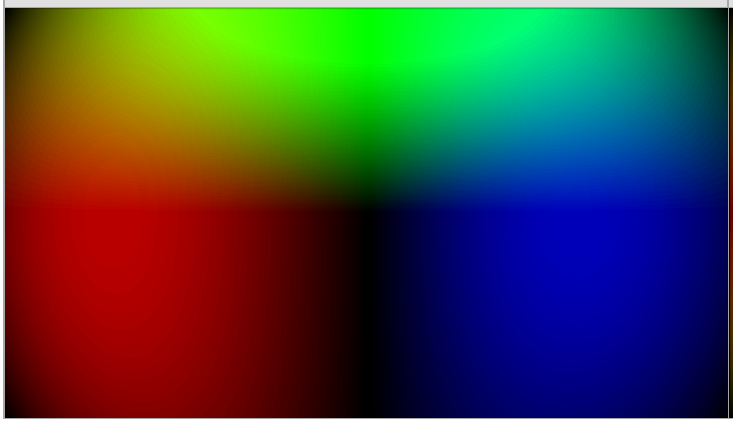
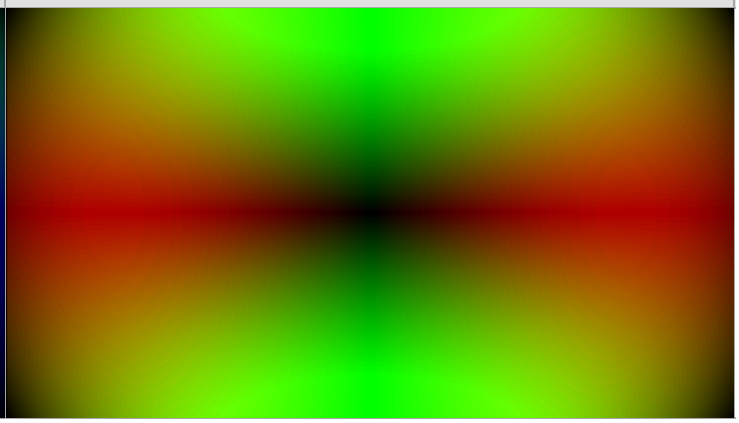
The provided resource is assumed to be in a state suitable for reading. For D3D12, the resource must be in state `D3D12_RESOURCE_STATE_NON_PIXEL_SHADER_RESOURCE` before calling **`NVSDK_NGX_EvaluateFeature`**.

NVSDK_NGX_DLSSG_Parameter_Bidirectional_Distortion_Field

The input distortion field.

Type: Resource

Optional. Use only if distortion effects, such as lens distortion, have been applied as post-processing filters.

Barrel Distortion, RGB channels	Barrel Distortion, absolute value of RG channels
	

Resource describing post-processing distortion effects, such as lens distortion, that have been applied to the final color. If you use this type of effect, the final color is no longer aligned with the input depth, motion vectors, and camera information. The bidirectional distortion field serves as a mapping between the pixels in the distorted color texture and the undistorted guide buffers.

When using the bidirectional distortion field:

- Motion vectors, depth, and camera information are in **undistorted space**.
- All color data is in **distorted space**
    - If HUDLess is provided, the distortion must be applied to the HUDLess texture as well.
    - If a UI texture is provided, any distortion applied to the UI/HUD must be applied equally in both the UI texture and the backbuffer color.
    - The relationship between the backbuffer, UI, and HUDLess color at each pixel must still hold, where the UI blends over the HUDLess color according to its alpha value, regardless of any distortion applied.

The bidirectional distortion field is a four-channel resource containing both the mapping from distorted space to undistorted space in the red and green channels, and the mapping from undistorted space to distorted space in the blue and alpha channels. Each mapping describes the offset, on the X and Y axes, that needs to be applied to a pixel in one space to reach its position in the other.

All values are in normalized pixel space, relative to the size of the screen. In normalized space, (0, 0) is defined as the top-left corner of the screen, and (1, 1) is the bottom-right. Values may be positive or negative, and may represent fractional pixel distances.

For example, consider a distortion where the pixel that was at the top left corner of the screen before the distortion was applied is moved to the center of the screen after applying distortion. To go from the distorted position to the undistorted position, the pixel needs to move half of the screen width to the left and half of the screen height up. Therefore, the value of the RG channels of the distortion field in the center pixel should be (-0.5, -0.5). For the reverse mapping, to go from the undistorted position to the distorted position, the pixel moves half of the screen right and down, so the value of the BA channels *at the top-left pixel* should be (0.5, 0.5).

Sample code for generating a bidirectional distortion field, as well as estimating the reverse mapping from the forward mapping can be found in **the Lens Distortion guide**.

The recommended resolution of the bidirectional distortion field is half of the width and height of the backbuffer. However, the resource may be any resolution as long as it is spatially aligned with the appropriate distorted and undistorted textures. Use of a smaller resource may improve performance at the possible expense of accuracy.

The distortion field may be specified as a subrect of a larger resource. See `NVSDK_NGX_DLSSG_Parameter_BidirectionalDistortionFieldSubrectBaseX/BaseY/Width/Height`.

To ensure that the data can be read with sufficient precision, use a format that has at least 8 bits for all four channels. Formats such as RGB10A2 will yield poor results and should not be used.

The provided resource is assumed to be in a state suitable for reading. For D3D12, the resource must be in state `D3D12_RESOURCE_STATE_NON_PIXEL_SHADER_RESOURCE` before calling **`NVSDK_NGX_EvaluateFeature`**.

NVSDK_NGX_DLSSG_Parameter_NoPostProcessingColor

The input resource for color before post-processing.

Type: Resource

Optional, currently unused.

The color buffer before any post-processing effects are applied. This resource is currently not used by the algorithm, but applications which have this data available may wish to provide it to ensure future compatibility with later versions of the algorithm.

NoPostProcessingColor may be specified as a subrect of a larger resource. The subrect extent is the same as the subrect defined for the backbuffer.

The provided resource is assumed to be in a state suitable for reading. For D3D12, the resource must be in state `D3D12_RESOURCE_STATE_NON_PIXEL_SHADER_RESOURCE` before calling **`NVSDK_NGX_EvaluateFeature`**.

NVSDK_NGX_Parameter_OutputInterpolated

The output resource to write the interpolated frame to.

Type: Resource

The algorithm will write the interpolated frame to this resource. The resource must be in a format with at least three channels containing floating-point data (FLOAT, UNORM, or SNORM). For example, this could be `DXGI_FORMAT_R8G8B8A8_UNORM` in D3D12, or `VK_FORMAT_R8G8B8A8_UNORM` using Vulkan. This format must also match the format of the input backbuffer color.

You may specify that the output be written to a subrect of a larger resource. The subrect extent is the same as the subrect defined for the backbuffer. The resolution of the underlying resource must be the same as the resolution of the backbuffer unless the flag **`NVSDK_NGX_DLSSG_EvalFlags_UpdateOnlyInsideExtents`** is set.

