NVIDIA DLSS (version 2.4)

Programming Guide

Document revision: 2.4.1

Released: 6 July 2022

Table of Contents

Contents

Table of Contents i

Abstract v

Revision History v

1 Introduction 1

2 Getting Started 1

2.1 System Requirements 1

2.2 Rendering Engine Requirements 2

2.3 DLSS Execution Times & GPU RAM Usage 2

2.4 DLSS Deployment Checklist 3

3 DLSS Integration 5

3.1 Pipeline Placement 5

3.1.1 DLSS During Early Phase Post Processing 5

3.1.2 Color Ranges for LDR and HDR 5

3.2 Integration Overview 6

3.2.1 DLSS Execution Modes 6

3.2.2 Dynamic Resolution Support 7

3.3 Supported Formats 9

3.4 Resource States 9

3.5 Mip-Map Bias 10

3.5.1 Mip-Map Bias Caveat: High-Frequency Textures 11

3.6 Motion Vectors 11

3.6.1 Motion Vector Format & Calculations 11

3.6.2 Motion Vector Flags 14

3.6.3 Motion Vector Scale 15

3.6.4 Conservative raster on Motion Vector for thin features 15

3.7 Sub-Pixel Jitter 16

3.7.1 Jitter Sample Patterns 17

3.7.2 Rendering with Jitter Offsets 17

3.7.3 Required Jitter Information 18

3.7.4 Troubleshooting Jitter Issues 19
3.8 Depth Buffer 19
3.8.1 Depth Buffer Flags 20
3.9 Exposure Parameter 20
3.9.1 Diagnosing bad exposure with the visualizer 20
3.9.2 Pre-Exposure Factor 22
3.10 Auto-exposure 22
3.11 Additional Sharpening 23
3.12 Scene Transitions 23
3.13 VRAM Usage 23
3.14 Biasing the Current Frame 24
3.15 Multi-view and Virtual Reality Support 24
3.16 Current DLSS Settings 25
3.16.1 DLSS Information Lines & Debug Hotkeys 26
3.17 NGX Logging 27
3.18 Sample Code 28
3.19 Integrating DLSS using Streamline SDK 28
4 Distributing DLSS in a Game 29
4.1 DLSS Release Process 29
4.2 Distributable Libraries 29
4.2.1 Removing the DLSS Library 29
4.2.2 Signing the DLSS Library 30
4.2.3 Notice of inclusion of third-party code 30
5 DLSS Code Integration 30
5.1 Adding DLSS to a Project 30
5.1.1 Linking the NGX SDK on Windows 30
5.1.2 Linking the NGX SDK on Linux 31
5.2 Initializing NGX SDK Object 31
5.2.1 Project ID 35
5.2.2 Engine Type 35
5.2.3 Engine Version 35
5.2.4 Thread Safety 35
5.2.5 Contexts and Command Lists 35

5.2.6 Verifying Availability of NGX Features and Allocating Parameter Maps 35
5.2.7 Overriding Feature Denial 39
5.2.8 Obtaining the Optimal Settings for DLSS 40
5.3 Feature Creation 41
5.4 Feature Evaluation 42
5.4.1 Vulkan Resource Wrapper 44
5.5 Feature Disposal 45
5.6 Shutdown 46
5.7 Init and Shutdown More than Once 46
6 Resource Management 46
6.1 D3D11 Specific 46
6.2 D3D12 Specific 47
6.3 Vulkan Specific 47
6.4 Common 48
7 Multi GPU Support 48
7.1 Linked Mode 48
7.2 Unlinked mode 49
8 Troubleshooting 49
8.1 Common Issues Causing Visual Artifacts 49
8.2 DLSS Debug Overlay 50
8.3 DLSS Debug Accumulation Mode 51
8.4 Jitter Troubleshooting 51
8.4.1 Initial Jitter Debugging 52
8.4.2 In-depth Jitter Debugging 53
8.5 Linux 55
8.6 Known Tooling Issues 56
8.7 Error Codes 56
9 Appendix 58
9.1 Transitioning from DLSS 2.0.x to 2.1.x 58
9.2 Minor Revision Updates 58
9.3 Future DLSS Parameters 58
9.4 Notices 59
9.4.1 Trademarks 60

9.4.2 Copyright 60
9.5 3rd Party Software 60
9.5.1 CURL 60
9.5.2 8x13 BITMAP FONT 61
9.5.3 d3dx12.h 61
9.5.4 xml 62
9.5.5 npy 63
9.5.6 stb 63
9.5.7 DirectX-Graphics-Samples 64
9.6 Linux driver compatibility 64

Abstract

The DLSS Programming Guide provides details on how to integrate and distribute DLSS in a game or 3D application. The Guide also provides embedded sample code and links to a full sample implementation on GitHub.

For information on using DLSS in Unreal Engine or Unity, please see https://www.unrealengine.com/marketplace/en-US/product/nvidia-dlss and https://docs.unity3d.com/2021.2/Documentation/Manual/deep-learning-super-sampling.html respectively.

Revision History

Revision	Changes	Date
2.4.1	- Update of minimum driver requirement	7/6/22
2.4	- Minor bug fixes and performance improvements - Added Streamline SDK pointers - Added Linux capability section	3/21/22
2.3.2	- Removed reference to DLSS Sharpening - Removed reference to UltraQuality in Execution modes due to user confusion	03/08/22
2.3.1	- Added a debug visualization to help diagnose exposure scale issues.	09/30/2021
2.2.3	- Adjusted the Texture LOD Bias recommendation for preservation of sharpness and intricate detail. - Added a few details to the Advanced Jitter debugging section. - Added a section about using conservative raster for improving the appearance of thin features	09/02/2021
2.2.2	- Updated based on easier access for DLSS SDK - Updated for Linux Support	7/12/2021
2.2.1	- Update NVSDK_NGX_VULKAN_RequiredExtensions() usage - Clarified that Vulkan application needs to run on Vulkan 1.1 or later	5/26/2021
2.2.0	- Added section on DLSS Auto-exposure - Slightly clarified the expected exposure value	4/9/2021

2.1.10	- Removed reference to DLSS Application Process	3/3/2021
2.1.9	- Update Resource States for Vulkan.	2/12/2021
2.1.8	- Added override for denied feature.	1/29/2021
2.1.7	- Added new entry points for NGX SDK. - Added information about new entry parameters. - Added information for new return error code. - Added information about post processing shaders that require depth buffer	1/13/2021
2.1.6	- Added section 3.6.3 Motion Vector Scale	12/10/2020
2.1.5	- Slightly clarified the usage of DLSS with multiple views, to make it clearer that it is not only for use with VR, but can be used in any multi-view use case	12/2/2020
2.1.4	- Added information about the proper use of the SDK API version value passed into the SDK at SDK initialization time - Added information about the NGX app logging hook API	11/2/2020
2.1.3	- Added section on VRAM usage - Added section on the current frame biasing - Fixed link to sample code on GitHub - Clarified motion vector resolution & dilation requirements - Added section on the JitterOffset debug overlay - Updated DLSS execution times	9/18/2020
2.1.2	- Fixed section 5 to include new parameters - Added caveat to mip-mapping section for high frequency textures - Removed unused parameters from depth buffer section	7/2/2020
2.1.1	- Clarified the mipmap bias requirement - Added section on Dynamic Resolution - Added section on DLSS in VR applications - Added section on the Exposure & Pre-Exposure parameters - Added section on DLSS Sharpness - Removed DLSS v1 transition notes from the Appendix	6/26/2020
2.0.2	- Added section describing the Depth Buffer parameter	4/10/2020

2.0.1	- Clarified SwapJitter debugging hotkey - Added section listing the JitterConfig debugging configurations	3/26/2020
2.0.0	- General edits throughout the document to update for DLSS v2 - Added renderer requirements - Added DLSS execution times - Added deployment checklist - Added section on jitter troubleshooting - Added section on resource states - Added section on DLSS debug logging	3/23/2020
1.3.9	- Added Vulkan Resource Metadata Wrapper - Fix bugs with debug overlay	11/08/2019
1.3.8	- Better explain motion vectors and added diagrams - Added detail on expected jitter and new debug mode - Further clarified LDR/HDR processing modes - Removed two resources and the associated data structures recently added to the SDK: camera vectors and transformation matrices - Fixed SDK enum and struct nomenclature to match the policy. - Added feature path list parameter to SDK API entry points in which feature DLL can be searched.	11/05/2019
1.3.7	- Added sample code to check for minimum driver version	10/15/2019
1.3.6	- Added information regarding on-screen debug info (3.7) - Added definition of LDR/SDR and HDR (3.1.3) - Added section describing future DLSS resources that are being considered by NVIDIA researchers - Clarified wording in the Code Integration sections	10/10/2019
1.3.5	- Updated section 3.5 regarding jitter pattern recommendation	9/13/2019
1.3.4	- MipMap LOD bias section - Clarified required/supported formats for various buffers - Updated initialization and feature creation sections (5.x) for new parameters, flags and options.	9/11/2019
1.3.3	- Updated Pipeline Placement section to include both placement options (pre post processing, and after post).	9/06/2019

	- Added troubleshooting section on Specular Aliasing	
1.3.2	- Updated Section 3.1 Pipeline Placement	9/06/2019
1.3.1	- Updates to various sections for ease of integration - Added Transitioning from DLSS 1.2.x to DLSS 1.3.x section 9.1 - Added New parameters for DLSS 1.3.x section 9.2	8/28/2019
1.2.0.0	- Added notice about approval requirements - Added section on jitter - Clarified motion vector requirements - Added dedicated section for buffer formats	8/15/2019
1.1.0.0	- Added sections for motion vectors, format support and pipeline integration. - Alignment of doc version number with DLSS release version - Support for debug overlays - Deprecated scratch buffer setup - Added links to sample code	7/08/2019
1.0.0.7	- Support for Vulkan titles - VS 2012 & VS 2013 static lib inclusion	5/17/2019
1.0.0.6	- Inclusion of DLSS Sample Code (section 5 & 6) - Inclusion of RTX Developer Guidelines in SDK docs - General document cleanup	4/10/2019
1.0.0.5	- Initial release	March 2019

1 Introduction

The NVIDIA DLSS technology provides smart: feature enhancement, anti-aliasing and upscaling, in a highly performant library. The library is tuned to take advantage of the latest features of NVIDIA RTX GPUs. Using DLSS, developers can dedicate more frame time to high-end rendering techniques and effects to enhance the visual experience while still maintaining high framerates.

DLSS is built and distributed as a feature of NVIDIA NGX which itself is one of the core components of NVIDIA RTX (https://developer.nvidia.com/rtx). NVIDIA NGX makes it easy to integrate pre-built AI based features into games and applications. As an NGX feature, DLSS takes advantage of the NGX update facility. When NVIDIA improves DLSS, the NGX infrastructure can be used to update DLSS for a specific title on all clients which currently have the game installed.

There are three main components that make up the underlying NGX system:

1. **NGX SDK**: The SDK provides CUDA, Vulkan, DirectX 11 and DirectX 12 API's for applications to use the NVIDIA supplied AI features. This document covers how to integrate the DLSS feature into a game or application using the DLSS SDK (which is modelled on the NGX SDK but stands separately).
2. **NGX Core Runtime**: The NGX Core Runtime is a system component which determines which shared library to load for each feature and application (or game). The NGX runtime module is always installed to the end-user's system as part of the NVIDIA Graphics Driver if supported NVIDIA RTX hardware is detected.
3. **NGX Update Module**: Updates to NGX features (including DLSS) are managed by the NGX Core Runtime itself. When a game or application instantiates an NGX feature, the runtime calls the NGX Update Module to check for new versions of the feature that is in use. If a newer version is found, the NGX Update Module downloads and swaps the DLL for the calling application.

2 Getting Started

2.1 System Requirements

The following is needed to load and run DLSS for both Windows and Linux:

- NVIDIA RTX GPU (GeForce, Titan or Quadro)
- The latest NVIDIA Graphics Driver is recommended. At a minimum for running DLSS 2.4.11+, you must have NVIDIA driver issued after March 3, 2022 (for instance 512.15). Please see Section 9.6 for Linux Driver Compatibility information.

The following is needed to load and run DLSS on Windows:

- Windows PC with Windows 10 v1709 (Fall 2017 Creators Update 64-bit) or newer

- The development environment for integrating the DLSS SDK into a game is:
    - Microsoft Visual Studio 2015 or newer.
    - Microsoft Visual Studio 2012 and 2013 are supported but may be deprecated in the future.

The following is needed to load and run DLSS on Linux:

- Linux kernel, 2.6.32 and newer
- DLSS SDK: glibc 2.11 or newer
- DLSS SDK Sample Code: gcc and g++ 8.4.0 or newer

2.2 Rendering Engine Requirements

The DLSS algorithm builds a high-resolution output buffer from information gathered over a series of frames. This document details what is needed to properly integrate DLSS and should be read in its entirety. As a summary, for DLSS to function with a high image quality, the rendering engine **must**:

- DirectX11, DirectX 12, or Vulkan based
    - **Additional note for Vulkan**: The Vulkan path of DLSS expects the application to run on a Vulkan version 1.1 or later.
- On each evaluate call (i.e. each frame), provide:
    - The raw color buffer for the frame (in HDR or LDR/SDR space).
    - Screen space motion vectors that are: accurate and calculated at 16 or 32 bits per-pixel; and updated each frame.
    - The depth buffer for the frame.
    - The exposure value (if processing in HDR space).
- Allow for sub-pixel viewport jitter and have good pixel coverage with at least 16 jitter phases (32 or more is preferred).
- Initialize NGX and DLSS using a valid `ProjectID` (See Section 5.2.1).
- The ability to negatively bias the LOD for textures and geometry.

To allow for future compatibility and ease ongoing research by NVIDIA, the engine can also *optionally* provide additional buffers and data. For information on these, see section 9.3.

2.3 DLSS Execution Times & GPU RAM Usage

The exact execution time of DLSS varies from engine to engine and is dependent on the integration. Factors such as how the engine memory manager works and whether additional buffer copies are required can affect the final performance. To give a rough guide, NVIDIA ran the DLSS library using a

command line utility (i.e. without a 3D renderer) across the full range of NVIDIA GeForce RTX GPUs and provides these results as a rough estimate of expected execution times. Developers can use these figures to estimate the potential savings DLSS provides.

In this test scenario:

1. The DLSS library allocates some RAM on the GPU internally. The amount of allocated memory can be queried using NVSDK_NGX_DLSS_GetStatsCallback(). The table below shows approximate amount of RAM allocated depending on output resolution:

	1920x1080	2560x1440	3840x2160	7840x4320
Allocated memory	60.83 MB	97.79 MB	199.65 MB	778.3 MB

   Note that this is a ballpark number - the actual number can be somewhat different. For instance, using InEnableOutputSubrects flags may result in higher memory usage. On the other hand, if your output color buffer has RGBA16 format, DLSS might be able to use it to store some internal temporary data, and then the amount of allocated memory will be smaller.

2. The DLSS algorithm is executed in 16-bit "Performance Mode" where the input is one quarter the number of pixels as the output:
   - 1920x1080 results are generated from an input buffer size of 960x540 pixels.
   - 2560x1440 results are generated from an input buffer size of 1280x720 pixels.
   - 3840x2160 results are generated from an input buffer size of 1920x1080 pixels.

GeForce GPU	1920x1080	2560x1440	3840x2160	7680x4320
RTX 2060 S	0.61 ms	1.01 ms	2.18 ms	10.07 ms
RTX 2080 TI	0.37 ms	0.58 ms	1.2 ms	5.83 ms
RTX 2080 (laptop)	0.56 ms	0.91 ms	1.98 ms	9.09 ms
RTX 3060 TI	0.45 ms	0.73 ms	1.52 ms	7.01 ms
RTX 3070	0.41 ms	0.66 ms	1.36 ms	6.43 ms
RTX 3080	0.32 ms	0.47 ms	0.94 ms	4.25 ms
RTX 3090	0.28 ms	0.42 ms	0.79 ms	3.45 ms

2.4 DLSS Deployment Checklist

During integration and testing, read and follow this document as a whole and before a game or application is released with DLSS included in it, confirm quality is high and at least the following are true.

Item	Confirmed ☑
Full production non-watermarked DLSS library (nvngx_dlss.dll) is packaged in the release build (see section 4.2)	☐
Game specific Application ID is used during initialization (see section 5.2.1)	☐
Mip-map bias set when DLSS is enabled (see section 3.5)	☐
Motion vectors for all scenes, materials and objects are accurate (see section 3.6)	☐
Static scenes resolve and compatible jitter confirmed (see section 3.7)	☐
Exposure value is properly sent each frame (or auto-exposure is enabled) (see section 3.9)	☐
DLSS modes are queried and user selectable in the UI (see section 3.2.1 and RTX UI Developer Guidelines) and/or dynamic resolution support is active and tested	☐

3 DLSS Integration

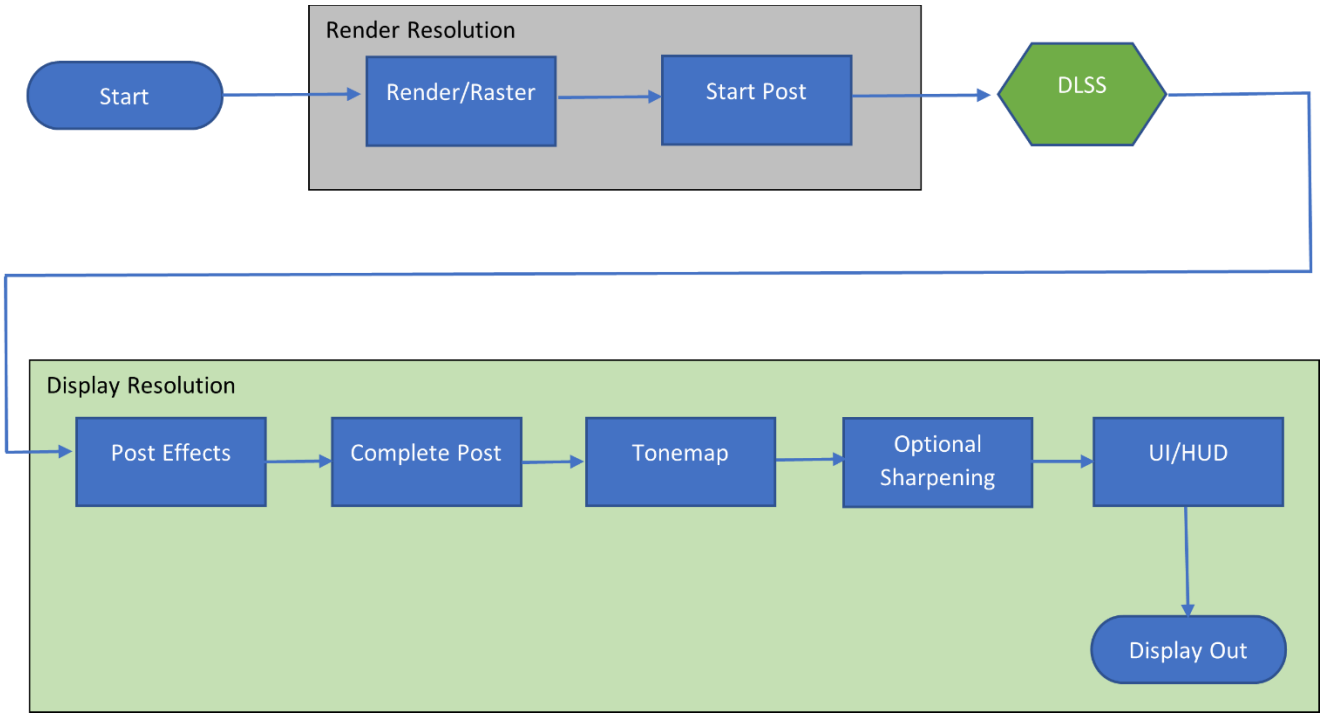
3.1 Pipeline Placement

The DLSS evaluation call must occur during the post-processing phase of the rendering pipeline before tone mapping. For best image quality place DLSS as close to the start of post processing as possible (before any/as few effects as possible are applied to the frame).