This resource's resolution, or the resolution of its subrect if used, must match the width and height used when creating the feature.

The provided resource is assumed to be in a state suitable for writing. For D3D12, the resource must be in state `D3D12_RESOURCE_STATE_UNORDERED_ACCESS` before calling **`NVSDK_NGX_EvaluateFeature`**.

NVSDK_NGX_Parameter_OutputReal

The output resource to write the real frame to.

Type: Resource

Optional.

The algorithm will write the game-rendered frame to this resource. In most cases, this is a copy of the input backbuffer. However, DLSS-FG may draw debugging information and utilities to the output frame. Providing the real frame allows these debugging aids to be drawn every frame, instead of just to the interpolated frames (which would cause these aids to appear to flicker). For performance reasons, you may with to only provide this resource when these debugging aids may be present, for example, in non-production builds.

The resource provided must be in the same format as the input backbuffer color. This format must have at least three channels containing floating-point data (FLOAT, UNORM, or SNORM). For example, this could be `DXGI_FORMAT_R8G8B8A8_UNORM` in D3D12, or `VK_FORMAT_R8G8B8A8_UNORM` using Vulkan.

When running multi-frame generation, the rendered frame will be copied to this resource on each call to **`NVSDK_NGX_EvaluateFeature`**. To avoid the performance overhead of these additional copies, this resource should only be provided to one of the Evaluate calls.

You may specify that the output be written to a subrect of a larger resource. The subrect extent is the same as the subrect defined for the backbuffer. The resolution of the underlying resource must be the same as the resolution of the backbuffer unless the flag **`NVSDK_NGX_DLSSG_EvalFlags_UpdateOnlyInsideExtents`** is set.

This resource's resolution, or the resolution of its subrect if used, must match the width and height used when creating the feature.

The provided resource is assumed to be in a state suitable for writing. For D3D12, the resource must be in state `D3D12_RESOURCE_STATE_UNORDERED_ACCESS` before calling **`NVSDK_NGX_EvaluateFeature`**.

NVSDK_NGX_DLSSG_Parameter_OutputDisableInterpolation

The output resource indicating whether or not interpolation was disabled for the current frame.

Type: Resource

Optional.

Buffer for the algorithm to communicate to the application whether or not interpolation was disabled for this frame. If this buffer is provided, the algorithm will write a non-zero value to the first byte if it determined that interpolation should be disabled for the frame, or zero otherwise.

Interpolation may be disabled because the algorithm has determined that it doesn't have sufficient information to generate a frame (for example, after the **`NVSDK_NGX_DLSSG_Parameter_Reset`** flag is set), or because it has determined that a quality frame could not be generated (for example, because a scene change occurred).

Whenever interpolation is disabled, the algorithm copies the input backbuffer to the output interpolated frame resource. In this case, the application may wish not to present either the interpolated or rendered frame (since they are the same).

The provided resource is assumed to be in a state suitable for writing. For D3D12, the resource must be in state `D3D12_RESOURCE_STATE_UNORDERED_ACCESS` before calling **`NVSDK_NGX_EvaluateFeature`**.

NVSDK_NGX_DLSSG_Parameter_Reset

If set, resets the internal state of the feature.

Type: Boolean as `unsigned int` (0 or 1)

When set (non-zero), the feature's internal state will be reset. Set this flag when the information in the current frame is unrelated to the previous frame, such as when a scene change occurs.

Resetting the feature effectively clears the algorithm's internal history, which includes any information about the previous frame. Since the algorithm no longer has a pair of frames to interpolate between, the interpolated output will be a copy of the input backbuffer until the next rendered frame is provided, at which point interpolation will resume.

Because interpolation between two unrelated frames generally yields poor results, the algorithm may detect scene changes and will reset the feature automatically if it determines that a scene change has likely occurred. However, applications should still set the reset flag during these scene changes as the algorithm may not always be able to determine whether or not a scene change has actually occurred.

The reset flag is assumed to be set on the first call to **`NVSDK_NGX_EvaluateFeature`** after the feature is created, regardless of the parameter's actual value.

NVSDK_NGX_DLSSG_Parameter_ClipToPrevClip

Transformation matrix from the current frame's clip space to the previous frame's clip space.

Type: Camera Matrix (`float[4][4]`) as `void*`

This matrix describes the transformation from the current frame's clip space to the previous frame's clip space.

NVSDK_NGX_DLSSG_Parameter_PrevClipToClip

Transformation matrix from the previous frame's clip space to the current frame's clip space.

Type: Camera Matrix (`float[4][4]`) as `void*`

This matrix describes the transformation from the previous frame's clip space to the current frame's clip space.

NVSDK_NGX_DLSSG_Parameter_MvecScaleX

Scale factor applied to motion vectors in the X direction.

Type: `float`

Optional. Default: 1.0

Factor by which the X component of the input motion vectors will be multiplied. Motion vectors are expected to be provided in screen-space pixels at the resolution of the motion vector resource. For motion vectors which use a different scale or direction, you can use this parameter to rescale and/or invert them.

Common values are ±1 (if no scale needs to be applied), or ± the width of the motion vector resource (if motion vectors are in normalized/UV space).

NVSDK_NGX_DLSSG_Parameter_MvecScaleY

Scale factor applied to motion vectors in the Y direction.

Type: `float`

Optional. Default: 1.0

Factor by which the X component of the input motion vectors will be multiplied. Motion vectors are expected to be provided in screen-space pixels at the resolution of the motion vector resource. For motion vectors which use a different scale or

direction, you can use this parameter to rescale and/or invert them.

Common values are ±1 (if no scale needs to be applied), or ± the height of the motion vector resource (if motion vectors are in normalized/UV space).

NVSDK_NGX_DLSSG_Parameter_CameraPosX

X-coordinate of the camera position.

Type: `float`

The X-coordinate of the camera's position in world space.

NVSDK_NGX_DLSSG_Parameter_CameraPosY

Y-coordinate of the camera position.

Type: `float`

The Y-coordinate of the camera's position in world space.

NVSDK_NGX_DLSSG_Parameter_CameraPosZ

Z-coordinate of the camera position.

Type: `float`

The Z-coordinate of the camera's position in world space.

NVSDK_NGX_DLSSG_Parameter_CameraUpX

X-component of the camera's up vector.

Type: `float`

The X-component of the camera's up vector in world space.

NVSDK_NGX_DLSSG_Parameter_CameraUpY

Y-component of the camera's up vector.

Type: `float`

The Y-component of the camera's up vector in world space.

NVSDK_NGX_DLSSG_Parameter_CameraUpZ

Z-component of the camera's up vector.

Type: `float`

The Z-component of the camera's up vector in world space.

NVSDK_NGX_DLSSG_Parameter_CameraRightX

X-component of the camera's right vector.

Type: `float`

The X-component of the camera's right vector in world space.

NVSDK_NGX_DLSSG_Parameter_CameraRightY

Y-component of the camera's right vector.