3.1.1 DLSS During Early Phase Post Processing

DLSS is best executed at the start of, or very early during, the post processing pipeline. During the call to DLSS, the frame is processed with features enhanced, anti-aliasing applied, and the frame being increased in resolution to the target resolution.

To be clear, if DLSS is placed at the start of post processing, all post effects must be able to handle the increased resolution and not have assumptions that processing can only occur at render resolution.



3.1.2 Color Ranges for LDR and HDR

DLSS can process color data stored as either LDR (aka SDR) or HDR values. The performance of DLSS is improved in LDR mode but there are some caveats.

1. The range of color values for LDR mode **must be** from 0.0 to 1.0.
2. In LDR mode, DLSS operates at lower precision and quantizes the data to 8 bits. For color reproduction to work at this precision, the input color buffer **must be** in a **perceptually linear**

encoding (like sRGB). This is often the case after tonemapping but is not guaranteed (for example if an HDR display panel is detected the tonemapper still outputs to linear space).

3. In LDR mode, the color data must **not** be provided in linear space. If DLSS processes linear colors in LDR mode, the output from DLSS exhibits visible color banding, color shifting or other visual artifacts.

If the input color buffer sent to DLSS **meets** these requirements, then set DLSS to process using LDR by setting "`NVSDK_NGX_DLSS_Feature_Flags_IsHDR`" to "0".

If the input color buffer sent to DLSS is stored in linear space or **does not meet** the requirements above for any reason, set the "`NVSDK_NGX_DLSS_Feature_Flags_IsHDR`" to "1". HDR mode operates internally with high range, high precision colors and can process all luminance values (i.e. there is no upper bound to the acceptable luminance for HDR values).

See section 5.3 for more details on the "`NVSDK_NGX_DLSS_Feature_Flags_IsHDR`" feature flag.

3.2 Integration Overview

DLSS integration comprises the following steps:

1. Initialize NGX and make sure no errors are returned.
2. Check if DLSS is available on the system.
3. Obtain optimal settings for each display resolution and DLSS Execution Mode (see section 3.2.1).
4. Create DLSS feature using helper methods.
5. Evaluate DLSS when upscaling to final resolution.
6. When there are changes to settings which affect DLSS (such as display resolution, toggling RTX, or changing input/output buffer formats) release the current feature and go back to step 3.
7. Perform cleanup and shutdown procedures when DLSS is no longer needed.

IMPORTANT: DLSS should only replace the primary upscale pass on the main render target and should not be used on secondary buffers like shadows, reflections etc.

3.2.1 DLSS Execution Modes

DLSS processes arbitrary input buffer sizes and outputs the result to an output buffer size of the final display resolution. The input resolutions the game should render to and send to DLSS for processing are determined by querying the "DLSS Optimal Settings" for each of the "PerfQualityValue" options. There are currently five "PerfQualityValues" defined:

1. Performance Mode
2. Balanced Mode
3. Quality Mode

4. Ultra-Performance Mode

Depending on the DLSS algorithm in use and the game performance levels, DLSS may enable all or some of the modes listed above. All should be checked but sometimes not all are enabled for a given configuration.

The RTX UI Developer Guideline outlines a DLSS Auto Mode. This is not a specific execution mode, but an additional user-friendly mode to map default mode to the current output resolution. (*Please refer to DLSS Auto mode and DLSS mode Defaults sections*)

In the game settings, **only display those modes that are enabled. Completely hide all other modes.** For the enabled modes, allow the end-user to switch between each enabled mode changing the render target resolution to match.

After querying the DLSS Optimal Setting, if more than one mode is enabled, the default selection (unless the user has chosen a specific mode) should be “Auto”, “Quality”, “Balanced”, “Performance”, “Ultra-Performance”.

For information on how to display the user facing DLSS mode selection, please see the “NVIDIA RTX UI Developer Guidelines” (the latest version is on the GitHub repository in the “docs” directory).

For more details on DLSS Optimal Settings, see section 5.2.8.

3.2.2 Dynamic Resolution Support

DLSS supports dynamic resolution whereby the input buffer can change dimensions from frame to frame whilst the output size remains fixed. As such, if the rendering engine supports dynamic resolution, DLSS can be used to perform the required upscale to the display resolution.

NOTE: If the output resolution (aka display resolution) changes, DLSS must be reinitialized.

```
static inline NVSDK_NGX_Result NGX_DLSS_GET_OPTIMAL_SETTINGS(
    NVSDK_NGX_Parameter *pInParams,
    unsigned int InUserSelectedWidth,
    unsigned int InUserSelectedHeight,
    NVSDK_NGX_PerfQuality_Value InPerfQualityValue,
    unsigned int *pOutRenderOptimalWidth,
    unsigned int *pOutRenderOptimalHeight,
    unsigned int *pOutRenderMaxWidth,
    unsigned int *pOutRenderMaxHeight,
    unsigned int *pOutRenderMinWidth,
    unsigned int *pOutRenderMinHeight,
    float *pOutSharpness)
```

To use DLSS with dynamic resolution, initialize NGX and DLSS as detailed in section 5.3. During the DLSS Optimal Settings calls for each DLSS mode and display resolution, the DLSS library returns the “optimal” render resolution as `pOutRenderOptimalWidth` and `pOutRenderOptimalHeight`. Those values must then be passed exactly as given to the next `NGX_API_CREATE_DLSS_EXT()` call.

DLSS Optimal Settings also returns four additional parameters that specify the permittable rendering resolution range that can be used during the DLSS Evaluate call. The `pOutRenderMaxWidth`,

pOutRenderMaxHeight and pOutRenderMinWidth, pOutRenderMinHeight values returned are inclusive: passing values between as well as exactly the Min or exactly the Max dimensions is allowed.

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

typedef struct NVSDK_NGX_D3D11_DLSS_Eval_Params
{
...
    NVSDK_NGX_Dimensions                InRenderSubrectDimensions;
...
} NVSDK_NGX_D3D11_DLSS_Eval_Params;


static inline NVSDK_NGX_Result NGX_D3D11_EVALUATE_DLSS_EXT(
    ID3D11DeviceContext *pInCtx,
    NVSDK_NGX_Handle *pInHandle,
    NVSDK_NGX_Parameter *pInParams,
    NVSDK_NGX_D3D11_DLSS_Eval_Params *pInDlssEvalParams)
```

The DLSS Evaluate calls for the supported graphics APIs can accept InRenderSubrectDimensions as an additional Evaluation parameter. That specifies the area of the input buffer to use for the current frame and can vary from frame to frame (thus enabling dynamic resolution support).

The subrectangle can be offset in the buffer using InColorSubrectBase (or the other *SubrectBase parameters) which specify the top-left corner of the subrect.

If the the InRenderSubrectDimensions passed to DLSS Evaluate are not in the supported range (returned by the DLSS Optimal Settings call), the call to DLSS Evaluate fails with an NVSDK_NGX_Result_FAIL_InvalidParameter error code.

NOTE: Not all combination of PerfQuality mode, resolution and ray tracing, support dynamic resolution. When dynamic resolution is not supported, the call to NGX_DLSS_GET_OPTIMAL_SETTINGS returns the same values for both the minimum and maximum render resolutions. This is also the case for older DLSS libraries that did not support dynamic resolution.

3.2.2.1 Dynamic Resolution Caveats

When using dynamic resolution with DLSS, it is important to:

1. Maintain a consistent texture mipmap bias as detailed in section 3.5. To maintain correct display resolution sampling, the mip level should be updated each time the render resolution changes (potentially every frame depending on the update rate of the dynamic resolution system).

   If changing the mip bias this often is not feasible, the developer can use an estimated bias or have a limited set of mipmap biases. Some experimentation may be needed to maintain on-screen texture sharpness.

2. To guarantee the accuracy of the DLSS image reconstruction, the aspect ratio of the render size must stay constant with the final display size. NVIDIA suggests calculating the projection matrix with this in mind.

```
// When rendering to a different resolution than the display resolution, ensure
// geometry appears similar once it is scaled up to the whole window/screen
float aspectRatio = displaySize.x / displaySize.y;

// Compute the projection matrix using this aspect ratio (not the Render ratio)
float4x4 projection =
    perspProjD3DStyle(dm::radians(m_CameraVerticalFov), aspectRatio, zNear, zFar);
```

3.3 Supported Formats

DLSS uses formatted reads so should handle most input buffer formats. With that said, DLSS expects the following inputs:

1. Color input buffer (the main frame): any supported buffer format for the API.
2. Motion vectors: RG32_FLOAT or RG16_FLOAT (for more information see section 3.6.1)
3. Depth buffer: any format with one channel in it (for instance R32_FLOAT or D32_FLOAT), as well as any depth-stencil format (for instance D24S8)
4. Output buffer (the destination for the processed full resolution frame): any supported buffer format for the API.
5. Previous Output buffer (used for frame history and accumulation): optional, but if provided should be RGBA16F.

3.4 Resource States

The game or application calling DLSS must ensure that the buffers passed to DLSS are setup with the right usage flags and are in the correct state at the evaluation call. DLSS requires that the:

1. Input buffers (e.g. color, motion vectors, depth and optionally history, exposure etc.) be in **pixel shader read state** (also known as a **Shader Resource View**, **HLSL "Texture"** or in Vulkan as a **"Sample Image")**. This also means that in the case of Vulkan these have to be created with the "**VK_IMAGE_USAGE_SAMPLED_BIT**" usage flag.
2. The Output buffer be in **UAV state** (also known as an **HLSL RWTexture** or in Vulkan as a "**Storage Image**"). This also means that in case of D3D12 it has to be created with the "**D3D12_RESOURCE_FLAG_ALLOW_UNORDERED_ACCESS**" flag and in case of Vulkan with the "**VK_IMAGE_USAGE_STORAGE_BIT**" usage flag.

After the evaluate call, DLSS accesses and processes the buffers and may change their state but always transitions buffers back to these known states.

3.5 Mip-Map Bias

When DLSS is active, the rendering engine must set the mip-map bias (sometimes called the texture LOD bias) to a value lower than 0. This improves overall image quality as textures are sampled at the *display* resolution rather than the lower render resolution in use with DLSS. NVIDIA recommend using:

```
   DlssMipLevelBias = NativeBias + log2(Render XResolution / Display XResolution) - 1.0 +
epsilon
```

As an example, if you use the optimal settings for DLSS Performance mode, the ratio `Render XResolution / Display XResolution = 0.5`. As a result, the recommended Miplevel bias is `-2.0` for this Performance mode.

🛑 Due to the temporal nature of DLSS, applying the recommended negative LOD Bias to the input can sometimes lead to increased temporal stability, in the form of flickering and/or moiré. This will be highly dependent on the scene content such as texture detail. For those scenes and objects where the flickering can become distracting it will be recommended to adjust the LOD bias to be less aggressive. You can typically try to adjust it on the problematic content but without going above `log2(Render XResolution / Display XResolution)`. Anything less than the recommended value will tend to be softer/blurrier, so this is going to be a tradeoff between preservation of detail and aliasing.

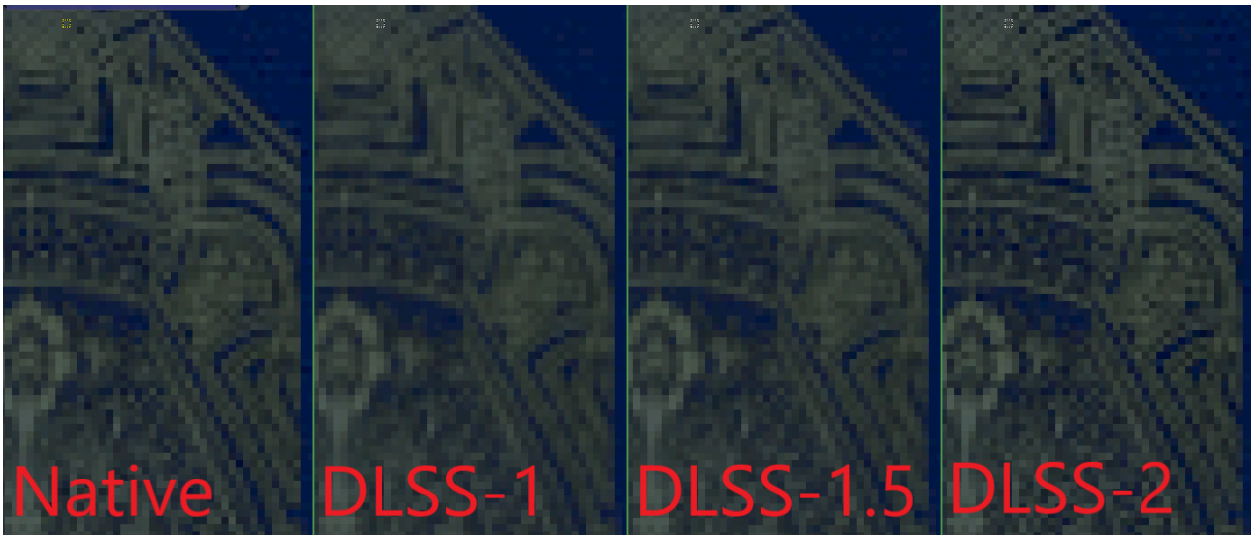


Figure 1. In the example above, we can see that DLSS with a -2.0 Texture LOD Bias is the closest to the native rendering in terms of preservation of details.

NOTE: Carefully check texture clarity when DLSS is enabled and confirm that it matches the texture clarity when rendering at native resolution with the default AA method. Pay attention to textures with text or other fine detail (e.g. posters on walls, number plates, newspapers etc).

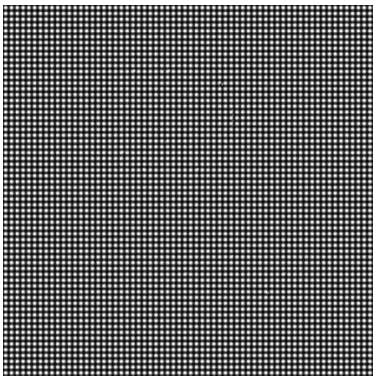
If there is a negative bias applied during native resolution rendering, some art assets may have been tuned for the default bias. When DLSS is enabled the bias may be too large or too small

compared to the default leading to poor image quality. In such case, adjust the “epsilon” for the DLSS mip level bias calculation.

NOTE: Some rendering engines have a global clamp for the mipmap bias. If such a clamp exists, disable it when DLSS is enabled.

3.5.1 Mip-Map Bias Caveat: High-Frequency Textures

If the mip levels are biased on textures with high frequency patterns, this can lead to artifacts when DLSS tries to reconstruct the full resolution frame. In particular, if trying to simulate an “LED screen”, a “stock ticker” or something similar that uses a texture such as the one below, override the mip-map bias for that material and leave it at the default.



Example high frequency “LED Screen” texture

3.6 Motion Vectors

DLSS uses per-pixel motion vectors as a key component of its core algorithm. The motion vectors map a pixel from the current frame to its position in the previous frame. That is, when the motion vector for the pixel is added to the pixel's current location, the result is the location the pixel occupied in the previous frame.

IMPORTANT: Incorrect or poor precision motion vectors are the most common cause of visual artifacts when DLSS is enabled. Please use a visualizer (such as the debug overlay – see section 8.2) to check motion vectors any time you notice visual artifacts with DLSS.

3.6.1 Motion Vector Format & Calculations

Motion vectors must be sent as floats with a render target of format RG32_FLOAT or RG16_FLOAT. The X and Y values of the 2D screen-space motion vectors are stored in the red and green channels of the texture in 32-bit or 16-bit floating point format (depending on the format).

The values of each motion vector represent the amount of movement given as the number of pixels calculated in screen space (ie the amount a pixel has moved at the **render resolution**) and assume:

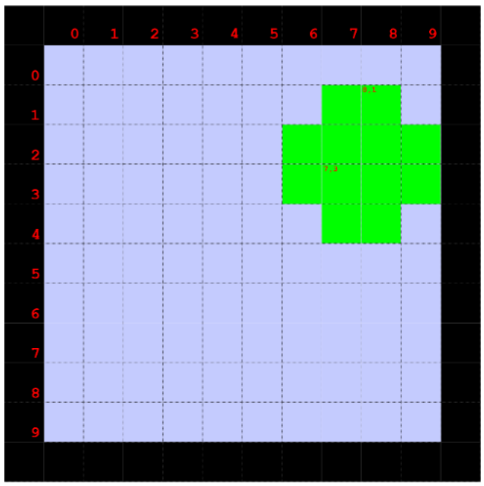
1. Screen space pixel values use [0,0] as the **upper left** of the screen and extend to the full resolution of the **render** target. As an example, if the render target is a 1080p surface, the pixel at the bottom right is [1919,1079].

a. DLSS can also optionally accept full resolution motion vectors which are calculated at display resolution. See section 3.6.2 for more information.

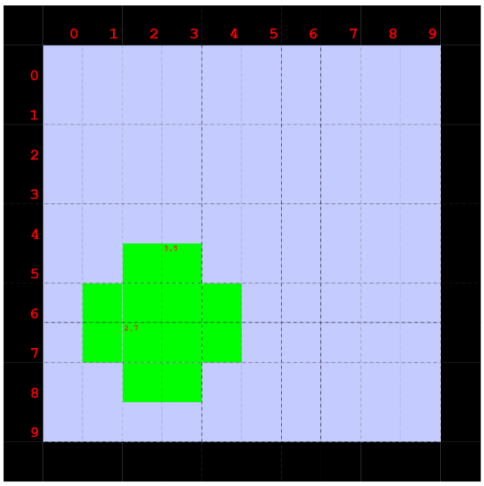
2. Motion vectors can be positive or negative (depending on the movement of the scene objects, camera and screen).

3. Motion vectors can include full and partial pixel movements (i.e. [3.0f,-1.0f] and [-0.65f,10.1f] are both valid).

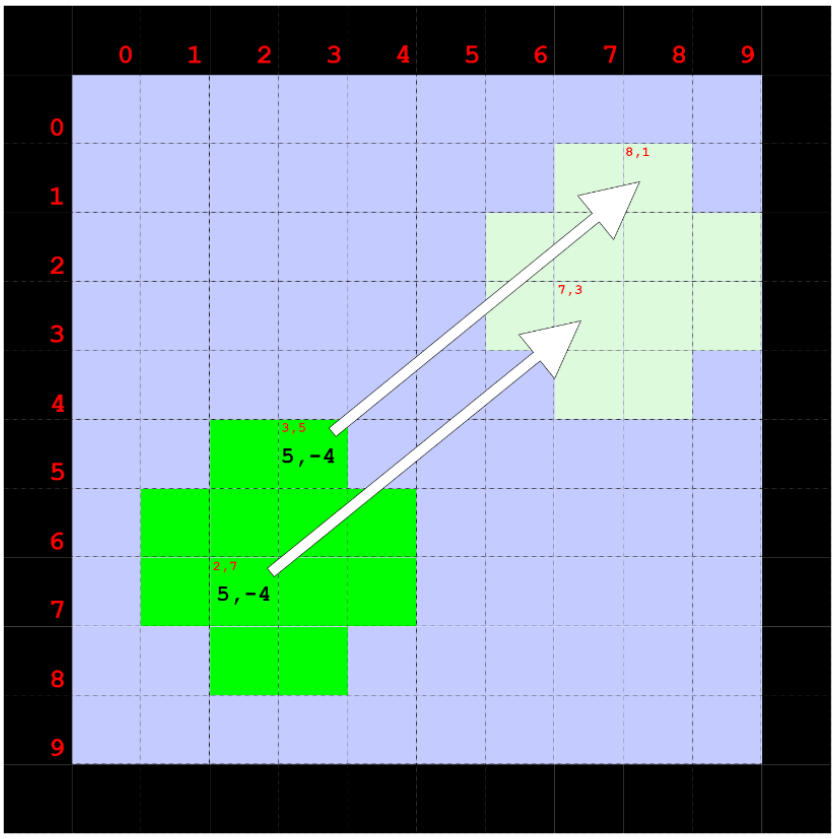
NOTE: If the game or rendering engine uses a custom format for motion vectors, it **must** be decoded **before** calling DLSS.



Frame "N-1"



Frame "N"



Example motion vector values for Frame "N"

3.6.1.1 Dense Motion Vector Resolve Shader

For games that use Unreal Engine 4 or another engine that calculates motion vectors using geometry movement, DLSS requires a slight tweak to the way motion vectors are calculated. Apply a pixel shader like the below to the default motion vector buffer.

NOTE: If you use or have merged NVIDIA's custom branch of UE4 with DLSS integrated, this change (or something very similar) has already been applied.

```
Texture2D DepthTexture;
Texture2D VelocityTexture;

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

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

```
float3 HomogenousToEuclidean(float4 V)
{
  return V.xyz / V.w;
}

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

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

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

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

    if (PrevClipPos.w > 0)
    {
      float2 PrevClip = HomogenousToEuclidean(PrevClipPos).xy;
      Velocity = ClipPos.xy - PrevClip.xy;
    }
  }

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

3.6.2 Motion Vector Flags

To ease integration into a wider variety of engines, DLSS accepts motion vectors in several ways. To switch between them (depending on rendering engine requirements) set the flags appropriately during DLSS Feature Creation (see section 5.3).

1. `NVSDK_NGX_DLSS_Feature_Flags_MVLowRes`: Motion vectors are typically calculated at the same resolution as the input color frame (i.e. at the render resolution). If the rendering engine supports calculating motion vectors at the display/output resolution **and** dilating the motion vectors, DLSS can accept those by setting the flag to "0". This is preferred, though uncommon, and can result in higher quality antialiasing of moving objects and less blurring of small objects and thin details.

   For clarity, if standard input resolution motion vectors are sent they do not need to be dilated, DLSS dilates them internally. If display resolution motion vectors are sent, they must be dilated.

2. `NVSDK_NGX_DLSS_Feature_Flags_MVJittered`: Set this flag to “1” when the motion vectors **do** include sub-pixel jitter. DLSS then internally subtracts jitter from the motion vectors using the jitter offset values that are provided during the “Evaluate” call. When set to “0”, DLSS uses the motion vectors directly without any adjustment.

3.6.3 Motion Vector Scale

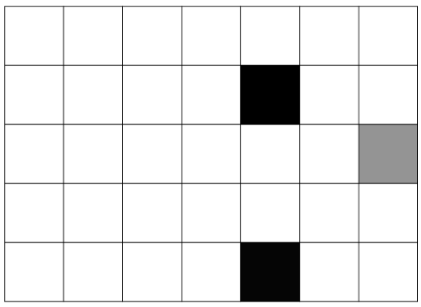
There are cases when an engine has existing motion vectors that do not match the scale or direction required for DLSS. One example, if the motion vectors are pointing in the direction of motion rather than towards the previous frame. Another example, if the motion vector values are in UV space rather than pixel space.

To allow this data to be used directly, DLSS provides motion vector scale parameters used during evaluation to enable some modification of the motion vector values when passed to DLSS. They are generally set from the eval parameters struct `NVSDK_NGX_D3D11_DLSS_Eval_Params` via `InMVScaleX` and `InMVScaleY` members. These should be set to 1.0 if motion vectors do not need to be scaled and should never be 0.0f. (see section 5.4 or *nvsdk_ngx_helpers.h*).

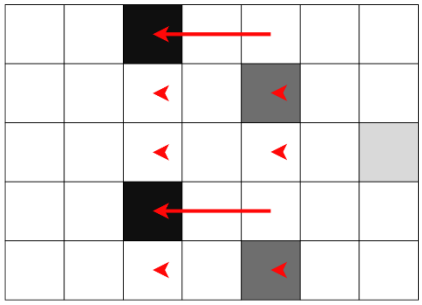
3.6.4 Conservative raster on Motion Vector for thin features

If you are rendering objects that have thin geometric features they tend to get in and out of view due to the low resolution of the input buffers and the rasterizer "missing out" on some of that geometry. This results in visible holes when that object is rendered.

DLSS can sometimes help reconstruct some of that missing geometry but in order to do that it needs accurate motion vectors. Because those features went out of view, neither the color nor the motion vector for that missing feature is present in the input that is sent to DLSS. In that scenario shown below, DLSS will incorrectly associate the previously drawn image of that object with the motion vector of the background. In that scenario, instead of trying to mend the holes in the object drawing, those holes will persist.



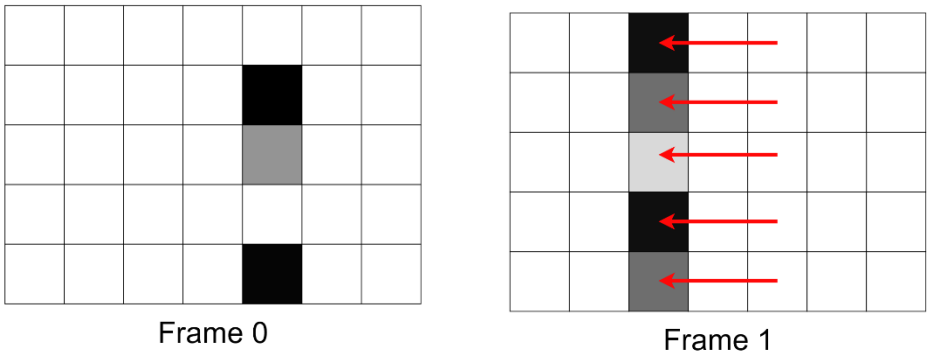
Frame 0



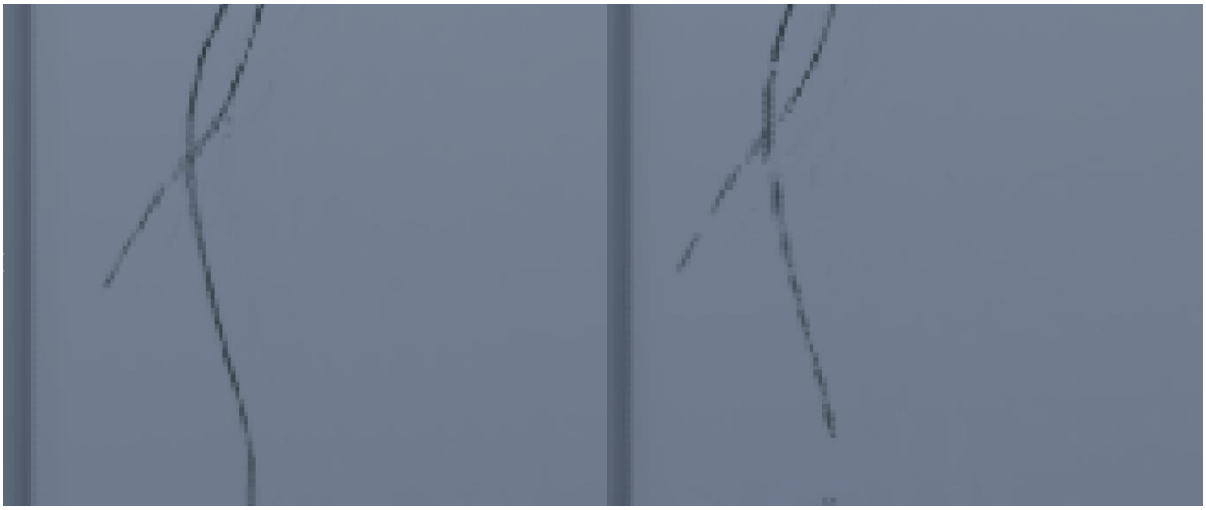
Frame 1

A potential workaround for that specific problem is to draw the motion vector (only!) for that object using the feature called Conservative raster (e.g., introduced in Direct3D 11.3 and Direct3D 12). The Conservative rasterization ensures that if a primitive touches a pixel, even by the slightest amount, it gets drawn. So, if the color information does not pick up that the object was present, the motion vector drawn with conservative raster will pick up that information.

As a result of using this, the motion of the previously drawn parts of the object will show the correct amount that the object moved on the screen, and it should allow DLSS to have a better chance at reconstructing.



Here's an in-game comparison. In this we can see that the left is using conservative raster for its motion vector and as a result has slightly less visible holes during reconstruction than the right image where we didn't make use of conservative raster.



Please note that this is not necessarily the panacea in fixing those flickering objects as DLSS will use many hints from the input data to better reconstruct the image and sometimes it may think, incorrectly, that it's best to drop that reprojected history.

3.7 Sub-Pixel Jitter

DLSS includes several temporal components in its anti-aliasing, feature enhancement and upscaling algorithm. To achieve high image quality, DLSS emulates a higher sample rate by having the rendering engine generate a subset of the desired sample locations each frame and temporally integrating them to produce the final image. The renderer provides the additional samples by applying sub-pixel jitter to the viewport or main camera view so to vary the rasterized frame over time.

Put another way, if we had four or more frames of rendered and rasterized pixels with no motion, the image DLSS produces should be identical to a 4x super-sampled image. This is the goal for DLSS but is not always achievable in practice.

This section describes:

1. How to choose what sample position to use for each frame.
2. How to render using the different sample offsets.
3. How to send the jitter information to DLSS.

3.7.1 Jitter Sample Patterns

There are many different patterns that can be used to apply jitter to a 3D scene. For best results, the jitter pattern must have good coverage across the entire pixel. NVIDIA found that using a Halton sequence (https://en.wikipedia.org/wiki/Halton_sequence) for the jitter pattern provided the best results. As such, the deep learning training undertaken for DLSS uses Halton for the training data.

If possible, use a Halton sequence for sub-pixel jitter when DLSS is enabled even if a different pattern is used when TAA or an alternate AA mode is enabled. If other patterns are used, DLSS should still function correctly (provided the rest of the jitter requirements in this section are followed) but NVIDIA has not tested other patterns.

3.7.1.1 Pattern Phases

In addition to the type of sequence, it is important to cycle through the pattern effectively so that, over time, there is good coverage of the entire pixel area.

A good choice for the number of phases (i.e. the number of unique samples in the pattern before repeating) is:

```
Total Phases = Base Phase Count * (Target Resolution / Render Resolution) ^ 2
```

The Base Phase Count is the number of phases often used for regular temporal anti-aliasing (like TXAA or TAA). A good starting value for the Base is “8” which provides good pixel coverage when no scaling is applied. For DLSS, the Base is then scaled by the scaling ratio of the pixel area.

For instance, with a render target of 1080p and a target output of 2160p (aka 4k), the 1080p pixel is four times the size of a 4K pixel. Hence, four times as many phases are required to make sure each 4K pixel is still covered by 8 samples. So, in this example, the total number of phases would be 32:

```
Total Phases = 8 * (2160 / 1080) ^ 2
```

3.7.2 Rendering with Jitter Offsets

Depending on the rendering engine used, how to actually generate a jittered frame may vary. If the renderer already includes TAA or TXAA, examine how jitter is applied when that is in use and do the same thing when DLSS is enabled.

Speaking in general terms, to render with jitter, apply human imperceptible movement to the camera or viewport such that when the 3D scene is rasterized there are slight changes to the resulting frame (especially edges). A typical procedure for applying the jitter offset is to modify the renderer's projection matrix:

```
// Each frame, update the ProjectionJitter values with those from the jitter sequence
// for the current Phase. Then offset the main projection matrix for the frame.
ProjectionMatrix.M[2][0] += ProjectionJitter.X;
ProjectionMatrix.M[2][1] += ProjectionJitter.Y;
```

Using this method should provide a shift in the camera-space coordinates rather than shifting the world-space coordinates and then projecting.

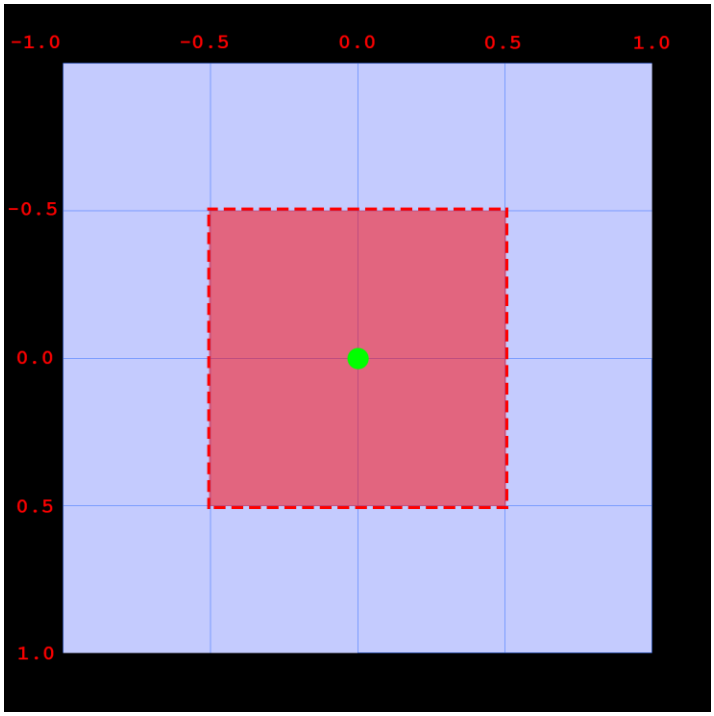
3.7.3 Required Jitter Information

To correctly integrate the current frame with those from the past, DLSS must know how the jitter for this frame was applied. At each call to DLSS evaluate, set the current frame's jitter amount as the jitter offsets in the parameter list (using the correct type structure for the graphics API that is in use):