Type: `float`

The Y-component of the camera's right vector in world space.

NVSDK_NGX_DLSSG_Parameter_CameraRightZ

Z-component of the camera's right vector.

Type: `float`

The Z-component of the camera's right vector in world space.

NVSDK_NGX_DLSSG_Parameter_CameraFwdX

X-component of the camera's forward vector.

Type: `float`

The X-component of the camera's forward vector in world space.

NVSDK_NGX_DLSSG_Parameter_CameraFwdY

Y-component of the camera's forward vector.

Type: `float`

The Y-component of the camera's forward vector in world space.

NVSDK_NGX_DLSSG_Parameter_CameraFwdZ

Z-component of the camera's forward vector.

Type: `float`

The Z-component of the camera's forward vector in world space.

NVSDK_NGX_DLSSG_Parameter_CameraNear

Near clipping plane distance.

Type: `float`

The distance to the camera's near clipping plane.

NVSDK_NGX_DLSSG_Parameter_CameraFar

Far clipping plane distance.

Type: `float`

The distance to the far clipping plane.

NVSDK_NGX_DLSSG_Parameter_DepthInverted

Whether or not the depth buffer is inverted.

Type: Boolean as `unsigned int` (0 or 1)

Optional. Default: false (0)

When set (non-zero), indicates that the input depth buffer values are inverted (1.0 is near, 0.0 is far).

NVSDK_NGX_DLSSG_Parameter_LinearizedDepth_Scale

Optional scalar on the linearized depth before algorithm processing.

Type: `float`

Optional. Default: 1.0f.

DLSS-FG performs some internal processing steps in linear depth units (`1 / depth` if inverted, or `1 / (1 - depth)` otherwise). This parameter applies an additional scale to the linearized depth values before processing. While most integrations don't need to modify this parameter, you might find it useful to provide a smaller value (e.g. 0.1f) if your input depth units are compressed into a small dynamic range.

NVSDK_NGX_DLSSG_Parameter_LinearizedDepth_NearFarPartition

Optional heuristic that defines the Z-distance to the depth plane that artificially partitions near from far objects.

Type: `float`

Optional. Default: 600.0f.

This parameter defines a z-distance to a depth plane which artificially partitions foreground objects from background objects. This distance is measured in linear depth units (`1 / depth` if inverted, or `1 / (1 - depth)` otherwise). DLSS-FG uses this plane to restrict background motion from covering foreground motion. The default value works well for most integrations, but you might adjust this to a larger value (e.g. 4000.0f) if your input depth units are compressed into a small dynamic range.

NVSDK_NGX_DLSSG_Parameter_MinRelativeLinearDepthObjectSeparation

Minimum relative linear depth object separation.

Type: `float`

Optional. Default: 40.0f

Heuristic which helps the algorithm determine whether or not two pixels are part of contiguous surfaces, measured in linear depth units (`1 / depth` if inverted, or `1 / (1 - depth)` otherwise).

A default value of 40.0f is recommended for most cases. A smaller value may be needed if the range of values in the depth buffer is compressed unusually close to 1.0.

NVSDK_NGX_DLSSG_Parameter_Bidirectional_Distortion_Field_LowPrecision_IsLowPrecision

Indicates if the bidirectional distortion field is in low precision.

Type: Boolean as `int` (0 or 1)

When set to a non-zero value, indicates that the bidirectional distortion field is in low precision. The default value is 0.

NVSDK_NGX_DLSSG_Parameter_Bidirectional_Distortion_Field_LowPrecision_Bias

Bias for low precision bidirectional distortion field.

Type: `float`

The bias value for the low precision bidirectional distortion field.

NVSDK_NGX_DLSSG_Parameter_Bidirectional_Distortion_Field_LowPrecision_Scale

Scale for low precision bidirectional distortion field.

Type: `float`

The scale value for the low precision bidirectional distortion field. The default value is 0.0f.

NVSDK_NGX_DLSSG_Parameter_EvalFlags

Bitfield containing flags that control various properties of the algorithm.

Type: `unsigned int`

Bit-flags that control various aspects of the frame generation algorithm. The following flags are supported:

- `NVSDK_NGX_DLSSG_EvalFlags_UpdateOnlyInsideExtents`: When using a subrect for the backbuffer, the algorithm's default behavior is to copy the input pixels to the output resource outside of the subrect. If this flag is set, the algorithm will only write within the output subrect, leaving pixels outside the boundary unchanged.

NVSDK_NGX_DLSSG_Parameter_MultiFrameIndex

Index (starting at 1) of the frame to generate for multi-frame generation.

Type: `unsigned int`

Optional for single-frame generation.

When using multi-frame generation, this parameter specifies which intermediate frame to generate between the previous and current rendered frames. The value must be between 1 and **NVSDK_NGX_DLSSG_Parameter_MultiFrameCount** (inclusive).

For example, with 4x multi-frame generation (MultiFrameCount = 3):

- Index 1 generates a frame 25% between the frames
- Index 2 generates a frame 50% between the frames
- Index 3 generates a frame 75% between the frames

NVSDK_NGX_EvaluateFeature must be called a number of times equal to the value of **NVSDK_NGX_DLSSG_Parameter_MultiFrameCount**. The inputs parameters must be the same for each call, except for the MultiFrameIndex, which must increment consecutively. when generating three intermediate frames (4x multi-frame generation) calls must be made with MultiFrameIndex set to 1, 2, and then 3 in that order. It is undefined behavior for any index to be skipped or called out of order.

When using the **NVSDK_NGX_DLSSG_Parameter_Reset** parameter to reset the algorithm, the MultiFrameIndex must be 1 for that Evaluate call; the reset parameter is ignored for other values. This may break the sequence from the previous frame (for example, it is valid to call Evaluate with Index = 1 and 2 and then reset back to 1, skipping 3). Subsequent calls to Evaluate will again require consecutively increasing values of MultiFrameIndex.

In single-frame mode, this value does not need to be provided. If it is, its value must be 1.

NVSDK_NGX_DLSSG_Parameter_MultiFrameCount

The number of intermediate frames to generate between each pair of rendered input frames.

Type: `unsigned int`

Optional. Default: 1 (2x frame generation)

The number of frames to generate between each pair of rendered frames. For example, a value of 3 will produce three generated frames for each rendered frame, effectively achieving a 4x frame rate increase.

The MultiFrameCount must be in the (inclusive) range between 1 and the value of **NVSDK_NGX_DLSSG_Parameter_MultiFrameCountMax** as reported by `NVSDK_NGX_GetCapabilityParameters`.

The application must call **NVSDK_NGX_EvaluateFeature** a number of times equal to MultiFrameCount for each input frame pair, with incrementing **NVSDK_NGX_DLSSG_Parameter_MultiFrameIndex** parameter values from 1 to MultiFrameCount (inclusive).

The value of MultiFrameCount may be changed between input frame pairs. However, once it has been set for a frame pair, the application must complete the sequence of Evaluate calls (with MultiFrameIndex incrementing from 1 to MultiFrameCount) before changing the MultiFrameCount for the subsequent frame pair.