```
typedef struct NVSDK_NGX_D3D11_DLSS_Eval_Params
{
    NVSDK_NGX_D3D11_Feature_Eval_Params Feature;
    /* Jitter Offsets in pixel space */
    float                          InJitterOffsetX;
    float                          InJitterOffsetY;
}
```

The jitter offset values:

1. Should always be between -0.5 and +0.5 (as jitter should always result in movement that is **within** a source pixel).
2. Are provided in pixel-space at the render target size (not the output or target size – jitter **cannot** be provided at output resolution as is optionally possible with motion vectors).
3. Represent the jitter applied to the projection matrix for viewport jitter irrespective of whether the offsets were also applied to the motion vectors or not.
    a. If the motion vectors do have jitter applied to them, be sure to set the appropriate flag during the DLSS Create call as detailed in section 3.6.2.
4. Use the same co-ordinate and direction system as motion vectors (see 3.6.1 above) with "`InJitterOffsetX == 0`" and "`InJitterOffsetY == 0`" meaning no jitter was applied.



*Jitter offset scale diagram. Shaded red area **is one full pixel**.*
The green marking is the origin with red outline the range of expected values.

3.7.4 Troubleshooting Jitter Issues

If there are issues such as screen shaking, distant objects not resolving, a “screen door” appearance on the output, or static objects (especially thin objects and fine texture detail) appearing “fuzzy”, there may be an issue with:

1. How jitter is applied in the renderer.
2. The supplied motion vectors.
3. The jitter offset values sent to DLSS.

To assist in debugging, NVIDIA has added several tools and visualizers to the SDK version of the DLSS library. For information on how to use these tools and debug issues with jitter, see section 8.3.

3.8 Depth Buffer

To assist in better object tracking and pixel alignment, DLSS uses the depth buffer generated by the engine during rasterization. The algorithm assumes that the near plane is 0.0 and the far plane is 1.0 (but this can be inverted, see below).

We recommend using nearest upsampling of gbuffer data for post processing shaders that require it.

3.8.1 Depth Buffer Flags

To ease integration into a wider variety of engines, DLSS accepts different depth configurations. Depending on the renderer, set the flags appropriately during DLSS Feature Creation (see section 5.3).

1. `NVSDK_NGX_DLSS_Feature_Flags_DepthInverted`: set this flag to "1" if the engine uses depth with a near plane at 1.0 and far plane at 0.0.

3.9 Exposure Parameter

When processing in HDR, DLSS needs the renderer's exposure value for the current frame. This is the value which when multiplied to the input color values brings middle gray to an expected level. This is typically the same value provided to the renderer's tonemapper to consistently deal with the compression of HDR to LDR colors (e.g. when outputting to a standard LDR monitor).

NOTE: A basic exposure value may be calculated via the following function (where `MidGray` is the amount of reflected light which represents the perceptual middle between full reflected brightness and full absorption - a value of "0.18" is typical for `MidGray`; and `AverageLuma` is the average luminance for the entire frame):

```
ExposureValue = MidGray / (AverageLuma * (1.0 - MidGray))
```

The renderer must provide DLSS with the correct `ExposureValue` during each DLSS evaluate call using a 1x1 texture referenced in the `pInExposureTexture` parameter. Only the first channel is sampled in the texture so multiple formats will work but something such as `R16F` is preferred.

NOTE: This value is sent to DLSS as a 1x1 texture to avoid a round-trip to the CPU as the value is typically calculated per-frame on the GPU.

If `ExposureValue` is missing or DLSS does not receive a correct value (which can vary based on eye adaptation in some game engines), DLSS may produce a poor reconstruction of the high-resolution frame with artifacts such as:

1. Ghosting of moving objects.
2. Blurriness, banding, or pixilation of the final frame or it being too dark or too bright.
3. Aliasing especially of moving objects.
4. Exposure lag.

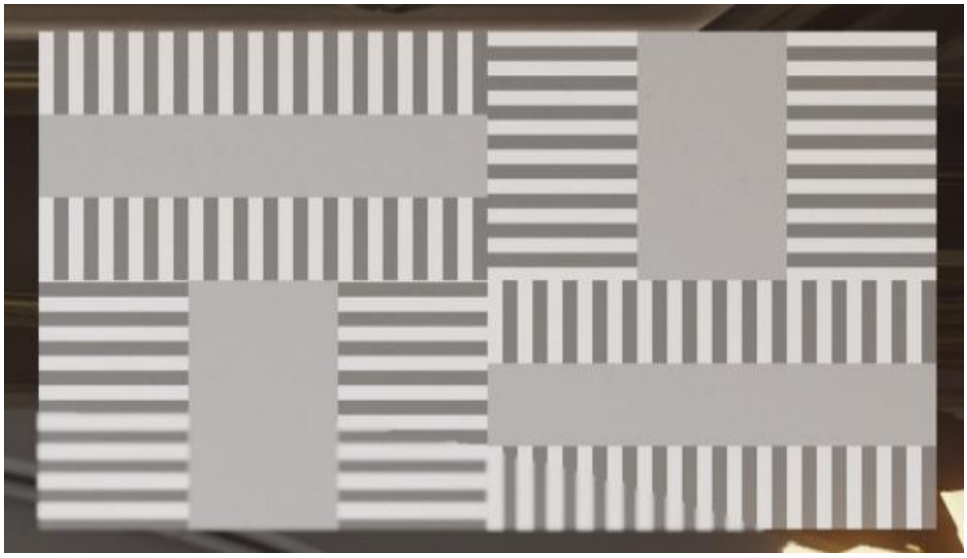
If one or the above issues are present even when `ExposureValue` is provided, a scale factor and/or bias may be required.

3.9.1 Diagnosing bad exposure with the visualizer

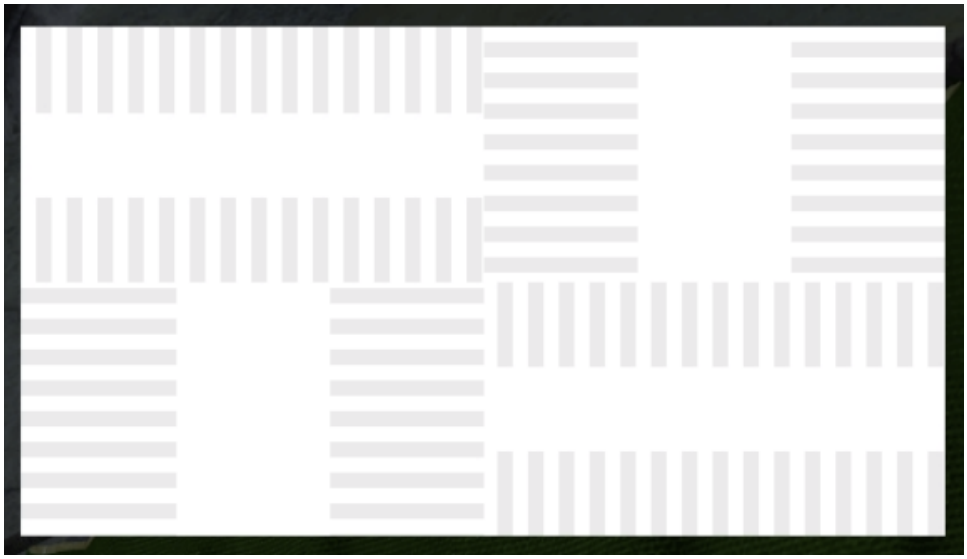
With the recent SDK we added a visual aide to help diagnose exposure scale problems.

You can display that pattern in the top right corner by cycling through the debug visualizers (see section 8.2).

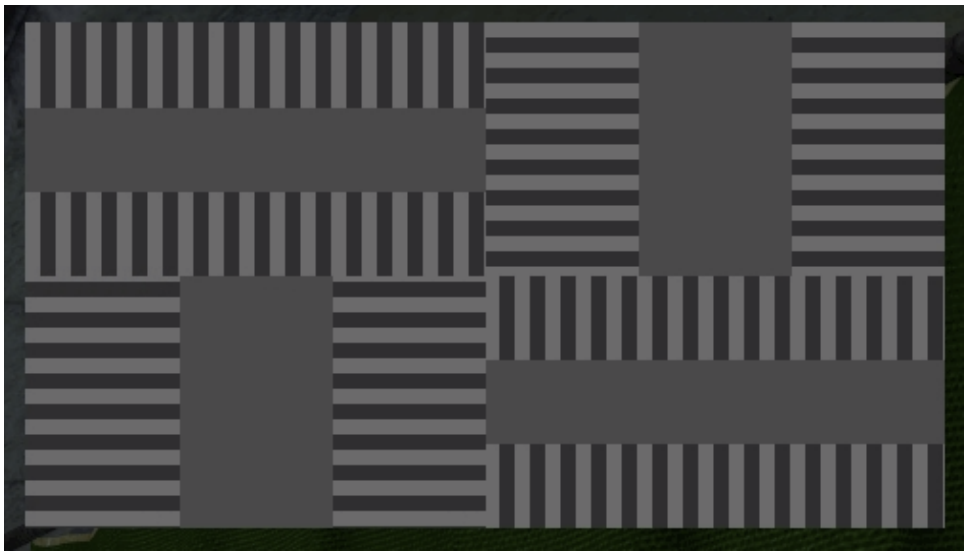
If the exposure scale value (and pre-exposure) that is being passed to DLSS is accurate you should be able to see the pattern as below. The pattern is centered around a brightness value of 187 (on a scale of 0 and 255 in sRGB 8 bit space).



If on the other hand the value that you are passing through the exposure scale texture is undervalued, leading DLSS to see a vastly underexposed image, you should see this pattern below, where everything is very bright:



And on the opposite side, if you are passing a value through the exposure scale that is overestimated, that will lead DLSS to see an overexposed image which again will be visible with this pattern below, where everything is very dark:



3.9.2 Pre-Exposure Factor

Some engines implement additional exposure tuning by pre-multiplying frames with a "pre-exposure factor" that is later removed (divided out) during tonemapping. Due to the DLSS algorithm's heavy use of previous frame history, if this factor is not accounted for, it can lead to poor reconstruction of the high-resolution frame as the inputs are, in effect, "double-exposed". Visual artifacts are the same as those seen if a bad `pInExposureTexture` texture is used (see section 3.9).

Please note that this is not a common use-case and developers are encouraged to confirm visual artifacts are not being caused by issues such as incorrect sub-pixel jitter or bad motion vectors. Please check them first and check with a core rendering engine lead as to whether a pre-exposure value is used or whether the input buffer passed to DLSS is already "eye adapted".

If there is a pre-exposure value, you must pass it to DLSS during every DLSS evaluation call using the `InPreExposure` parameter (available from DLSS v2.1.0).

NOTE: The pre-exposure value is optional and is sent to DLSS is a CPU-side float passed via the parameter maps. It is not a texture like the main exposure value.

3.10 Auto-exposure

Using the main exposure parameter (see above) is the preferred method, however in some situations, taking advantage of the optional auto-exposure feature in the DLSS library can provide improved results. To use the auto-exposure values computed inside the DLSS library instead of `pInExposureTexture`, the developer must:

1. Set the `NVSDK_NGX_DLSS_Feature_Flags_AutoExposure` parameter during DLSS Feature Creation

NOTE: Depending on the version of the DLSS algorithm, GPU and output resolution, there may be a small increase in processing time (~0.02ms) when auto-exposure is enabled.

Auto-exposure may be toggled using the DLSS SDK DLL by pressing the CTRL+ALT+Y hotkey while in the game (overriding the flag passed at creation time).

3.11 Additional Sharpening

The deep learning model used in DLSS is trained to produce sharp images, however, for those developers wishing to provide a sharper image, the NVIDIA Image Scaling SDK is now publicly available as part of the DLSS SDK and in the NVIDIA Image Scaling repository in Github. It is highly recommended that developers who use the sharpening filter provide an end-user slider to control the sharpening value. (Please refer to the Image Scaling documentation for more information)

NOTE: The sharpening and softening pass provided in previous versions of the SDK are now deprecated. All developers are encouraged to use the Image Scaling SDK via the DLSS SDK or directly on GitHub.

3.12 Scene Transitions

DLSS is built around spatio-temporal reconstruction techniques and leverages temporal information gathered from previous frames. The algorithm can be confused if there is a complete (or significant) change in the scene from frame to frame. This can result in visual artifacts such as lagged transitions or "ghost images" from the previous scene carrying over to the new scene.

To avoid these artifacts, during the DLSS Evaluation call for the first frame after a major transition, set the `InReset` parameter from the `DLSS_Eval` parameter list to a non-zero value. This instructs DLSS to void its internal temporal component and restart its processing with the incoming inputs.

NOTE: Improper use of this can result in temporal flickering, heavy aliasing or other visual artifacts.

3.13 VRAM Usage

Modern rendering engines often track VRAM usage and modify rendering parameters and art assets based on occupancy. To assist with this tracking, DLSS offers a NVSDK_NGX_DLSS_GetStatsCallback() function which can be used to query the amount of VRAM currently allocated internally by DLSS. Most convenient way of using the function is by using NGX_DLSS_GET_STATS() inline helper.

Note that the amount of memory that the function returns is a global variable. Therefore, if you created multiple DLSS instances, the function returns TOTAL amount of memory currently used by all instances. It also includes any memory that DLSS may cache internally. Therefore, even if you released all DLSS instances, you may still see non-zero value of allocated memory. All cached memory gets released when Shutdown() is called though.

Also Note that, by default, when you call ReleaseFeature(), the memory used by the feature isn't released – it's being cached internally for such case when you will immediately re-create feature again – this memory would then be reused. However, DLSS supports a hint

NVSDK_NGX_Parameter_FreeMemOnReleaseFeature which will force memory release on each ReleaseFeature() call. This hint can be set as follows:

```
NVSDK_NGX_Parameter_SetI(ngxParams, NVSDK_NGX_Parameter_FreeMemOnReleaseFeature, 1)).
```

3.14 Biasing the Current Frame

NVIDIA is continuing to research methods to improve feature tracking in DLSS. From time to time, the DLSS feature tracking quality can be reduced and DLSS may then "erase" a particular feature or can produce a "ghost" or trail behind a moving feature. This can occur on:

1. Small particles (like snow or dust particles).
2. Objects that display an animated/scrolling texture.
3. Very thin objects (such as power lines).
4. Objects with missing motion vectors (many particle effects).
5. Disoccluding objects with motion vectors that have very large values (such as a road surface disoccluding from under a fast-moving car).

If a problematic asset is discovered during testing, one option is to instruct DLSS to bias the incoming color buffer over the colors from previous frames. To do so, create a 2D binary mask that flags the non-occluded pixels that represent the problematic asset and send it to the DLSS evaluate call as the `BiasCurrentColor` parameter. The DLSS model then uses an alternate technique for pixels flagged by the mask.

The mask itself should be the same resolution as the color buffer input, use R8G8B8A8_UNORM /R16_FLOAT/R8_UNORM format (or any format with an R component) and have a value of "1.0" for masked pixels with all other pixels set to "0.0".

NOTE: Only use this mask on problematic assets after the DLSS integration has been completed and confirmed as fully functional. There may be increased aliasing within the mask borders.

3.15 Multi-view and Virtual Reality Support

DLSS natively supports multiple views, including virtual reality (VR) by creating multiple DLSS instances with one associated to each view (which, in the case of VR, will likely correspond to each eye).

To use DLSS with multiple views, create multiple DLSS instances with the standard call to `NGX_<API>_CREATE_DLSS_EXT`) and have the DLSS evaluation calls be made on the DLSS instance handle associated with the view currently being upsampled.

Many VR titles render both eyes to the same render target. In that case, use sub-rectangles to restrict DLSS to the appropriate region of interest in the render target (see section 5.4 for details). The sub-rectangle parameters specify:

1. The "base" (top-left corner) of the sub-rectangle.

2. The size of the sub-rectangles assumed to be the input/output dimensions specified at instance handle creation time or the input dimensions specified at evaluation time, in the case of the *input sub-rectangles*, if dynamic resolution is being used (see section 3.2.2 for details).

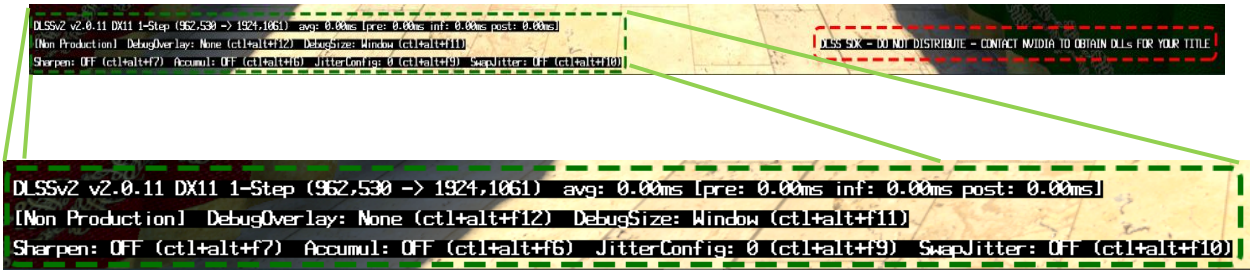
3. The size of the *output sub-rectangle* is assumed to be the output dimensions specified at instance handle creation time regardless of whether dynamic resolution is in use.

NOTE: To use sub-rectangles to specify a subregion of the *output resource* for DLSS to output to, the flag `InEnableOutputSubrects` in `NVSDK_NGX_DLSS_Create_Params` must be set to `true` at DLSS instance handle creation time (see section 5.3 for details). Sub-rectangles are supported on the *input resources* regardless of whether `InEnableOutputSubrects` is set.

3.16 Current DLSS Settings

The DLSS library included as part of the DLSS SDK includes a permanent watermark (outlined in red below) and an optional on-screen display of certain information about the current DLSS parameters and settings (outlined in green below). These are provided for debugging purposes; the watermark is **not** included in the production library.

NOTE: End-user machines do not show the DLSS parameters and debug lines but some DLSS information like the version number does appear with a production DLSS library if the developer's or tester's machine has the appropriate registry key set (see next paragraph). This is not a bug.



To enable the DLSS debug information lines (outlined in green above), locate the registry file "`dlss_debug_onscreendisplay_on.reg`" that was in the SDK archive or is in the "`Utils`" folder on the GitHub repository. From Windows Explorer, double-click the file and merge the changes to the Windows Registry. To disable the debug lines, follow the same procedure with the "`dlss_debug_onscreendisplay_off.reg`" file.

For develop and debug builds, registry key that enables DLSS on-screen indicator is:

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

It must be DWORD. Any positive value enables DLSS indicator for Develop and Debug builds of DLSS. On the contrary, Release build of DLSS requires value **1024** to enable DLSS indicator.

For Linux, set environment variable __NGX_SHOW_INDICATOR=1.

3.16.1 DLSS Information Lines & Debug Hotkeys

From left to right and top to bottom, the DLSS debugging information displays:

1. The DLSS version (in this example v2.0.11).
2. Current 3D API in use (DirectX 11, DirectX 12 or Vulkan).
3. Render to target scaling factor. The first number is the input buffer size (i.e. the current render target size); the second number is the output buffer size (i.e. the display resolution).
4. The remainder of the line shows timing information (depending on the driver in use and setup of the system, timing may display zero or estimates of execution time).
5. The debugging hotkeys begin on the second line and continue onto the third:
    5.1. The current debug overlay name is output first. To cycle through the available debug overlays, use CTRL+ALT+F12. For a list of debug overlays and how to use them, see section 8.2.
    5.2. To toggle between the debug overlay displayed as a window in the top right of the screen and a fullscreen display, use CTRL+ALT+F11.
    5.3. To toggle the debug “Accumulation Mode”, use CTRL+ALT+F6. For more information on how to use the Accumulation Mode, see section 8.3.
    5.4. To toggle NaN visualization, use CTRL+ALT+O. NaNs will be displayed as bright red. To debug NaNs in the input, cycle through the debug overlays as described above.
    5.5. To cycle negation of the different jitter offsets, use CTRL+ALT+F10. For more information on debugging jitter, see section 8.4 and for a full list of the different combinations, see 3.16.1.1 below.
    5.6. To exchange the X and Y jitter offsets (i.e. have DLSS use the incoming X offset as the Y offset and vice versa), use CTRL+ALT+F10. For more information on debugging jitter, see section 8.4.

3.16.1.1 Jitter Offset Configurations

To assist in debugging issues with sub-pixel jitter, the DLSS SDK library can optionally adjust the jitter offset components using the CTRL+ALT+F9 hotkey. These are the configurations:

```
// By default, jitter offsets are used as sent from the engine
Config 0: OFF

// Combinations halving and negating both vector components
Config 1:
       JitterOffsetX *= 0.5f;
       JitterOffsetY *= 0.5f;
Config 2:
       JitterOffsetX *= 0.5f;
       JitterOffsetY *= -0.5f;
Config 3:
       JitterOffsetX *= -0.5f;
       JitterOffsetY *= 0.5f;
Config 4:
       JitterOffsetX *= -0.5f;
       JitterOffsetY *= -0.5f;
```

```
// Combinations doubling and negating both vector components
Config 5:
       JitterOffsetX *= 2.0f;
       JitterOffsetY *= 2.0f;
Config 6:
       JitterOffsetX *= 2.0f;
       JitterOffsetY *= -2.0f;
Config 7:
       JitterOffsetX *= -2.0f;
       JitterOffsetY *= 2.0f;
Config 8:
       JitterOffsetX *= -2.0f;
       JitterOffsetY *= -2.0f;

// Combinations negating one or both vector components
Config 9:
       JitterOffsetX *= 1.0f;
       JitterOffsetY *= -1.0f;
Config 10:
       JitterOffsetX *= -1.0f;
       JitterOffsetY *= 1.0f;
Config 11:
       JitterOffsetX *= -1.0f;
       JitterOffsetY *= -1.0f;

// Combinations halving and negating individual vector components
Config 12:
       JitterOffsetX *= 0.5f;
Config 13:
       JitterOffsetY *= 0.5f;
Config 14:
       JitterOffsetX *= -0.5f;
Config 15:
       JitterOffsetY *= -0.5f;

// Combinations doubling and negating individual vector components
Config 16:
       JitterOffsetX *= 2.0f;
Config 17:
       JitterOffsetY *= 2.0f;
Config 18:
       JitterOffsetX *= -2.0f;
Config 19:
       JitterOffsetY *= -2.0f;
```

3.17 NGX Logging

If the DLSS on-screen debugging information does not provide adequate detail, there are more verbose logs generated by NGX. These logs are saved to files in the path included during NGX initialization and can also be displayed to a separate console window if desired.

The latest DLSS debugging registry keys are available in the "`utils`" directory.

1. The "`ngx_log_on.reg`" file enables the NGX logging system (meaning log files are generated).

2. The "`ngx_log_off.reg`" file disables the NGX logging system (meaning no log files are generated).
3. The "`ngx_log_verbose.reg`" enables the verbose level of NGX logging.
4. The "`ngx_log_window_on.reg`" enables the display of a separate on-screen console window showing the logs in real-time.
    a. Certain fullscreen games and applications can exhibit unexpected behavior when the NGX logging window is used. If that occurs, try running the game in Windowed mode or disable the NGX logging window by running "`ngx_log_window_off.reg`".
5. The "`ngx_log_window_off.reg`" disables the separate on-screen console window.

IMPORTANT: NGX or DLSS may silently fail to load or initialize if the path provided to NGX for logging is not writeable. The developer must ensure they provide a valid path if logging is enabled. Creating a directory in "`%USERPROFILE%\AppData\Local\`" and using that for logging is a common option.

In addition, the app may also directly raise the logging level used by NGX, and have the NGX log line piped back to the app via a callback, among other logging-related settings that the app may set. These features may be used by the app by setting additional parameters in the `NVSDK_NGX_FeatureCommonInfo` struct that the app may pass in when initializing NGX. Please see section 5.2 below for more details.

For Linux, set environment variable __NGX_LOG_LEVEL=1.

3.18 Sample Code

The latest sample app is found on the DLSS Developer Zone Page and is bundled as a self-contained ZIP included with each release of the DLSS SDK:

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

Instructions to compile and build are found in `.../DLSS_Sample_App/README.md`.

The sample app is written using the NVIDIA "Donut" framework. The application code is in "`.../DLSS_Sample_App/ngx_dlss_demo`". The "`NGXWrapper.cpp`" file contains the NGX calls, which are invoked from "`DemoMain.cpp`".

3.19 Integrating DLSS using Streamline SDK

DLSS can be integrated into any application through the Streamline SDK. You will need the following:

- Streamline SDK
- DLSS Plugin for Streamline – *sl.dlss.dll* available in the Streamline SDK
- DLSS Binary – a compatible *nvngx_dlss.dll* which can be used by sl.dlss.dll

The DLSS Plugin for Streamline (*sl.dlss.dll*) serves as a wrapper for *nvngx_dlss.dll*. For more details on how to integrate DLSS into applications using the Streamline SDK, please refer to the programming guide shipped as a part of that SDK:

https://github.com/NVIDIAGameWorks/Streamline

4 Distributing DLSS in a Game

NVIDIA has encapsulated the DLSS technology to ensure developers can use the functionality with minimal build or packaging changes. Follow the steps in this section to include DLSS in game or application builds.

4.1 DLSS Release Process

NVIDIA hosted DLSS publicly for all development partners. To ensure the best quality experience for end-users, we encourage developers to thoroughly test DLSS on different DLSS modes, output resolutions and varying game content.

4.2 Distributable Libraries

The release library for DLSS is located in the "`./lib/`**`<plat>_<arch>`**`/rel/`" folder of the GitHub repository. Ensure that after game installation, the file resides in the same folder as the game executable (or DLL if you are building a plugin).

IMPORTANT: The development DLL is located in "`./lib/`**`<plat>_<arch>`**`/dev/`" and **MUST NOT be included or distributed in any public release**. This library includes on-screen notices and has an overlay designed for debugging only (see section 8.2).

NGX AI Powered Feature	NGX Feature DLL
DLSS	nvngx_dlss.dll / libnvidia-ngx-dlss.so

Note: The DLSS library can be loaded from an alternate directory path if required. The entry point, `NVSDK_NGX_D3D12_Init_ext`, accepts a parameter, `InFeatureInfo` which has a `NVSDK_NGX_PathListInfo` item which contains a list of paths to search through. For more information, see sections 5.1 and 5.2 below.

4.2.1 Removing the DLSS Library

The installer for the game or application should treat the DLSS library in the same way as other components and remove the library when uninstalling.

4.2.2 Signing the DLSS Library

The DLSS DLL is cryptographically signed and securely loaded by the NVIDIA driver (NGX Core). If the game or application DLLs are signed during the build or packaging process, the signature must be appended to the existing NVIDIA signature (do NOT strip the NVIDIA signature).

IMPORTANT: If the NVIDIA signature is missing or corrupted on the DLSS DLL, NGX Core is not able to load the library and DLSS functionality will fail.

4.2.3 Notice of inclusion of third-party code

The DLSS DLL that will be part of the game or application package includes third party code that needs to be acknowledged in the public documentation of the final product (the game or application).

Please include the full text of copyright and license blurbs that are found in section 9.5.

5 DLSS Code Integration

5.1 Adding DLSS to a Project

The NGX DLSS SDK includes four header files:

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

The header files are located in the "`./include`" folder. In the game project, include `nvsdk_ngx_helpers.h`.

NOTE: Vulkan headers follow the above naming convention with a "_vk" suffix and must be included with Vulkan applications.

During development, copy the `nvngx_dlss.dll / libnvidia-ngx-dlss.so` from the "`./lib/`**`<plat>_<arch>`**`/dev/`" directory on GitHub to the folder where the main game executable or DLL is located so the NGX runtime can properly find and load the DLL. If required by the game or the game build or packaging system, the DLSS library can be packaged in a different location to the main executable. In such case, the entry point, `NVSDK_NGX_D3D12_Init_ext`, accepts a parameter, `InFeatureInfo` which has a `NVSDK_NGX_PathListInfo` item which contains a list of paths to search through. For more information, see section 5.1.

5.1.1 Linking the NGX SDK on Windows

In addition to including the NGX header files, the project must also link against:

1. `nvsdk_ngx_`**`s`**`.lib` (if the project uses static runtime library (**/MT**) linking),

   or

2. `nvsdk_ngx_d.lib` (if the project uses dynamic runtime library (**/MD**) linking).

Both files are located in the "`./lib/<plat>_<arch>`" folder.

5.1.2 Linking the NGX SDK on Linux

In addition to including the NGX header files, the project must also link against:

`libsdk_nvngx.a`

and

`libstdc++.so`

and

`libdl.so`

5.2 Initializing NGX SDK Object

DLSS is a feature of NGX which is shipped as part of the NVIDIA display driver. The NGX SDK uses a similar set of calls for its supported APIs (Vulkan, D3D11, D3D12 and CUDA) so you can initialize NGX (and then DLSS) using code that matches or is very similar to the following sample code. Ensure that calls match the API used by the game or rendering engine in use. Any interoperability between APIs, for example, D3D11 to CUDA, must be handled by the game or application outside of the NGX SDK.

Additional note for Vulkan: the application must enable the instance and device extensions as queried by `NVSDK_NGX_VULKAN_RequiredExtensions` on the instance and device which will be used for NGX. This API is expected to contain "VK_EXT_buffer_device_address" by default, however, if application is interested in using "VK_KHR_EXT_buffer_device_address" one can replace " VK_EXT_buffer_device_address " extension string with "VK_KHR_buffer_device_address" for initialization of vulkan since NGX supports both of these extensions.

To initialize an NGX SDK instance, use one of the following methods:

```
typedef struct NVSDK_NGX_PathListInfo
{
    wchar_t **Path;
    // Path-list length
    unsigned int Length;
} NVSDK_NGX_PathListInfo;

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;

// A logging callback provided by the app to allow piping log lines back to the app.
// Please take careful note of the signature and calling convention.
// The callback must be able to be called from any thread.
// It must also be fully thread-safe and any number of threads may call into it concurrently.
```

```
// It must fully process message by the time it returns, and there is no guarantee that
// message will still be valid or allocated after it returns.
// message will be a null-terminated string and may contain multibyte characters.
#if defined(__GNUC__) || defined(__clang__)
typedef void NVSDK_CONV(*NVSDK_NGX_AppLogCallback)(const char* message,
NVSDK_NGX_Logging_Level loggingLevel, NVSDK_NGX_Feature sourceComponent);
#else
typedef void(NVSDK_CONV* NVSDK_NGX_AppLogCallback)(const char* message,
NVSDK_NGX_Logging_Level loggingLevel, NVSDK_NGX_Feature sourceComponent);
#endif

typedef struct NGSDK_NGX_LoggingInfo
{
    // Fields below were introduced in SDK version 0x0000014

    // App-provided logging callback
    NVSDK_NGX_AppLogCallback LoggingCallback;

    // The minimum logging level to use. If this is higher
    // than the logging level otherwise configured, this will override
    // that logging level. Otherwise, that logging level will be used.
    NVSDK_NGX_Logging_Level MinimumLoggingLevel;

    // Whether or not to disable writing log lines to sinks other than the app log
    //callback.
    // This may be useful if the app provides a logging callback. LoggingCallback must be
    //non-null and point
    // to a valid logging callback if this is set to true.
    bool DisableOtherLoggingSinks;

} NGSDK_NGX_LoggingInfo;

typedef struct NVSDK_NGX_FeatureCommonInfo
{
    // List of all paths in descending order of search sequence to locate a feature dll
    // in, other than the default path - application folder.
    NVSDK_NGX_PathListInfo PathListInfo;

    // Used internally by NGX
    // Introduced in SDK version 0x0000013
    NVSDK_NGX_FeatureCommonInfo_Internal* InternalData;
    // Fields below were introduced in SDK version 0x0000014

    NGSDK_NGX_LoggingInfo LoggingInfo;

} NVSDK_NGX_FeatureCommonInfo;

// NVSDK_NGX_Init
// -------------------------------------
//
// InApplicationId:
//      Unique Id provided by NVIDIA
//
// InApplicationDataPath:
//      Folder to store logs and other temporary files (write access required)
//
// InDevice: [d3d11/12 only]
//      DirectX device to use
//
// InFeatureInfo:
//      Contains information common to all features, presently only a list of all paths
```

```
//      feature dlls can be located in, other than the default path - application directory.
//
// InSDKVersion:
//       The minimum API version required to be supported by the drivers installed on the
//       user's machine. Certain SDK features require a minimum API version to be supported
//       by the user's installed drivers. The call to the SDK initialization function
//       will fail if the drivers do not support at least API version InSDKVersion. The
//       application should pass in the appropriate InSDKVersion for its required set of
//       SDK features.
//
// DESCRIPTION:
//      Initializes new SDK instance.
//

NVSDK_NGX_Result  NVSDK_CONV 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_CONV 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);



#ifdef __cplusplus
NVSDK_NGX_Result  NVSDK_CONV NVSDK_NGX_VULKAN_Init(unsigned long long InApplicationId, const
wchar_t *InApplicationDataPath, VkInstance InInstance, VkPhysicalDevice InPD, VkDevice
InDevice, const NVSDK_NGX_FeatureCommonInfo *InFeatureInfo = nullptr, NVSDK_NGX_Version
InSDKVersion = NVSDK_NGX_Version_API);
#else
NVSDK_NGX_Result  NVSDK_CONV NVSDK_NGX_VULKAN_Init(unsigned long long InApplicationId, const
wchar_t *InApplicationDataPath, VkInstance InInstance, VkPhysicalDevice InPD, VkDevice
InDevice, const NVSDK_NGX_FeatureCommonInfo *InFeatureInfo, NVSDK_NGX_Version InSDKVersion);
#endif

// NGX return-code conversion-to-string utility only as a debug/logging aide - not for
official use.
const wchar_t* NVSDK_CONV GetNGXResultAsString(NVSDK_NGX_Result InNGXResult);

typedef enum NVSDK_NGX_EngineType
{
    CUSTOM = 0,
    UNREAL,
    UNITY,
    OMNIVERSE,
    NUM_GENERIC_ENGINES
} NVSDK_NGX_EngineType;


////////////////////////////////////////////////////////////////////////////////////////////
/////
// NVSDK_NGX_Init_with_ProjectID
// -------------------------------------
//
// InProjectId:
//      Unique Id provided by the rendering engine used
//
// InEngineType:
//      Rendering engine used by the application / plugin.
//      Use NVSDK_NGX_ENGINE_TYPE_CUSTOM if the specific engine type is not supported
explicitly
```

```
//
// InEngineVersion:
//      Version number of the rendering engine used by the application / plugin.
//
// InApplicationDataPath:
//      Folder to store logs and other temporary files (write access required),
//      Normally this would be a location in Documents or ProgramData.
//
// InDevice: [d3d11/12 only]
//      DirectX device to use
//
// InFeatureInfo:
//      Contains information common to all features, presently only a list of all paths
//      feature dlls can be located in, other than the default path - application directory.
//
// DESCRIPTION:
//      Initializes new SDK instance.
//
NVSDK_NGX_Result  NVSDK_CONV NVSDK_NGX_D3D11_Init_with_ProjectID(const char *InProjectId,
NVSDK_NGX_EngineType InEngineType, const char *InEngineVersion, const wchar_t
*InApplicationDataPath, ID3D11Device *InDevice, const NVSDK_NGX_FeatureCommonInfo
*InFeatureInfo = nullptr, NVSDK_NGX_Version InSDKVersion = NVSDK_NGX_Version_API);

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

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

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

Certain SDK features may require a certain minimum driver version. The required features that the driver must support are gated by the `InSDKVersion` value passed into the SDK initialization functions for each API. Thus, if an application requires features only available in SDK API version 0x0000014 or higher, then it should pass in `0x0000014` for `InSDKVersion`. If it does not require said features, it should pass in a lower value (such as `0x0000013`, which also generally serves as a good baseline API version for most applications, since it supports most SDK features, and is also widely supported by the drivers installed on most user machines; please see the SDK headers to determine if your application uses SDK features that require a higher API version). The minimum API version required for the various SDK features is documented in the NGX SDK headers; please see the definition of `struct NVSDK_NGX_FeatureCommonInfo` as an example.

5.2.1 Project ID

Project Id refers to a unique ID(InProjectId) that is specific to certain 3rd party engines like Unreal or Omniverse. The DLSS integration for those engines should already take care of passing the value to DLSS. Make sure that the Project ID is set in the engine's editor.

Project ID is assumed to be GUID-like, for instance:

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

For CUSTOM engine type the driver validates that passed project ID satisfies GUID-likeness requirements. If you get error from NVSDK_NGX_*_Init_with_ProjectID(), you can see what exact problem was by inspecting NGX log.

Use NVSDK_NGX_*_Init_with_ProjectID() function along with a Project ID, Engine Type and Engine Version, as described in their respective sections.

***Note**: If your NVIDIA Developer Technologies contact provides you with a NVIDIA Application ID (`InApplicationId`), please use the NVSDK_NGX_*_Init() functions to initialize NGX and pass in the given application ID instead of Project ID.*

5.2.2 Engine Type

It refers to the rendering engine (InEngineType) used by the application.

5.2.3 Engine Version

Engine version (InEngineVersion) should be the same version that is reported by the core engine.

5.2.4 Thread Safety

The NGX API is **not** thread safe. The client application must ensure that thread safety is enforced as needed. Making evaluations, calls or invocations on the same NGX Feature and associated NGX parameter object from multiple threads can result in unpredictable behavior.

5.2.5 Contexts and Command Lists

For DLSS titles that use DirectX 11, the NGX API preserves the state of the immediate D3D11 context. This is **not** the case with D3D12 command lists or Vulkan command buffers. For applications using DirectX 12 or Vulkan, the client application must manage the command list and command buffer state as needed.

5.2.6 Verifying Availability of NGX Features and Allocating Parameter Maps

Successful initialization of the NGX SDK instance indicates that the target system is capable of running NGX features. However, each feature can have additional dependencies, for example, a minimum driver version. It is therefore good practice to check if the specific feature (i.e. DLSS) is available. For this purpose, NGX provides an NVSDK_NGX_Parameter interface which can be used to query read-only parameters provided by the NGX runtime and obtained using the following API:

```
////////////////////////////////////////////////////////////////////////////////////////////
//////
// NVSDK_NGX_AllocateParameters
// ------------------------------------------------------
//
// OutParameters:
//      Parameters interface used to set any parameter needed by the SDK
//
// DESCRIPTION:
//      This interface allows allocating a simple parameter setup using named fields, whose
//      lifetime the app must manage.
//      For example one can set width by calling Set(NVSDK_NGX_Parameter_Denoiser_Width,100)
//      or provide CUDA buffer pointer by calling
//      Set(NVSDK_NGX_Parameter_Denoiser_Color,cudaBuffer)
//      For more details please see sample code.
//      Parameter maps output by NVSDK_NGX_AllocateParameters must NOT be freed using
//      the free/delete operator; to free a parameter map
//      output by NVSDK_NGX_AllocateParameters, NVSDK_NGX_DestroyParameters should be used.
//      Unlike with NVSDK_NGX_GetParameters, parameter maps allocated with
//      NVSDK_NGX_AllocateParameters
//      must be destroyed by the app using NVSDK_NGX_DestroyParameters.
//      Also unlike with NVSDK_NGX_GetParameters, parameter maps output 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,
//      NVSDK_NGX_GetCapabilityParameters
//      should be used.
//      This function may return NVSDK_NGX_Result_FAIL_OutOfDate if an older driver, which
//      does not support this API call is being used. In such a case, NVSDK_NGX_GetParameters
//      may be used as a fallback.
//      This function may only be called after a successful call into NVSDK_NGX_Init.
//
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV
NVSDK_NGX_D3D11_AllocateParameters(NVSDK_NGX_Parameter** OutParameters);
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV
NVSDK_NGX_D3D12_AllocateParameters(NVSDK_NGX_Parameter** OutParameters);
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV
NVSDK_NGX_VULKAN_AllocateParameters(NVSDK_NGX_Parameter** OutParameters);

////////////////////////////////////////////////////////////////////////////////////////////
//////
// NVSDK_NGX_GetCapabilityParameters
// ------------------------------------------------------
//
// OutParameters:
//      The parameters interface populated with NGX and feature capabilities
//
// DESCRIPTION:
//      This interface allows the app to create a new parameter map
//      pre-populated with NGX capabilities and available features.
//      The output parameter map can also be used for any purpose
//      parameter maps output by NVSDK_NGX_AllocateParameters can be used for
//      but it is not recommended to use NVSDK_NGX_GetCapabilityParameters
//      unless querying NGX capabilities and available features
//      due to the overhead associated with pre-populating the parameter map.
//      Parameter maps output by NVSDK_NGX_GetCapabilityParameters must NOT be freed using
//      the free/delete operator; to free a parameter map
//      output by NVSDK_NGX_GetCapabilityParameters, NVSDK_NGX_DestroyParameters should be
//      used.
//      Unlike with NVSDK_NGX_GetParameters, parameter maps allocated with
//      NVSDK_NGX_GetCapabilityParameters
```

```
//      must be destroyed by the app using NVSDK_NGX_DestroyParameters.
//      This function may return NVSDK_NGX_Result_FAIL_OutOfDate if an older driver, which
//      does not support this API call is being used. This function may only be called
//      after a successful call into NVSDK_NGX_Init.
//      If NVSDK_NGX_GetCapabilityParameters fails with NVSDK_NGX_Result_FAIL_OutOfDate,
//      NVSDK_NGX_GetParameters may be used as a fallback, to get a parameter map pre-
//      populated with NGX capabilities and available features.
//
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV
NVSDK_NGX_D3D11_GetCapabilityParameters(NVSDK_NGX_Parameter** OutParameters);
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV
NVSDK_NGX_D3D12_GetCapabilityParameters(NVSDK_NGX_Parameter** OutParameters);
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV
NVSDK_NGX_VULKAN_GetCapabilityParameters(NVSDK_NGX_Parameter** OutParameters);

////////////////////////////////////////////////////////////////////////////////////////
//////
// NVSDK_NGX_DestroyParameters
// -------------------------------------------------------
//
// InParameters:
//      The parameters interface to be destroyed
//
// DESCRIPTION:
//      This interface allows the app to destroy the parameter map passed in. 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 return NVSDK_NGX_Result_FAIL_OutOfDate if an older driver, which
//      does not support this API call is being used. This function may only be called
//      after a successful call into NVSDK_NGX_Init.
//
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV
NVSDK_NGX_D3D11_DestroyParameters(NVSDK_NGX_Parameter* InParameters);
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV
NVSDK_NGX_D3D12_DestroyParameters(NVSDK_NGX_Parameter* InParameters);
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV
NVSDK_NGX_VULKAN_DestroyParameters(NVSDK_NGX_Parameter* InParameters);

////////////////////////////////////////////////////////////////////////////////////////
//////
// NVSDK_NGX_GetParameters
// -------------------------------------------------------
//
// OutParameters:
//      Parameters interface used to set any parameter needed by the SDK
//
// DESCRIPTION:
//      This interface allows simple parameter setup using named fields.
//      For example one can set width by calling Set(NVSDK_NGX_Parameter_Denoiser_Width,100)
//      or provide CUDA buffer pointer by calling
//      Set(NVSDK_NGX_Parameter_Denoiser_Color,cudaBuffer)
//      For more details please see sample code. Please note that allocated memory
//      will be freed by NGX so free/delete operator should NOT be called.
//      Parameter maps output by NVSDK_NGX_GetParameters are also pre-populated
//      with NGX capabilities and available features.
//      Unlike with NVSDK_NGX_AllocateParameters, parameter maps output by
//      NVSDK_NGX_GetParameters
//      have their lifetimes managed by NGX, and must not
```

```
//      be destroyed by the app using NVSDK_NGX_DestroyParameters.
//      NVSDK_NGX_GetParameters is deprecated and apps should move to using
//      NVSDK_NGX_AllocateParameters and NVSDK_NGX_GetCapabilityParameters when possible.
//      Nevertheless, due to the possibility that the user will be using an older driver
//      version, NVSDK_NGX_GetParameters may still be used as a fallback if
//      NVSDK_NGX_AllocateParameters
//      or NVSDK_NGX_GetCapabilityParameters return NVSDK_NGX_Result_FAIL_OutOfDate.
//
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV NVSDK_NGX_D3D11_GetParameters(NVSDK_NGX_Parameter
**OutParameters);
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV NVSDK_NGX_D3D12_GetParameters(NVSDK_NGX_Parameter
**OutParameters);
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV NVSDK_NGX_VULKAN_GetParameters(NVSDK_NGX_Parameter
**OutParameters);
```

The app should allocate a new parameter map pre-populated with the above-mentioned read-only parameters provided by NGX, using the NVSDK_NGX_GetCapabilityParameters interface. Note that if the user is using an older driver, NVSDK_NGX_GetCapabilityParameters may fail with NVSDK_NGX_Result_FAIL_OutOfDate. In that case, the app may fall back on using NVSDK_NGX_GetParameters to get a parameter map pre-populated with the above-mentioned read-only parameters. Note however, that NVSDK_NGX_GetParameters is deprecated, and its use as a fallback by apps is intended as a stopgap, until a new enough driver version is widely installed.

Sometimes NGX feature is denied for certain apps. It can be also queried using NVSDK_NGX_GetParameters with enum: NVSDK_NGX_Parameter_SuperSampling_ FeatureInitResult.

NOTE: NVSDK_NGX_Parameter objects output by NVSDK_NGX_GetParameters are automatically released by NGX during the shutdown call. However, parameter objects output by NVSDK_NGX_GetCapabilityParameters must be destroyed by the app using NVSDK_NGX_DestroyParameters.

For example, to check if DLSS is available on the user's system when running in Vulkan, use the following:

```
int DLSS_Supported = 0;
int needsUpdatedDriver = 0;
unsigned int minDriverVersionMajor = 0;
unsigned int minDriverVersionMinor = 0;

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

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

NVSDK_NGX_Result ResultUpdatedDriver =
 Params->Get(NVSDK_NGX_Parameter_SuperSampling_NeedsUpdatedDriver, &needsUpdatedDriver);

NVSDK_NGX_Result ResultMinDriverVersionMajor =
```

```
 Params->Get(NVSDK_NGX_Parameter_SuperSampling_MinDriverVersionMajor, &minDriverVersionMajor);
NVSDK_NGX_Result ResultMinDriverVersionMinor =
 Params->Get(NVSDK_NGX_Parameter_SuperSampling_MinDriverVersionMinor, &minDriverVersionMinor);

if (NVSDK_NGX_SUCCEED(ResultUpdatedDriver))
{
    if (needsUpdatedDriver)
    {
        // NVIDIA DLSS cannot be loaded due to outdated driver.
        if (NVSDK_NGX_SUCCEED(ResultMinDriverVersionMajor) &&
            NVSDK_NGX_SUCCEED(ResultMinDriverVersionMinor))
        {
            // Min Driver Version required: minDriverVersionMajor.minDriverVersionMinor
        }
        // Fallback to default AA solution (TAA etc)
    }
    else
    {
        // driver update is not required – so application is not expected to
        // query minDriverVersion in this case
    }
}

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

if (NVSDK_NGX_FAILED(ResultDlssSupported) || !DLSS_Supported )
{
 // NVIDIA DLSS not available on this hardward/platform.

 // Fallback to default AA solution (TAA etc)
}

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

if (NVSDK_NGX_FAILED(ResultDlssSupported) || !DLSS_Supported )
{
 // NVIDIA DLSS is denied on for this application.

 // Fallback to default AA solution (TAA etc)
}

if(bShouldDestroyCapabilityParams)
{
  NVSDK_NGX_VULKAN_DestroyParameters(Params);
  Params = nullptr;
}
```

5.2.7 Overriding Feature Denial

Application developers that are using DLSS should have the ability to override whether or not a feature is denied, in order to help them test what would happen in case we were to ever disable the feature for their application.

Overriding the feature denial mechanism will have to be supported via windows regkeys (similar to how most of the other overrides are supported in NGX). All regkeys for this override must be written to HKLM\SOFTWARE\NVIDIA Corporation\Global\NGXCore.

The various regkeys required for developing this override are:

OverrideFeatureDeny

a. Type: REG_DWORD
b. Values: whether or not to enable overriding - 0 = disable overriding, non-zero: enable overriding
c. Default behavior: disable overriding

OverrideFeatureDeny_CMSID

a. Type: REG_SZ
b. Values: CMS ID for the specific application being overridden; if app doesn't have a unique CMS ID or if app dev doesn't know the CMS ID for their app, do not create this regkey
c. Default behavior: apply override behavior to all apps

OverrideFeatureDeny_Feature

a. Type: REG_SZ
b. Values: feature name for the specific feature being overridden; do not create regkey if all features are to be overridden
c. Default behavior: apply override behavior to all features

OverrideFeatureDeny_Deny

a. Type: REG_DWORD
b. Values: whether or not to deny the feature - 0 = allow, 1 = deny
c. Default behavior: allow

In Linux, overriding is not yet supported.

5.2.8 Obtaining the Optimal Settings for DLSS

Once DLSS is confirmed as available on the user's system, obtain the optimal render target size for DLSS. This size is based on various factors including the type of GPU and whether RTX (raytracing) is on or off.

To obtain the optimal resolution settings for DLSS issue the query below. The developer should confirm the returned `DLSSMode` for each combination of:

- Target display resolution size
- `PerfQualityValue`

NOTE: The `PerfQualityValue` is currently defined in the NGX DLSS header as having five possible values. All five `PerfQualityValue` values should be checked and if they are enabled, they should be selectable in the Game UI (and hidden if disabled). For more information on these values, see section 3.2.1.

5.2.8.1 Optimal resolution settings query:

```
//   Assuming the game is looping through all combinations for the following:
//
```

```
//   Resolution (TargetWidth, TargetHeight)
//   PerfQualityValue
//     [0 MaxPerformance, 1 Balance, 2 MaxQuality, 3 UltraPerformance, 4 UltraQuality]

unsigned int RenderWidth, RenderHeight;
float Sharpness = 0.0f; // Sharpening is deprecated, see section 3.11

DLSSMode = NGX_DLSS_GET_OPTIMAL_SETTINGS(
    Params,
    TargetWidth, TargetHeight,
    PerfQualityValue,
    &RecommendedOptimalRenderWidth, &RecommendedOptimalRenderHeight,
    &DynamicMaximumRenderSizeWidth, &DynamicMaximumRenderSizeHeight,
    &DynamicMinimumRenderSizeWidth, &DynamicMinimumRenderSizeHeight,
    &Sharpness);

if (RecommendedOptimalRenderWidth == 0 || RecommendedOptimalRenderHeight == 0)
   (
{
    // This PerfQuality mode has not been made available yet.
     // Please request another PerfQuality mode.

}
else
{
    // Use DLSS for this combination
    // - Create feature with RecommendedOptimalRenderWidth, RecommendedOptimalRenderHeight
    // - Render to (RenderWidth, RenderHeight) between Min and Max inclusive
    // - Call DLSS to upscale to (TargetWidth, TargetHeight)
}
```

5.3 Feature Creation

When all necessary parameters are set, create the DLSS feature. The following code can be used to create the feature:

```
// For a reference – taken from nvngx_ngx_defs.h
typedef enum NVSDK_NGX_DLSS_Feature_Flags
{
    NVSDK_NGX_DLSS_Feature_Flags_IsInvalid      = 1 << 31,

    NVSDK_NGX_DLSS_Feature_Flags_None           = 0,
    NVSDK_NGX_DLSS_Feature_Flags_IsHDR          = 1 << 0,
    NVSDK_NGX_DLSS_Feature_Flags_MVLowRes       = 1 << 1,
    NVSDK_NGX_DLSS_Feature_Flags_MVJittered     = 1 << 2,
    NVSDK_NGX_DLSS_Feature_Flags_DepthInverted  = 1 << 3,
    NVSDK_NGX_DLSS_Feature_Flags_Reserved_0     = 1 << 4,
    NVSDK_NGX_DLSS_Feature_Flags_AutoExposure   = 1 << 6,
} NVSDK_NGX_DLSS_Feature_Flags;

typedef struct NVSDK_NGX_Feature_Create_Params
{
    unsigned int InWidth;           // Obtained via NGX_DLSS_GET_OPTIMAL_SETTINGS
    unsigned int InHeight;          // Obtained via NGX_DLSS_GET_OPTIMAL_SETTINGS
    unsigned int InTargetWidth;     // Target width (final resolution)
    unsigned int InTargetHeight;    // Target height (final resolution)
    NVSDK_NGX_PerfQuality_Value InPerfQualityValue;
      // User selection in UI: MaxPerf, Balanced, MaxQuality, UltraPerformance, UltraQuality
```

```
} NVSDK_NGX_Feature_Create_Params;

typedef struct NVSDK_NGX_DLSS_Create_Params
{
    NVSDK_NGX_Feature_Create_Params Feature;
    /*** OPTIONAL ***/
    int     InFeatureCreateFlags;    // Combination of NVSDK_NGX_DLSS_Feature_Flags
    bool    InEnableOutputSubrects;// Whether to use subrects in the output resource
} NVSDK_NGX_DLSS_Create_Params;

NGX_D3D11_CREATE_DLSS_EXT(
CommandCtx,           // Command context
&FeatureHandle[Node], // Handle for the new feature
Params,               // Parameters obtained with NVSDK_NGX_AllocateParameters
&DlssCreateParams     // Parameters in NVSDK_NGX_DLSS_Create_Params struct
);

NGX_D3D12_CREATE_DLSS_EXT(
CommandList[Node],    // Command list for current GPU node
CreationNodeMask,     // Multi GPU only (default 1)
VisibilityNodeMask    // Multi GPU only (default 1)
&FeatureHandle[Node], // Handle for the new feature
Params,               // Parameters obtained with NVSDK_NGX_AllocateParameters
&DlssCreateParams     // Parameters in NVSDK_NGX_DLSS_Create_Params struct
);

NGX_VULKAN_CREATE_DLSS_EXT(
VkCommandBuffer,      // Vulkan Command Buffer
CreationNodeMask,     // Multi GPU only (default 1)
VisibilityNodeMask    // Multi GPU only (default 1)
&FeatureHandle[Node], // Handle for the new feature
Params,               // Parameters obtained with NVSDK_NGX_AllocateParameters
&DlssCreateParams     // Parameters in NVSDK_NGX_DLSS_Create_Params struct
);
```

NOTE: If NVSDK_NGX_AllocateParameters may fail with NVSDK_NGX_Result_FAIL_OutOfDate, if the user is using an older driver version. In that case, the app should fall back on using NVSDK_NGX_GetParameters to get a parameter map to use in the above functions. Note however, that NVSDK_NGX_GetParameters is soon to be deprecated, and its use as a fallback by apps is intended as a stopgap, until a new enough driver version is widely installed.