If this parameter is not provided, it defaults to 1 (2x/single-frame generation).

NVSDK_NGX_DLSSG_Parameter_UserDebugText

User-provided debug text which will be displayed on screen in non-production builds.

Type: `const char*` as `void*`

Optional. Only available in non-production builds.

This parameter allows providing debug text that will be displayed on screen in non-production builds. The text can be used for debugging and profiling purposes.

NVSDK_NGX_DLSSG_Parameter_CameraViewToClip

Transformation matrix from camera view space to clip space.

Type: Camera Matrix (`float[4][4]`) as `void*`

This matrix describes the transformation from camera view space to clip space. It is used to improve the quality of frame interpolation.

NVSDK_NGX_DLSSG_Parameter_ClipToCameraView

Transformation matrix from clip space to view space.

Type: Camera Matrix (`float[4][4]`) as `void*`

This matrix describes the transformation from clip space to camera view space. It is used to improve the quality of frame interpolation.

NVSDK_NGX_DLSSG_Parameter_ClipToLensClip

Transformation matrix describing lens distortion in clip space.

Type: Camera Matrix (`float[4][4]`) as `void*`

This matrix describes lens distortion in clip space. It is used to improve the quality of frame interpolation when lens distortion is present.

NVSDK_NGX_DLSSG_Parameter_JitterOffsetX

Jitter offset in the X direction.

Type: `float`

The jitter offset in the X direction.

NVSDK_NGX_DLSSG_Parameter_JitterOffsetY

Jitter offset in the Y direction.

Type: `float`

The jitter offset in the Y direction.

NVSDK_NGX_DLSSG_Parameter_CameraPinholeOffsetX

Pinhole offset in the X direction for the camera.

Type: `float`

The pinhole offset in the X direction for the camera.

NVSDK_NGX_DLSSG_Parameter_CameraPinholeOffsetY

Pinhole offset in the Y direction for the camera.

Type: `float`

The pinhole offset in the Y direction for the camera.

NVSDK_NGX_DLSSG_Parameter_CameraFOV

Field of view of the camera, in degrees or radians.

Type: `float`

The field of view of the camera.

NVSDK_NGX_DLSSG_Parameter_CameraAspectRatio

Aspect ratio of the camera.

Type: `float`

The aspect ratio of the camera.

NVSDK_NGX_DLSSG_Parameter_ColorBuffersHDR

Indicates if the color buffers use HDR.

Type: Boolean as `unsigned int` (0 or 1)

When set to a non-zero value, indicates that the color buffers use HDR.

NVSDK_NGX_DLSSG_Parameter_CameraMotionIncluded

Indicates if camera motion is included in the input motion vector field.

Type: Boolean as `unsigned int` (0 or 1)

When set to a non-zero value, indicates that camera motion is included in the input motion vector field.

NVSDK_NGX_DLSSG_Parameter_NotRenderingGameFrames

Indicates if the frames currently being rendered by the game are not "game frames".

Type: Boolean as `unsigned int` (0 or 1)

When set to a non-zero value, indicates that the frames currently being rendered by the game are not "game frames".

NVSDK_NGX_DLSSG_Parameter_OrthoProjection

Indicates if orthographic projection is used by the camera.

Type: Boolean as `unsigned int` (0 or 1)

When set to a non-zero value, indicates that orthographic projection is used by the camera.

NVSDK_NGX_DLSSG_Parameter_MvecInvalidValue

Invalid value for motion vectors.

Type: `float`

The value used to indicate invalid motion vectors.

NVSDK_NGX_DLSSG_Parameter_MvecDilated

Indicates if motion vectors are dilated.

Type: Boolean as `unsigned int` (0 or 1)

When set to a non-zero value, indicates that motion vectors are dilated.

NVSDK_NGX_DLSSG_Parameter_MenuDetectionEnabled

Enables full-screen menu detection.

Type: Boolean as `unsigned int` (0 or 1)

When set to a non-zero value, enables full-screen menu detection.

NVSDK_NGX_DLSSG_Parameter_BackbufferSubrectBaseX

X-coordinate of the base of the backbuffer subrect.

Type: int

Optional. Default: 0

Defines the X-coordinate of the top-left corner of the backbuffer subrect, in pixels, relative to the top-left origin of the full backbuffer resource.

This parameter defines the subrect region for all backbuffer-related resources, which includes the input backbuffer color (**`NVSDK_NGX_DLSSG_Parameter_Backbuffer`**), the generated output color (**`NVSDK_NGX_Parameter_OutputInterpolated`**), and the real frame output (**`NVSDK_NGX_Parameter_OutputReal`**).

NVSDK_NGX_DLSSG_Parameter_BackbufferSubrectBaseY

Y-coordinate of the base of the backbuffer subrect.

Type: int

Optional. Default: 0

Defines the Y-coordinate of the top-left corner of the backbuffer subrect, in pixels, relative to the top-left origin of the full backbuffer resource.

This parameter defines the subrect region for all backbuffer-related resources, which includes the input backbuffer color (**NVSDK_NGX_DLSSG_Parameter_Backbuffer**), the generated output color (**NVSDK_NGX_Parameter_OutputInterpolated**), and the real frame output (**NVSDK_NGX_Parameter_OutputReal**).

NVSDK_NGX_DLSSG_Parameter_BackbufferSubrectWidth

Width of the backbuffer subrect.

Type: int

Optional. Default: 0

Specifies the width of the backbuffer subrect in pixels. If this and all other backbuffer subrect parameters are left at their default value of 0, DLSS-FG will assume the subrect's width is the same as the width of the full backbuffer resource.

This parameter defines the subrect region for all backbuffer-related resources, which includes the input backbuffer color (**NVSDK_NGX_DLSSG_Parameter_Backbuffer**), the generated output color (**NVSDK_NGX_Parameter_OutputInterpolated**), and the real frame output (**NVSDK_NGX_Parameter_OutputReal**).

NVSDK_NGX_DLSSG_Parameter_BackbufferSubrectHeight

Height of the backbuffer subrect.

Type: int

Optional. Default: 0

Specifies the height of the backbuffer subrect in pixels. If this and all other backbuffer subrect parameters are left at their default value of 0, DLSS-FG will assume the subrect's height is the same as the height of the full backbuffer resource.

This parameter defines the subrect region for all backbuffer-related resources, which includes the input backbuffer color (**NVSDK_NGX_DLSSG_Parameter_Backbuffer**), the generated output color (**NVSDK_NGX_Parameter_OutputInterpolated**), and the real frame output (**NVSDK_NGX_Parameter_OutputReal**).

NVSDK_NGX_DLSSG_Parameter_MVecsSubrectBaseX

X-coordinate of the base of the motion vectors subrect.

Type: int

Optional. Default: 0

Defines the X-coordinate of the top-left corner of the subrect for the input motion vectors resource, in pixels, relative to the top-left origin of the full motion vectors resource.