5.4 Feature Evaluation

Features are evaluated by executing an inference call on a specific algorithm and deep learning model. For DLSS the evaluation calls are:

```
// D3D12      - NVSDK_NGX_D3D12_Feature_Eval_Params
// VULKAN     - NVSDK_NGX_VK_Feature_Eval_Params
typedef struct NVSDK_NGX_D3D11_Feature_Eval_Params
{
    ID3D11resource* pInColor;       // Color buffer
    ID3D11resource* pInOutput;      // Output buffer
    /*** OPTIONAL for DLSS ***/
    float       InSharpness;        // Deprecated, see section 3.11
} NVSDK_NGX_D3D11_Feature_Eval_Params
```

```
typedef enum NVSDK_NGX_ToneMapperType
{
    NVSDK_NGX_TONEMAPPER_STRING = 0,
    NVSDK_NGX_TONEMAPPER_REINHARD,
    NVSDK_NGX_TONEMAPPER_ONEOVERLUMA,
    NVSDK_NGX_TONEMAPPER_ACES,
    NVSDK_NGX_TONEMAPPERTYPE_NUM
} NVSDK_NGX_ToneMapperType;

typedef enum NVSDK_NGX_GBufferType
{
    NVSDK_NGX_GBUFFER_ALBEDO = 0,
    NVSDK_NGX_GBUFFER_ROUGHNESS,
    NVSDK_NGX_GBUFFER_METALLIC,
    NVSDK_NGX_GBUFFER_SPECULAR,
    NVSDK_NGX_GBUFFER_SUBSURFACE,
    NVSDK_NGX_GBUFFER_NORMALS,
    // unique identifier for drawn object or how the object is drawn
    NVSDK_NGX_GBUFFER_SHADINGMODELID,
    NVSDK_NGX_GBUFFER_MATERIALID,  // unique identifier for material
    NVSDK_NGX_GBUFFERTYPE_NUM = 16
} NVSDK_NGX_GBufferType;

// D3D12       - D3D12_GBuffer
// VULKAN      - VK_GBuffer
typedef struct NVSDK_NGX_D3D11_GBuffer
{
    ID3D11Resource* pInAttrib[GBUFFERTYPE_NUM];
} NVSDK_NGX_D3D11_GBuffer;

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

// D3D12       - NVSDK_NGX_D3D12_DLSS_Eval_Params
// VULKAN      - NVSDK_NGX_VK_DLSS_Eval_Params
typedef struct NVSDK_NGX_D3D11_DLSS_Eval_Params
{
    NVSDK_NGX_D3D11_Feature_Eval_Params Feature;
    ID3D11Resource*               pInDepth;               // Depth buffer
    ID3D11Resource*               pInMotionVectors;       // MVs buffer
    /* Jitter Offsets in pixel space */
    float                         InJitterOffsetX;
    float                         InJitterOffsetY;
    NVSDK_NGX_Dimensions          InRenderSubrectDimensions;
    /*** OPTIONAL ***/
    int                           InReset;// Resets history buffer for scene transitions.
    float                         InMVScaleX;             // MotionVector Scale X
    float                         InMVScaleY;             // MotionVector Scale Y
    ID3D11Resource*               pInTransparencyMask;
    ID3D11Resource*               pInExposureTexture;     // Exposure texture 1x1
    NVSDK_NGX_Coordinates         InColorSubrectBase;// Offsets into input/output buffers
    NVSDK_NGX_Coordinates         InDepthSubrectBase;
    NVSDK_NGX_Coordinates         InMVSubrectBase;
    NVSDK_NGX_Coordinates         InTranslucencySubrectBase;
    NVSDK_NGX_Coordinates         InOutputSubrectBase;
    float                         InPreExposure;// Pre-exposure if applicable
    /*** OPTIONAL - only for research purposes ***/
    /* per pixel mask identifying alpha-blended objects like particles etc. */
```

```
    NVSDK_NGX_D3D11_GBuffer       GBufferSurface;
    NVSDK_NGX_ToneMapperType      InToneMapperType;
    ID3D11Resource*               pInMotionVectors3D;
    ID3D11Resource*               pInIsParticleMask;
    ID3D11Resource*               pInAnimatedTextureMask;
    ID3D11Resource*               pInDepthHighRes;
    ID3D11Resource*               pInPositionViewSpace;
    float                         InFrameTimeDeltaInMsec;
    ID3D11Resource*               pInRayTracingHitDistance;
    ID3D11Resource*               pInMotionVectorsReflections;
} NVSDK_NGX_D3D11_DLSS_Eval_Params;

NGX_D3D11_EVALUATE_DLSS_EXT(
Context,             // Command context
FeatureHandle,       // Handle for the new feature
Params,              // Parameters obtained with NVSDK_NGX_AllocateParameters
D3D11DlssEvalParams // Parameters in NVSDK_NGX_D3D11_DLSS_Eval_Params struct
);

NGX_D3D12_EVALUATE_DLSS_EXT(
CommandList[Node],   // Command list for GPU node
FeatureHandle[Node], // Handle for the new feature
Params,              // Parameters obtained with NVSDK_NGX_AllocateParameters
D3D12DlssEvalParams // Parameters in NVSDK_NGX_D3D12_DLSS_Eval_Params struct
);

NGX_VULKAN_EVALUATE_DLSS_EXT(
VkCommandBuffer,      // Vulkan Command Buffer
&FeatureHandle[Node], // Handle for the feature
Params,               // Parameters obtained with NVSDK_NGX_AllocateParameters
VkDlssEvalParams     // Parameters in NVSDK_NGX_VK_DLSS_Eval_Params struct
);
```

IMPORTANT: NGX modifies the Vulkan and D3D12 command list states. The calling process must save and restore its own Vulkan or D3D12 state before and after making the NGX evaluate feature calls.

NOTE: If NVSDK_NGX_AllocateParameters may fail with NVSDK_NGX_Result_FAIL_OutOfDate, if the user is using an older driver version. In that case, the app should fall back on using NVSDK_NGX_GetParameters to get a parameter map to use in the above functions. Note however, that NVSDK_NGX_GetParameters is soon to be deprecated, and its use as a fallback by apps is intended as a stopgap, until a new enough driver version is widely installed.

5.4.1 Vulkan Resource Wrapper

A resource wrapper is used to pass metadata which is otherwise not available. This wrapper structure must be used as an argument instead of directly passing in Vulkan resources and can be used for VkImageView and VkBuffer resources. At present, only VkImageView resources are used as parameters to NGX. The following is the wrapper struct, and creation function signature:

```
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;
```

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

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

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; // True if the resource is available for read and write access.
                    // For VkImage: VkImageUsageFlags for the associated VkImage includes
                    // VK_IMAGE_USAGE_STORAGE_BIT
} NVSDK_NGX_Resource_VK;

static NVSDK_NGX_Resource_VK NVSDK_NGX_Create_ImageView_Resource_VK(VkImageView imageView,
VkImage image, VkImageSubresourceRange subresourceRange, VkFormat format, unsigned int width,
unsigned int height, bool readWrite);
```

5.5 Feature Disposal

When a feature is no longer needed, it should be released by calling the following method:

```
// NVSDK_NGX_Release
// -------------------------------------
//
// InHandle:
//      Handle to feature to be released
//
// DESCRIPTION:
//      Releases feature with a given handle.
//      Handles are not reference counted so
//      after this call it is invalid to use provided handle.
//

NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV NVSDK_NGX_D3D11_ReleaseFeature(NVSDK_NGX_Handle
*InHandle);

NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV NVSDK_NGX_D3D12_ReleaseFeature(NVSDK_NGX_Handle
*InHandle);

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

Once released, the feature handle cannot be used any longer.

A feature handle should only be released once the command lists (Direct3D) or command buffers (Vulkan) that were used in Evaluate_XXX() calls are no longer in flights as that command list could still reference internal states and resources associated with the feature.

5.6 Shutdown

To release the NGX SDK instance and all resources allocated with it, use the following method:

```
// NVSDK_NGX_Shutdown
// -------------------------------------
//
// DESCRIPTION:
//      Shuts down the current SDK instance and releases all resources.
//

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

The Shutdown() function should only be called once when all associated features have been released. That also means no work associated with those features is still in flight.

5.7 Init and Shutdown More than Once

Starting from SDK version 0x15 we support multiple devices. This means you can call Init() more than once – once per device. Legacy Shutdown() call does not have device pointer – so it will shut down NGX on ALL devices. To support shutting down NGX on one device, but not on others - new API called Shutdown1(Device) is introduced.

6 Resource Management

DLSS requires certain internal resources to be created. If a game requires full control over the NGX resource lifecycle and VRAM allocation, it is possible to register specific callbacks for resource creation and destruction. As well, parameter maps allocated using NVSDK_NGX_AllocateParameters or NVSDK_NGX_GetCapabilityParameters must be destroyed by the app using NVSDK_NGX_DestroyParameters. This can be achieved by doing the following:

6.1 D3D11 Specific

Registering callbacks:

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

```
    }
}

Params->Set(NVSDK_NGX_Parameter_BufferAllocCallback, NGXBufferAllocCallback);
```

Destroying a parameter map:

```
NVSDK_NGX_D3D11_DestroyParameters(Params); // Note: Params must
// have been allocated using NVSDK_NGX_D3D11_AllocateParameters
// or NVSDK_NGX_D3D11_GetCapabilityParameters, NOT the soon to be deprecated
// NVSDK_NGX_D3D12_GetParameters. Parameter maps allocated
// using NVSDK_NGX_D3D11_GetParameters will have their memory managed by NGX.
```

6.2 D3D12 Specific

Registering callbacks:

```
void NGXResourceAllocCallback(D3D12_RESOURCE_DESC *InDesc, int InState,
CD3DX12_HEAP_PROPERTIES *InHeap, ID3D12Resource **OutResource)
{
    *OutResource = nullptr;
    // Or obtain matching resource from the cache etc
    HRESULT hr = MyDevice->CreateCommittedResource(InHeap, D3D12_HEAP_FLAG_NONE,
        InDesc, (D3D12_RESOURCE_STATES)InState,
        nullptr, IID_PPV_ARGS(OutResource));
    if (hr != S_OK)
    {
        // Handle error
    }
}

Params->Set(NVSDK_NGX_Parameter_ResourceAllocCallback, NGXResourceAllocCallback);
```

Destroying a parameter map:

```
NVSDK_NGX_D3D12_DestroyParameters(Params); // Note: Params must
// have been allocated using NVSDK_NGX_D3D12_AllocateParameters
// or NVSDK_NGX_D3D12_GetCapabilityParameters, NOT the soon to be deprecated
// NVSDK_NGX_D3D12_GetParameters. Parameter maps allocated
// using NVSDK_NGX_D3D12_GetParameters will have their memory managed by NGX.
```

6.3 Vulkan Specific

Destroying a parameter map:

```
NVSDK_NGX_VULKAN_DestroyParameters(Params); // Note: Params must
// have been allocated using NVSDK_NGX_VULKAN_AllocateParameters
// or NVSDK_NGX_VULKAN_GetCapabilityParameters, NOT the soon to be deprecated
// NVSDK_NGX_VULKAN_GetParameters. Parameter maps allocated
// using NVSDK_NGX_VULKAN_GetParameters will have their memory managed by NGX.
```

Additional note for Vulkan: the application must enable the instance and device extensions as queried by `NVSDK_NGX_VULKAN_RequiredExtensions` on the instance and device which will be used for NGX. This API is expected to contain “VK_EXT_buffer_device_address” by default, however, if

application is interested in using “VK_KHR_EXT_buffer_device_address”. They can replace " VK_EXT_buffer_device_address “ extension string with “VK_KHR_buffer_device_address” for initialization of vulkan since NGX supports both of these extensions.

6.4 Common

```
void NGXResourceReleaseCallback(IUnknown *InResource)
{
    // Delay release as needed or manage based on use case
    SAFE_RELEASE(InResource);
}

Params->Set(NVSDK_NGX_Parameter_ResourceReleaseCallback,
    NGXResourceReleaseCallback);
```

NOTE: NGX does not hold a reference to any DirectX resource passed to it from the client side.

7 Multi GPU Support

7.1 Linked Mode

When creating DLSS feature in linked mode, one must specify Node on which DLSS feature is created. For that use CreationNodeMask. VisibilityNodeMask used during feature creation refers to resources created by DLSS feature internally.

By default, DLSS feature will be evaluated on the GPU where it has been created. It is possible to evaluate DLSS feature on any node which was included in the VisibilityNodeMask. To evaluate DLSS feature on another node, use NVSDK_NGX_EParameter_EvaluationNode parameter.

The code example below, shows how to use DLSS in Linked mode

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

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

// evaluate on default node
Status = NGX_D3D12_EVALUATE_DLSS(FeatureHandle[Node], Params,
  MyCommandList[Node], Color, MV, Output, PrevOutput, Depth));
if (NVSDK_NGX_FAILED(Status)) { // Handle error };

// evaluate on non-default node
if (Node + 1 < NodeCount)
{
  Params->Set(NVSDK_NGX_EParameter_EvaluationNode, Node + 1);
  Status = NGX_D3D12_EVALUATE_DLSS(FeatureHandle[Node], Params,
    MyCommandList[Node + 1], Color, MV, Output, PrevOutput, Depth));
```

```
  if (NVSDK_NGX_FAILED(Status)) { // Handle error };
}
```

NOTE: When using D3D11 basic multi-GPU/SLI (Alternate Frame Rendering - AFR), multi-GPU support is provided directly by the NVIDIA driver and there are no additional integration requirements.

7.2 Unlinked mode

In the unlinked mode each GPU is independent, so one can call NGX_Init() on each GPU independently of others. It's important to note here that we have several ways of shutting NGX down.

- NGX_Shutdown() call will shutdown NGX on ALL GPUs;
- NGX_Shutdown1(Device *) API will shutdown NGX just on single device that was passed as parameter;
- NGX_Shutdown1(nullptr) call works identically to NGX_Shutdown() call;

Another API relevant to this case is NGX_VULKAN_CreateFeature1(). This API is specifically introduced to create feature in unlinked mode for VULKAN. The reason that VULKAN is different here is that VULKAN does not allow inferring Device pointer from command buffer pointer. D3D11 and D3D12 do allow that so for them NGX_CreateFeature() will work fine in unlinked mode.

Note: this is supported only on 470+ drivers and only if you explicitly specify NGX API Version >= 0x15 when you call NGX_Init()

8 Troubleshooting

8.1 Common Issues Causing Visual Artifacts

1. Incorrect motion vectors: make sure you can always visualize and validate motion vectors (see the DLSS debug overlay section below).
    a. Make sure the motion vectors are pointing in a direction indicating how to get the pixel in screen space from the current frame to its position in the previous frame in screen space. So, if the game is running at 1080p and the object has moved from the left edge of the screen to the right edge, then the x value of the motion vector for that pixel on the right edge will be -1080.0.
    b. Make sure the motion vectors are expressed in 16-bit or 32-bit floating point values. Using integer values will not take sub-pixel values into account.
    c. Make sure motion vectors are in screen space and take dynamic object movements into account.
    d. Make sure all objects visible on screen have motion vectors calculated correctly. Common places to miss out motion vectors include animated foliage, the sky.

2. Wrong or out of sync jitter pattern: make sure you are using the sequence provided in section 3.7. The jitter pattern must match regardless of frame time etc.

3. Disabling TAA often changes the way the engine renders (jitter, depth, motion etc.) which can consequently break DLSS. When integrating DLSS, the TAA rendering pass should be replaced with DLSS but **everything else** in the rendering pipeline that is activated when TAA is turned on, must still be executed as if TAA is still active.

    a. Ensure that all input to DLSS is properly jittered. This includes secondary passes that might otherwise have not been jittered (e.g. object highlighting).

4. Incorrect exposure / or pre-exposure (see section 3.9): make sure you always pass in the correct exposure texture or pre-exposure value to avoid:

    a. Ghosting of moving objects.

    b. Blurriness, banding, or pixilation of the final frame or it being too dark or too bright.

    c. Aliasing especially of moving objects.

    d. A pronounced lag in the scene brightness changing when going from a dark scene to a bright scene (or vice versa).

5. In low VRAM scenarios, the game may crash if DLSS is invoked while required resources (Color, Motion Vectors etc.) are evicted from the GPU. If this happens please make sure all resources are made resident before using DLSS.

6. Use the on-screen debug text (see section 3.8) to verify the resolution of the buffers that are passed into the DLSS module.

7. Incorrect resource setup: make sure you create the buffers passed to DLSS with the requires usage flags (see chapter 3.4). Otherwise, the output may be black without further indication as to what went wrong.

8.2 DLSS Debug Overlay

The DLSS SDK development library includes a debugging overlay which allows you to visualize the inputs used by DLSS. To enable and cycle through the various debug layers, use the CTRL+ALT+F12 hotkey. The debug layer appears as an overlay in the top right side of the screen and includes several different states including:

1. Current input color buffer

2. Current frame's motion vectors

3. Current frame's depth buffer (black is the near plane passing through blue, purple, pink with white being the far plane)

4. The history of submitted jitter offsets in an XY scatter plot [-1, 1] (green is the current offset, white and red represent old offsets that are within the correct boundary [-0.5, 0.5] and outside

it, respectively). If old, interior offsets are colored yellow, it means that the recommended number of phases has not been met yet (see section 3.7 for more information)

5. Current frame Exposure scale texture
6. A pattern that helps diagnose bad exposure texture being passed to DLSS.

To toggle the debug visualization between an overlay window in the top right of the screen and fullscreen, use the CTRL+ALT+F11 hotkey.

NOTE: The DLSS production library does **not** include the Debug Overlay.

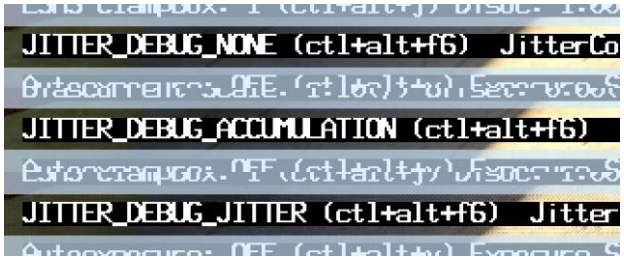
8.3 DLSS Debug Accumulation Mode

The DLSS SDK library includes a special mode that permanently accumulates all input. This can be useful when debugging issues with motion vector and jitter by examining static scenes. When enabled, a static scene should resolve *perfectly* with no aliasing, blurring or unclear areas. The static scene should look identical to the same scene rendered at full resolution without DLSS (and other AA and post).

To enable accumulation mode, press the CTRL+ALT+F6 hotkey.

The CTRL+ALT+F6 key will cycle between three modes :

- `JITTER_DEBUG_NONE` : normal DLSS display
- `JITTER_DEBUG_ACCUMULATION` : simple accumulation mode
- `JITTER_DEBUG_JITTER` : special advanced jitter debugging mode. See section **Error! Reference source not found.** below for details.



NOTE: If accumulation mode is enabled and anything in the scene moves or if the camera moves, you will see *heavy* ghosting and trails. That is expected.

8.4 Jitter Troubleshooting

As explained in section 3.7, DLSS requires sub-pixel jitter to properly resolve high-resolution output from lower resolution rendered frames. Different rendering engines implement sub-pixel jitter in different ways. Some flip or scale the amount of the offset, some apply jitter only when the camera is static, some add jitter to motion vectors. These differences can lead to graphical corruption when DLSS is applied to a frame. In such cases, try the following to debug the issue.

8.4.1 Initial Jitter Debugging

1. Use the jitter offset debug overlay to ensure the jitter offset values are **always** between -0.5 and +0.5 (inside the blue border and there are no red pixels). Also ensure that the distribution coverage and number of phases is sufficient for the target and render resolution by comparing with first “# Jitter Offsets” value against the second (see section 3.7 for more information).
2. Turn off all in-engine AA and turn off all post processing (or just render the albedo from the gbuffer).
3. With a static scene and no motion, the output from DLSS should match the same scene rendered at native full resolution (with no AA and no post processing).
4. Add a debug hotkey to the game or engine to toggle between:
    a. DLSS on with 50% rendering (typically called “DLSS Performance Mode”); and
    b. full resolution rendering with no AA (and DLSS disabled).
5. Use the in-built Microsoft Windows "Screen Magnifier" utility to zoom-in while the game is running and toggle back and forth between native rendering and DLSS.
    a. Screen Magnifier enable hotkey: Windows logo key + Plus sign (+) on the keyboard
    b. Screen Magnifier disable hotkey: Windows logo key + Esc
    c. Go into the Screen Magnifier settings and DISABLE "Smooth edges of images and text"
    d. Full information is available here: https://support.microsoft.com/en-us/help/11542/windows-use-magnifier-to-make-things-easier-to-see
6. When rendering a static scene or when looking at static objects in a scene, if the output from DLSS looks blurry or flickers or is otherwise misaligned, something is off between the renderer and DLSS.
    a. Confirm the jitter offset values are **always** between -0.5 and +0.5. They should **never** be outside this range. If they do go beyond that range, the renderer is moving the viewport outside the pixel (which will usually result in full screen shaking with DLSS on and off; or heavy blurriness when DLSS is on).
    b. Try exchanging the jitter X and Y offsets using the CTRL+ALT+F10 hotkey built into the DLSS SDK library (see section 3.8 for more information).
        i. If exchanging the axes resolves the issue, in the game or engine, exchange the jitter offsets before calling DLSS.
    c. Try negating and scaling the jitter input values using the CTRL+ALT+F9 hotkey built into the DLSS SDK library (see section 3.8 for more information).
        i. If negating or scaling one or both axes resolves the issue, in the game or engine, negate or scale the value of the appropriate component (or components) before calling DLSS.

d. Use the DLSS Debug Overlay (see section 8.2 above), or another method, to confirm that motion vectors are [0.0,0.0] for the entire screen (or at least on all static objects and areas of the screen where there is no movement).

    i. If motion vectors are not correct, fix them for the material, object, terrain or other system as required. DLSS assumes the engine provides accurate per-pixel motion vectors every frame.

8.4.2 In-depth Jitter Debugging

If the simple debugging steps above do not fix the issue, a more in-depth process may be required. The first step is to implement a configurable jitter pattern in the rendering engine.

1. First, add a "1 quadrant jitter" where the renderer always jitters to a specific quadrant every frame with a hotkey to cycle all four quadrants in order.

    a. If possible, add an on-screen indicator or debug text indicating the quadrant to which the renderer is currently jittering.

2. Second, add a "4 quadrant jitter" where the renderer jitters one frame to each quadrant cycling each quadrant in order automatically.

8.4.2.1 Single quadrant jitter debugging

1. Enable JITTER_DEBUG_JITTER mode using the CTRL+ALT+F6 hotkey (for more information, see section 8.3 above).

2. Ensure the output of DLSS is shown at 1:1 ratio on the screen. This debug mode will not be useful if ANY amount of scale change happens after DLSS is applied, either through the engine own upscaling or through the engine window being resized.

3. Find a static scene in the game and render the scene with the renderer jittering to a known and fixed quadrant.

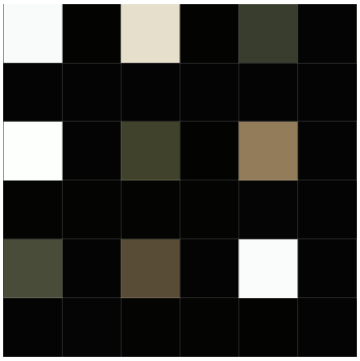
4. Examine the screen output and use Screen Magnifier to better examine individual pixels.

5. If the renderer is correctly jittering, you should see groups of four pixels with one pixel in each group colored and three pixels that are completely black.

    a. Cycle through each jitter quadrant and confirm the color moves to the correct quadrant **and** three pixels become completely black.

    b. If there are less than three black pixels in each grouping, the amount of jitter, the scale of the jitter or the sign of jitter being sent is incorrect.

Example output with single quadrant jitter.



Single quadrant jitter output under full zoom with correct jitter.
Note the groups of four pixels; three black and one colored/filled.

6. If the test in step 3 shows that the jitter is incorrect, try exchanging, negating or scaling the jitter vector components using the DLSS debug hotkeys (see section 8.4.1 above).

7. If changing the jitter scale or negating a component does not resolve the issue, find a static scene and:

    a. Capture one screenshot while the renderer jitters to each quadrant (resulting in four screenshots total).

    b. Turn on full resolution rendering, no AA, no post, no DLSS and capture one screenshot.

c. Combine the four DLSS screenshots (using Adobe Photoshop or another tool) and examine the resulting image (which **will** show heavy aliasing – that is expected). Compare the combined DLSS image against the full resolution screen capture. Look for more aliasing in one direction or if the aliasing is more pronounced vertically or horizontally. In such case, check the amount of jitter applied in the rendering engine to the projection matrix:

    i. If there is more vertical aliasing, check the Y axis jitter.

    ii. Similarly, if there is more horizontal aliasing, check the X axis.

8.4.2.2 Four quadrant jitter debugging

1. Enable DLSS JITTER_DEBUG_JITTER mode using the CTRL+ALT+F6 hotkey (for more information, see section 8.3 above).
2. Ensure the output of DLSS is shown at 1:1 ratio on the screen. This debug mode will not be useful if ANY amount of scale change happens after DLSS is applied, either through the engine own upscaling or through the engine window being resized.
3. Examine the screen output, using the Screen Magnifier to better examine individual pixels.
4. With the renderer cycling jitter to each of the four quadrants and DLSS enabled, the final image will be aliased and should match full resolution rendering with no AA, no DLSS and no post-processing.
    a. It is easiest to compare simple static flat objects like planks of wood, tree trunks, metal posts, fence posts or other objects with straight edges.
    b. Do NOT compare particles, transparent objects, or shadows.
5. If the test in step 3 shows differences in what DLSS displays versus native rendering, try exchanging, negating or scaling the jitter vector components using the DLSS debug hotkeys (see section 8.4.1 above).
6. If changing the jitter scale or negating a component does not resolve the issue, find a static scene and look for more aliasing in one direction or if the aliasing is more pronounced vertically or horizontally. In such case, check the amount of jitter applied in the rendering engine to the projection matrix:
    a. If there is more vertical aliasing, check the Y axis jitter.
    b. Similarly, if there is more horizontal aliasing, check the X axis.

8.5 Linux

For Linux, kepress Ctrl+Alt+F<NR> can switch terminals. In this case, keypress can be overridden with a config file named settings.ngxconfig and placed in ~/ngx/. For example, we can have it like in /root/ngx/settings.ngxconfig. An example of settings.ngxconfig:

```
- dlss_debug
```

```
debug_accumulation_key = "ctl+alt+f6",                  ;
debug_jitter_config_rotate_key = "ctl+alt+f9",          ;
debug_jitter_swap_coordinates_key = "ctl+alt+f10", ;
debug_visualization_size_key = "ctl+alt+fu",            ; instead of ctl+alt+f11 example
debug_visualization_mode_key = "ctl+alt+fv",            ; instead of ctl+alt+f12 example
```

8.6 Known Tooling Issues

1. RenderDoc does not support DLSS applications.

8.7 Error Codes

If an error is detected during NGX execution, NGX returns one of the following error codes:

- NVSDK_NGX_Result_FAIL_FeatureNotSupported
  Feature is not supported on current hardware.
- NVSDK_NGX_Result_FAIL_PlatformError
  Platform error - for example - check d3d12 debug layer log for more information.
- NVSDK_NGX_Result_FAIL_FeatureAlreadyExists
  Feature with given parameters already exists.
- NVSDK_NGX_Result_FAIL_FeatureNotFound
  Feature with provided handle does not exist.
- NVSDK_NGX_Result_FAIL_InvalidParameter
  Invalid parameter was provided.
- NVSDK_NGX_Result_FAIL_NotInitialized
  SDK was not initialized properly.
- NVSDK_NGX_Result_FAIL_UnsupportedInputFormat
  Unsupported format used for input/output buffers.
- NVSDK_NGX_Result_FAIL_RWFlagMissing
  Feature input/output needs RW access (UAV) (d3d11/d3d12 specific).
- NVSDK_NGX_Result_FAIL_MissingInput
  Feature was created with specific input, but none is provided at evaluation.
- NVSDK_NGX_Result_FAIL_UnableToInitializeFeature
  Feature is misconfigured or not available on the system.

- NVSDK_NGX_Result_FAIL_OutOfDate
  NGX runtime libraries are old and need an update. Inform user to install latest display driver provided by NVIDIA.
- NVSDK_NGX_Result_FAIL_OutOfGPUMemory
  Feature requires more GPU memory than is available on the system.
- NVSDK_NGX_Result_FAIL_UnsupportedFormat
  Format used in input buffer(s) is not supported by feature.
- NVSDK_NGX_Result_FAIL_UnableToWriteToAppDataPath
  Path provided in InApplicationDataPath cannot be written to
- NVSDK_NGX_Result_FAIL_UnsupportedParameter
  Unsupported parameter was provided (e.g. specific display size or mode is unsupported)
- NVSDK_NGX_Result_FAIL_ Denied
  The feature or application was denied (contact NVIDIA for further details)

9 Appendix

9.1 Transitioning from DLSS 2.0.x to 2.1.x

Moving from DLSS version 2.0.x to 2.1 should be a seamless process with the two being binary compatible. There are several new features available in v2.1.x:

1. Dynamic resolution support (see section 3.2.2)
2. Additional DLSS execution modes “Ultra-Performance” and “Ultra-Quality” (see section 3.2.1)
3. VR support (see section 3.13)
4. A pre-exposure factor is now supported if the renderer requires it (see section 3.9.2)

9.2 Minor Revision Updates

This section highlights the main updates for each minor revision:

1. 2.1.x to 2.2.x - Auto Exposure (see section 3.10)
2. 2.2.x to 2.3.x - Model improvement and opening DLSS publicly for all developers
3. 2.3.x to 2.4.x - Streamline Integration (see section 3.19)

9.3 Future DLSS Parameters

DLSS is a rolling suite of algorithms and neural networks that are under constant research and development by various groups at NVIDIA. As part of this research, NVIDIA is examining ways to use different data generated by the rendering engine to improve the overall image quality and performance of DLSS.

The following is a list of rendering engine resources that the DLSS library can optionally accept, and which may be used in future DLSS algorithms. If the developer can include some, or all, of these parameters, it can assist NVIDIA’s ongoing research and may allow future improved algorithms to be used without game or engine code changes.

For details on how to pass these resources, please see the DLSS header files and if needed, discuss with your NVIDIA technical contact. Please also check that there is no undue performance impact when preparing and providing these resources to the DLSS library.

1. G-Buffer:
    a. Albedo (supported format – 8-bit integer)
    b. Roughness (supported format – 8-bit integer)
    c. Metallic (supported format – 8-bit integer)
    d. Specular (supported format – 8-bit integer)1
    e. Subsurface (supported format – 8-bit integer)
    f. Normals (supported format – RGB10A2, Engine-dependent)

    g. Shading Model ID / Material ID : unique identifier for drawn object / material, essentially a segmentation mask for objects - common use case is to not accumulate, if the warped nearest material identifier is different from the current, (supported format – 8-bit or 16-bit integer, Engine-dependent)
2. HDR Tonemapper type: String, Reinhard, OneOverLuma or ACES
3. 3D motion vectors - (supported format – 16-bit or 32-bit floating-point)
4. Is-particle mask: to identify which pixels contains particles, essentially that are not drawn as part of base pass (supported format – 8-bit integer)
5. Animated texture mask: A binary mask covering pixels occupied by animated textures (supported format – 8-bit integer)
6. High Resolution depth: (supported format – D24S8)
7. View-space position: (supported format – 16-bit or 32-bit floating-point)
8. Frame time delta (in milliseconds): helps in determining the amount to denoise or anti-alias based on the speed of the object from motion vector magnitudes and fps as determined by this delta
9. Ray tracing hit distance: For Each effect - good approximation to the amount of noise in a ray-traced color (supported format – 16-bit or 32-bit floating-point)
10. Motion vector for reflections: motion vectors of reflected objects like for mirrored surfaces (supported format – 16-bit or 32-bit floating-point)

9.4 Notices

NVIDIA makes no representation or warranty that products based on these specifications will be suitable for any specified use without further testing or modification. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to ensure the product is suitable and fit for the application planned by customer and to do 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 specification. NVIDIA does not accept any 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 specification, 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 specification. 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 specification is permissible only if reproduction is approved by NVIDIA in writing, is reproduced without alteration, and is accompanied by all associated conditions, limitations, and notices.

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.

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 NVIDIA terms and conditions of sale for the product.

9.4.1 Trademarks

NVIDIA and the NVIDIA logo 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.

9.4.2 Copyright

www.nvidia.com

9.5 3rd Party Software

9.5.1 CURL

COPYRIGHT AND PERMISSION NOTICE

Copyright © 1996 - 2018, Daniel Stenberg, daniel@haxx.se, and many contributors, see the THANKS file.

All rights reserved.

Permission to use, copy, modify, and distribute this software for any purpose with or without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder.

9.5.2 8x13 BITMAP FONT

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

```
* (c) Copyright 1993, Silicon Graphics, Inc.
* ALL RIGHTS RESERVED
* Permission to use, copy, modify, and distribute this software for
* any purpose and without fee is hereby granted, provided that the above
* copyright notice appear in all copies and that both the copyright notice
* and this permission notice appear in supporting documentation, and that
* the name of Silicon Graphics, Inc. not be used in advertising
* or publicity pertaining to distribution of the software without specific,
* written prior permission.
*
* THE MATERIAL EMBODIED ON THIS SOFTWARE IS PROVIDED TO YOU "AS-IS"
* AND WITHOUT WARRANTY OF ANY KIND, EXPRESS, IMPLIED OR OTHERWISE,
* INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR
* FITNESS FOR A PARTICULAR PURPOSE.  IN NO EVENT SHALL SILICON
* GRAPHICS, INC.  BE LIABLE TO YOU OR ANYONE ELSE FOR ANY DIRECT,
* SPECIAL, INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES OF ANY
* KIND, OR ANY DAMAGES WHATSOEVER, INCLUDING WITHOUT LIMITATION,
* LOSS OF PROFIT, LOSS OF USE, SAVINGS OR REVENUE, OR THE CLAIMS OF
* THIRD PARTIES, WHETHER OR NOT SILICON GRAPHICS, INC.  HAS BEEN
* ADVISED OF THE POSSIBILITY OF SUCH LOSS, HOWEVER CAUSED AND ON
* ANY THEORY OF LIABILITY, ARISING OUT OF OR IN CONNECTION WITH THE
* POSSESSION, USE OR PERFORMANCE OF THIS SOFTWARE.
```

9.5.3 d3dx12.h

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

```
*     Copyright © 2017 NVIDIA Corporation.  All rights reserved.
```

```
*
*  NOTICE TO USER:
*
*  This software is subject to NVIDIA ownership rights under U.S. and international Copyright laws.
*
*  This software and the information contained herein are PROPRIETARY and CONFIDENTIAL to NVIDIA
*  and are being provided solely under the terms and conditions of an NVIDIA software license agreement
*  and / or non-disclosure agreement.  Otherwise, you have no rights to use or access this software in any manner.
*
*  If not covered by the applicable NVIDIA software license agreement:
*  NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOFTWARE FOR ANY PURPOSE.
*  IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR IMPLIED WARRANTY OF ANY KIND.
*  NVIDIA DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
*  INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
*  IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES,
*  OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,  WHETHER IN AN ACTION OF CONTRACT,
*  NEGLIGENCE OR OTHER TORTIOUS ACTION,  ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOURCE CODE.
*
*  U.S. Government End Users.
*  This software is a "commercial item" as that term is defined at 48 C.F.R. 2.101 (OCT 1995),
*  consisting  of "commercial computer  software"  and "commercial computer software documentation"
*  as such terms are  used in 48 C.F.R. 12.212 (SEPT 1995) and is provided to the U.S. Government only as a commercial
end item.
*  Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through 227.7202-4 (JUNE 1995),
*  all U.S. Government End Users acquire the software with only those rights set forth herein.
*
*  Any use of this software in individual and commercial software must include,
*  in the user documentation and internal comments to the code,
*  the above Disclaimer (as applicable) and U.