NVSDK_NGX_DLSSG_Parameter_MVecsSubrectBaseY

Y-coordinate of the base of the motion vectors subrect.

Type: int

Optional. Default: 0

Defines the Y-coordinate of the top-left corner of the subrect for the input motion vectors resource, in pixels, relative to the top-left origin of the full motion vectors resource.

NVSDK_NGX_DLSSG_Parameter_MVecsSubrectWidth

Width of the motion vectors subrect.

Type: int

Optional. Default: 0

Specifies the width of the subrect for the input motion vectors resource in pixels. If this and all other motion vectors subrect parameters are left at their default value of 0, DLSS-FG will assume the subrect's width is the same as the width of the full motion vectors resource.

NVSDK_NGX_DLSSG_Parameter_MVecsSubrectHeight

Height of the motion vectors subrect.

Type: int

Optional. Default: 0

Specifies the height of the subrect for the input motion vectors resource in pixels. If this and all other motion vectors subrect parameters are left at their default value of 0, DLSS-FG will assume the subrect's height is the same as the height of the full motion vectors resource.

NVSDK_NGX_DLSSG_Parameter_DepthSubrectBaseX

X-coordinate of the base of the depth subrect.

Type: int

Optional. Default: 0

Defines the X-coordinate of the top-left corner of the input depth subrect, in pixels, relative to the top-left origin of the full depth resource.

NVSDK_NGX_DLSSG_Parameter_DepthSubrectBaseY

Y-coordinate of the base of the depth subrect.

Type: int

Optional. Default: 0

Defines the Y-coordinate of the top-left corner of the input depth subrect, in pixels, relative to the top-left origin of the full depth resource.

NVSDK_NGX_DLSSG_Parameter_DepthSubrectWidth

Width of the depth subrect.

Type: int

Optional. Default: 0

Specifies the width of the input depth subrect in pixels. If this and all other depth subrect parameters are left at their default value of 0, DLSS-FG will assume the subrect's width is the same as the width of the full depth resource.

NVSDK_NGX_DLSSG_Parameter_DepthSubrectHeight

Height of the depth subrect.

Type: int

Optional. Default: 0

Specifies the height of the input depth subrect in pixels. If this and all other depth subrect parameters are left at their default value of 0, DLSS-FG will assume the subrect's height is the same as the height of the full depth resource.

NVSDK_NGX_DLSSG_Parameter_HUDLessSubrectBaseX

X-coordinate of the base of the HUDLess subrect.

Type: int

Optional. Default: 0

Defines the X-coordinate of the top-left corner of the subrect for the input HUDLess resource, in pixels, relative to the top-left origin of the full HUDLess resource.

NVSDK_NGX_DLSSG_Parameter_HUDLessSubrectBaseY

Y-coordinate of the base of the HUDLess subrect.

Type: int

Optional. Default: 0

Defines the Y-coordinate of the top-left corner of the subrect for the input HUDLess resource, in pixels, relative to the top-left origin of the full HUDLess resource.

NVSDK_NGX_DLSSG_Parameter_HUDLessSubrectWidth

Width of the HUDLess subrect.

Type: int

Optional. Default: 0

Specifies the width of the HUDLess subrect in pixels. If this and all other HUDLess subrect parameters are left at their default value of 0, DLSS-FG will assume the subrect's width is the same as the width of the full HUDLess resource.

NVSDK_NGX_DLSSG_Parameter_HUDLessSubrectHeight

Height of the HUDLess subrect.

Type: int

Optional. Default: 0

Specifies the height of the HUDLess subrect in pixels. If this and all other HUDLess subrect parameters are left at their default value of 0, DLSS-FG will assume the subrect's height is the same as the height of the full HUDLess resource.

NVSDK_NGX_DLSSG_Parameter_UISubrectBaseX

X-coordinate of the base of the UI subrect.

Type: int

Optional. Default: 0

Defines the X-coordinate of the top-left corner of the subrect for the input UI resource, in pixels, relative to the top-left origin of the full UI resource.

NVSDK_NGX_DLSSG_Parameter_UISubrectBaseY

Y-coordinate of the base of the UI subrect.

Type: int

Optional. Default: 0

Defines the Y-coordinate of the top-left corner of the subrect for the input UI resource, in pixels, relative to the top-left origin of the full UI resource.

NVSDK_NGX_DLSSG_Parameter_UISubrectWidth

Width of the UI subrect.

Type: int

Optional. Default: 0

Specifies the width of the UI resource subrect in pixels. If this and all other UI subrect parameters are left at their default value of 0, DLSS-FG will assume the subrect's width is the same as the width of the full UI resource.

NVSDK_NGX_DLSSG_Parameter_UISubrectHeight

Height of the UI subrect.

Type: int

Optional. Default: 0

Specifies the height of the UI resource subrect in pixels. If this and all other UI subrect parameters are left at their default value of 0, DLSS-FG will assume the subrect's height is the same as the height of the full UI resource.

NVSDK_NGX_DLSSG_Parameter_BidirectionalDistortionFieldSubrectBaseX

X-coordinate of the base of the bidirectional distortion field subrect.

Type: int

Optional. Default: 0

Defines the X-coordinate of the top-left corner of the input distortion field subrect, in pixels, relative to the top-left origin of the full distortion field resource.

NVSDK_NGX_DLSSG_Parameter_BidirectionalDistortionFieldSubrectBaseY

Y-coordinate of the base of the bidirectional distortion field subrect.

Type: int

Optional. Default: 0

Defines the Y-coordinate of the top-left corner of the input distortion field subrect, in pixels, relative to the top-left origin of the full distortion field resource.

NVSDK_NGX_DLSSG_Parameter_BidirectionalDistortionFieldSubrectWidth

Width of the bidirectional distortion field subrect.

Type: int

Optional. Default: 0

Specifies the width of the distortion field subrect in pixels. If this and all other distortion field subrect parameters are left at their default value of 0, DLSS-FG will assume the subrect's width is the same as the width of the full distortion field resource.

NVSDK_NGX_DLSSG_Parameter_BidirectionalDistortionFieldSubrectHeight

Height of the bidirectional distortion field subrect.

Type: int

Optional. Default: 0

Specifies the height of the distortion field subrect in pixels. If this and all other distortion field subrect parameters are left at their default value of 0, DLSS-FG will assume the subrect's height is the same as the height of the full distortion field resource.

NVSDK_NGX_Parameter_FrameGeneration_Available

Whether or not frame generation is available on the current system.

Type: Boolean as `unsigned int` (0 or 1)

If set (non-zero), indicates that frame generation is available on the current system. If this parameter is not present, assume a default value of false (0).

NVSDK_NGX_Parameter_FrameGeneration_FeatureInitResult

If frame generation could not be initialized successfully, the `NVSDK_NGX_Result` indicating the reason for failure.

Type: `NVSDK_NGX_Result` as `int`

This parameter contains the result code from the last attempt to initialize frame generation. If initialization was successful, this value will be **`NVSDK_NGX_Result_Success`**.

NVSDK_NGX_Parameter_FrameGeneration_MinDriverVersionMajor

Major version number of the minimum display driver required for frame generation.

Type: `unsigned int`

This parameter specifies the major version number of the minimum display driver required for frame generation.

NVSDK_NGX_Parameter_FrameGeneration_MinDriverVersionMinor

Minor version number of the minimum display driver required for frame generation.

Type: `unsigned int`

This parameter specifies the minor version number of the minimum display driver required for frame generation.

NVSDK_NGX_Parameter_FrameGeneration_NeedsUpdatedDriver

If frame generation is not available, whether or not an updated driver is needed for frame generation.

Type: Boolean as `unsigned int` (0 or 1)

When set, indicates that frame generation is not available because it requires a higher driver version than the version currently installed on the system.

NVSDK_NGX_DLSSG_Parameter_MultiFrameCountMax

Maximum number of interpolated frames supported for multi-frame generation.

Type: `unsigned int`

The maximum number of interpolated frames that this version of the algorithm can generate between each pair of game-rendered frames on the current system. When calling **`NVSDK_NGX_EvaluateFeature`**, the application must not set **`NVSDK_NGX_DLSSG_Parameter_MultiFrameCount`** to a higher value.

The application should query this parameter before enabling multi-frame generation. This value should also be used to populate the application's settings with the available multipliers.

If frame generation is supported on the current system (as reported by the parameter **`NVSDK_NGX_Parameter_FrameGeneration_Available`**), single-frame generation must also be supported, at a minimum. If this parameter is not set when using `NVSDK_NGX_GetCapabilityParameters`, or it has a value less than 1, assume a default value of 1.

NVSDK_NGX_Parameter_DLSSGGetCurrentSettingsCallback

Callback for getting current DLSSG settings.

Type: `NVSDK_NGX_DLSSGGetCurrentSettingsCallback` as `void*`

This callback function is called to get the current DLSSG settings. The callback should populate the provided parameter map with the current settings.

NVSDK_NGX_Parameter_DLSSGEstimateVRAMCallback

Callback for estimating VRAM usage.

Type: `NVSDK_NGX_DLSSGEstimateVRAMCallback` as `void*`

This callback function is called to estimate the VRAM usage for DLSSG. The callback should calculate the estimated VRAM usage based on the provided parameters and return the result in bytes.

DLSS-FG Helper Functions

DLSS Frame Generation provides helper function wrappers around NGX SDK functionality. These helpers are defined in:

- `nvsdk_ngx_helpers_dlssg.h` for D3D12 helpers.
- `nvsdk_ngx_helpers_dlssg_vk.h` for Vulkan helpers.

The following helper functions are available for both D3D12 and Vulkan:

Helper Function	Description
NGX_CREATE_DLSSG	Creates a new DLSS-FG instance.
NGX_EVALUATE_DLSSG	Evaluates a DLSS-FG instance.
NGX_ESTIMATE_VRAM_DLSSG	Retrieves an estimate for the amount of memory DLSS-FG will allocate when created at a particular resolution.

These helper functions reference parameter structs. These are defined in:

- `nvsdk_ngx_params_dlssg.h` for API-agnostic parameters
- `nvsdk_ngx_helpers_dlssg.h` or `nvsdk_ngx_helpers_dlssg_vk.h` for API-specific parameters.

Struct Name	Description
NVSDK_NGX_DLSSG_Create_Params	Parameters passed to DLSS-FG during feature creation.
NVSDK_NGX_DLSSG_Eval_Params	API-specific resource parameters to DLSS-FG evaluate.
NVSDK_NGX_DLSSG_Opt_Eval_Params	API-agnostic parameters to DLSS-FG evaluate.

Helper Functions

NGX_CREATE_DLSSG

Creates a new DLSS-FG instance.

```
static inline NVSDK_NGX_Result NGX_D3D12_CREATE_DLSSG(
    ID3D12GraphicsCommandList* pInCmdList, unsigned int InCreationNodeMask,
    unsigned int InVisibilityNodeMask, NVSDK_NGX_Handle** ppOutHandle,
    NVSDK_NGX_Parameter* pInParams, NVSDK_NGX_DLSSG_Create_Params* pInDlssgCreateParams);
static inline NVSDK_NGX_Result NGX_VK_CREATE_DLSSG(
    VkCommandBuffer pInCmdBuf, unsigned int InCreationNodeMask, unsigned int InVisibilityNodeMask,
    NVSDK_NGX_Handle** ppOutHandle, NVSDK_NGX_Parameter* pInParams,
    NVSDK_NGX_DLSSG_Create_Params* pInDlssgCreateParams);
```

This helper is a wrapper around **NVSDK_NGX_CreateFeature**. It creates a DLSS-FG instance with the parameters set in `pInDlssgCreateParams`.

Parameters

- `pInCmdList` / `pInCmdBuf`: The graphics API command list or command buffer where NGX will enqueue its resource creation commands.
- `InCreationNodeMask`: A bitmask identifying the GPU node on which you want to create the DLSS-FG feature. Use 0 for single-GPU setups.
- `InVisibilityNodeMask`: A bitmask identifying the GPU node(s) from which the created DLSS-FG feature will be visible. Use 0 for single-GPU setups.
- `ppOutHandle`: An output pointer that NGX will populate with the handle to the newly created DLSS-FG instance. You will use this handle for subsequent calls to **NVSDK_NGX_EvaluateFeature** and **NVSDK_NGX_ReleaseFeature**.
- `pInParams`: A pointer to an **NVSDK_NGX_Parameter** map. The helper function will populate this map with parameters from `pInDlssgCreateParams`. You can also set additional or advanced creation parameters in this map beyond what the struct covers.
- `pInDlssgCreateParams`: A pointer to an **NVSDK_NGX_DLSSG_Create_Params** struct containing the required parameters for creating the DLSS-FG instance.

NGX_EVALUATE_DLSSG

Evaluates a DLSS-FG instance.

```
static inline NVSDK_NGX_Result NGX_D3D12_EVALUATE_DLSSG(
    ID3D12GraphicsCommandList* pInCmdList, NVSDK_NGX_Handle* pInHandle,
    NVSDK_NGX_Parameter* pInParams, NVSDK_NGX_D3D12_DLSSG_Eval_Params* pInDlssgEvalParams,
    NVSDK_NGX_DLSSG_Opt_Eval_Params* pInDlssgOptEvalParams);
static inline NVSDK_NGX_Result NGX_VK_EVALUATE_DLSSG(
    VkCommandBuffer pInCmdBuf, NVSDK_NGX_Handle* pInHandle, NVSDK_NGX_Parameter* pInParams,
    NVSDK_NGX_VK_DLSSG_Eval_Params* pInDlssgEvalParams,
    NVSDK_NGX_DLSSG_Opt_Eval_Params* pInDlssgOptEvalParams);
```

This helper is a wrapper around **NVSDK_NGX_EvaluateFeature**. It evaluates a DLSS-FG instance with the parameters set in `pInDlssgEvalParams` and `pInDlssgOptEvalParams`.

Parameters

- `pInCmdList` / `pInCmdBuf`: The graphics API command list or command buffer where NGX will enqueue its rendering commands for DLSS-FG evaluation.
- `pInHandle`: The handle to the DLSS Frame Generation instance you want to evaluate, previously created by **NGX_CREATE_DLSSG**.
- `pInParams`: A pointer to an **NVSDK_NGX_Parameter** map. The helper function will populate this map with parameters from `pInDlssgEvalParams` and `pInDlssgOptEvalParams`. You can also set additional or advanced evaluation parameters in this map beyond what these structs cover.
- `pInDlssgEvalParams`: A pointer to an **NVSDK_NGX_DLSSG_Eval_Params** struct containing the required parameters for evaluating the DLSS Frame Generation instance.
- `pInDlssgOptEvalParams`: A pointer to an **NVSDK_NGX_DLSSG_Opt_Eval_Params** struct containing additional parameters for evaluating the DLSS Frame Generation instance.

NGX_ESTIMATE_VRAM_DLSSG

Retrieves an estimate for the amount of memory DLSS-FG will allocate when created at a particular resolution.

```
static inline NVSDK_NGX_Result NGX_D3D12_ESTIMATE_VRAM_DLSSG(
    NVSDK_NGX_Parameter* InParams, uint32_t mvecDepthWidth, uint32_t mvecDepthHeight,
    uint32_t colorWidth, uint32_t colorHeight, uint32_t colorBufferFormat,
    uint32_t mvecBufferFormat, uint32_t depthBufferFormat, uint32_t hudLessBufferFormat,
    uint32_t uiBufferFormat, size_t* estimatedVRAMInBytes);
static inline NVSDK_NGX_Result NGX_VK_ESTIMATE_VRAM_DLSSG(
    NVSDK_NGX_Parameter* InParams, uint32_t mvecDepthWidth, uint32_t mvecDepthHeight,
    uint32_t colorWidth, uint32_t colorHeight, uint32_t colorBufferFormat,
    uint32_t mvecBufferFormat, uint32_t depthBufferFormat, uint32_t hudLessBufferFormat,
    uint32_t uiBufferFormat, size_t* estimatedVRAMInBytes);
```

This helper estimates the amount of VRAM that DLSS-FG will allocate when created with the specified parameters. This can be used to check if there will be sufficient VRAM available *before* attempting to create a DLSS-FG instance.

Parameters

- `InParams`: NGX parameter object containing capability parameters (as returned by **`NVSDK_NGX_GetCapabilityParameters`**)
- `mvecDepthWidth/Height`: Width and height of the motion vector and depth buffers. This should match the `RenderWidth` and `RenderHeight` that will be used to create the feature.
- `colorWidth/Height`: Width and height of color buffers. This should match the `Width` and `Height` that will be used to create the feature.
- `colorBufferFormat`: Format of the color buffer in the native graphics API (e.g. `DXGI_FORMAT_R8G8B8A8_UNORM` for D3D12 or `VK_FORMAT_R8G8B8A8_UNORM` for Vulkan)
- `mvecBufferFormat`: Format of the motion vector buffer in the native graphics API
- `depthBufferFormat`: Format of the depth buffer in the native graphics API
- `hudLessBufferFormat`: Format of the HUDLess buffer in the native graphics API
- `uiBufferFormat`: Format of the UI buffer in the native graphics API
- `estimatedVRAMInBytes`: Output parameter that receives the estimated VRAM usage in bytes

Returns

`NVSDK_NGX_Result` indicating success or failure. Common values include:

- **`NVSDK_NGX_Result_Success`** if the function executed successfully.
- **`NVSDK_NGX_Result_FAIL_OutOfDate`** if this functionality is not available in the version of the feature currently in use, or the parameters provided were not populated with NGX capabilities.

Helper Parameters

NVSDK_NGX_DLSSG_Create_Params

Parameters passed to DLSS-FG during feature creation.

```
typedef struct NVSDK_NGX_DLSSG_Create_Params
{
    unsigned int Width;
    unsigned int Height;
    unsigned int NativeBackbufferFormat;
    unsigned int RenderWidth;
    unsigned int RenderHeight;
    bool DynamicResolutionScaling;
} NVSDK_NGX_DLSSG_Create_Params;
```

Fields

- `Width` / `Height`: The resolution of the input/output color resources.
- `NativeBackbufferFormat`: The format of the input backbuffer color resource, as defined by the graphics API in use (e.g., `DXGI_FORMAT_R10G10B10A2_UNORM`).
- `RenderWidth` / `RenderHeight`: The resolution at which the game was initially rendered. If using an upscaler such as DLSS Super Resolution, this is the input resolution to DLSS-SR. In most cases, the depth and motion vector resources should be this resolution.
- `DynamicResolutionScaling`: Boolean `true` if dynamic resolution scaling is enabled, `false` otherwise. If `true`, RenderWidth and RenderHeight are ignored.

NVSDK_NGX_DLSSG_Eval_Params

API-specific resource parameters to DLSS-FG evaluate.

```
typedef struct NVSDK_NGX_D3D12_DLSSG_Eval_Params
{
    ID3D12Resource* pBackbuffer;
    ID3D12Resource* pDepth;
    ID3D12Resource* pMVecs;
    ID3D12Resource* pHudless;
    ID3D12Resource* pUI;
    ID3D12Resource* pNoPostProcessingColor;
    ID3D12Resource* pBidirectionalDistortionField;
    ID3D12Resource* pOutputInterpFrame;
    ID3D12Resource* pOutputRealFrame;
    ID3D12Resource* pOutputDisableInterpolation;
} NVSDK_NGX_D3D12_DLSSG_Eval_Params;

typedef struct NVSDK_NGX_VK_DLSSG_Eval_Params
{
    NVSDK_NGX_Resource_VK* pBackbuffer;
    NVSDK_NGX_Resource_VK* pDepth;
    NVSDK_NGX_Resource_VK* pMVecs;
    NVSDK_NGX_Resource_VK* pHudless;
    NVSDK_NGX_Resource_VK* pUI;
    NVSDK_NGX_Resource_VK* pNoPostProcessingColor;
    NVSDK_NGX_Resource_VK* pBidirectionalDistortionField;
    NVSDK_NGX_Resource_VK* pOutputInterpFrame;
    NVSDK_NGX_Resource_VK* pOutputRealFrame;
    NVSDK_NGX_Resource_VK* pOutputDisableInterpolation;
} NVSDK_NGX_VK_DLSSG_Eval_Params;
```

Fields

- `pBackbuffer`: The input backbuffer color. See **`NVSDK_NGX_DLSSG_Parameter_Backbuffer`**.
- `pDepth`: The depth buffer for the current frame. See **`NVSDK_NGX_DLSSG_Parameter_Depth`**.
- `pMVecs`: The input motion vectors. See **`NVSDK_NGX_DLSSG_Parameter_MVecs`**.
- `pHudless`: The input hudless color buffer. See **`NVSDK_NGX_DLSSG_Parameter_HUDLess`**.
- `pUI`: The input UI elements buffer. See **`NVSDK_NGX_DLSSG_Parameter_UI`**.
- `pNoPostProcessingColor`: The input color buffer without post-processing effects. See **`NVSDK_NGX_DLSSG_Parameter_NoPostProcessingColor`**.
- `pBidirectionalDistortionField`: The input bidirectional distortion field. See **`NVSDK_NGX_DLSSG_Parameter_BidirectionalDistortionField`**.
- `pOutputInterpFrame`: The output resource to write the interpolated frame to. See **`NVSDK_NGX_Parameter_OutputInterpolated`**.
- `pOutputRealFrame`: The output resource to write the real frame to. See **`NVSDK_NGX_Parameter_OutputReal`**.
- `pOutputDisableInterpolation`: The output resource indicating whether or not interpolation was disabled for the current frame. See **`NVSDK_NGX_DLSSG_Parameter_OutputDisableInterpolation`**.

NVSDK_NGX_DLSSG_Opt_Eval_Params

API-agnostic parameters to DLSS-FG evaluate.

```
typedef struct NVSDK_NGX_DLSSG_Opt_Eval_Params
{
    float cameraViewToClip[4][4];
    float clipToCameraView[4][4];
    float clipToLensClip[4][4];
    float clipToPrevClip[4][4];
    float prevClipToClip[4][4];
    float jitterOffset[2];
    float mvecScale[2];
    float cameraPinholeOffset[2];
    float cameraPos[3];
    float cameraUp[3];
    float cameraRight[3];
    float cameraFwd[3];
    float cameraNear;
    float cameraFar;
    float cameraFOV;
    float cameraAspectRatio;
    bool colorBuffersHDR;
    bool depthInverted;
    bool cameraMotionIncluded;
    bool reset;
    bool automodeOverrideReset;
    bool notRenderingGameFrames;
    bool orthoProjection;
    float motionVectorsInvalidValue;
    bool motionVectorsDilated;
    bool menuDetectionEnabled;
    NVSDK_NGX_Coordinates mvecsSubrectBase = { 0, 0 };
    NVSDK_NGX_Dimensions mvecsSubrectSize = { 0, 0 };
    NVSDK_NGX_Coordinates depthSubrectBase = { 0, 0 };
    NVSDK_NGX_Dimensions depthSubrectSize = { 0, 0 };
    NVSDK_NGX_Coordinates hudLessSubrectBase = { 0, 0 };
    NVSDK_NGX_Dimensions hudLessSubrectSize = { 0, 0 };
    NVSDK_NGX_Coordinates uiSubrectBase = { 0, 0 };
    NVSDK_NGX_Dimensions uiSubrectSize = { 0, 0 };
    NVSDK_NGX_Coordinates bidirectionalDistFieldSubrectBase = { 0, 0 };
    NVSDK_NGX_Dimensions bidirectionalDistFieldSubrectSize = { 0, 0 };
    NVSDK_NGX_PrecisionInfo bidirectionalDistFieldPrecisionInfo = { 0, 0.0f, 0.0f };
    float minRelativeLinearDepthObjectSeparation = 40.0f;
    NVSDK_NGX_Coordinates backbufferSubrectBase = { 0, 0 };
    NVSDK_NGX_Dimensions  backbufferSubrectSize = { 0, 0 };
} NVSDK_NGX_DLSSG_Opt_Eval_Params;
```

Fields

- `cameraViewToClip`: Matrix transformation from the camera view to the clip space. Should NOT contain temporal AA jitter offset.
- `clipToCameraView`: Matrix transformation from the clip space to the camera view space.

- `clipToLensClip`: Matrix transformation describing lens distortion in clip space.
- `clipToPrevClip`: Matrix transformation from the current clip to the previous clip space.
- `prevClipToClip`: Matrix transformation from the previous clip to the current clip space.
- `jitterOffset`: Clip space jitter offset.
- `mvecScale`: Scale factors used to normalize motion vectors (so the values are in [-1,1] range).
- `cameraPinholeOffset`: Camera pinhole offset.
- `cameraPos`: Camera position in world space.
- `cameraUp`: Camera up vector in world space.
- `cameraRight`: Camera right vector in world space.
- `cameraFwd`: Camera forward vector in world space.
- `cameraNear`: Camera near view plane distance.
- `cameraFar`: Camera far view plane distance.
- `cameraFOV`: Camera field of view in radians.
- `cameraAspectRatio`: Camera aspect ratio defined as view space width divided by height.
- `colorBuffersHDR`: Specifies if tagged color buffers are full HDR (rendering to an HDR monitor) or not.
- `depthInverted`: Specifies if depth values are inverted (value closer to the camera is higher) or not.
- `cameraMotionIncluded`: Specifies if camera motion is included in the MVec buffer.
- `reset`: Specifies if previous frame has no connection to the current one (motion vectors are invalid).
- `automodeOverrideReset`: Specifies if Automode overrode the input Reset flag to true.
- `notRenderingGameFrames`: Specifies if application is not currently rendering game frames (paused in menu, playing video cut-scenes).
- `orthoProjection`: Specifies if orthographic projection is used or not.
- `motionVectorsInvalidValue`: Specifies which value represents an invalid (un-initialized) value in the motion vectors buffer.
- `motionVectorsDilated`: Specifies if motion vectors are already dilated or not.
- `menuDetectionEnabled`: Specifies whether or not fullscreen menu detection should be run.
- `mvecsSubrectBase`: Base coordinates for motion vectors subrect.
- `mvecsSubrectSize`: Size of motion vectors subrect.
- `depthSubrectBase`: Base coordinates for depth subrect.
- `depthSubrectSize`: Size of depth subrect.
- `hudLessSubrectBase`: Base coordinates for HUDLess subrect.
- `hudLessSubrectSize`: Size of HUDLess subrect.
- `uiSubrectBase`: Base coordinates for UI subrect.
- `uiSubrectSize`: Size of UI subrect.
- `bidirectionalDistFieldSubrectBase`: Base coordinates for bidirectional distortion field subrect.
- `bidirectionalDistFieldSubrectSize`: Size of bidirectional distortion field subrect.
- `bidirectionalDistFieldPrecisionInfo`: Precision information for bidirectional distortion field.
- `minRelativeLinearDepthObjectSeparation`: Optional heuristic that specifies the minimum depth difference between two objects in screen-space.
- `backbufferSubrectBase`: Base coordinates for backbuffer subrect.
- `backbufferSubrectSize`: Size of backbuffer subrect.

Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation ("NVIDIA") makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer ("Terms of Sale"). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer's own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer's product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

Trademarks

NVIDIA, the NVIDIA logo, GeForce and GeForce RTX are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright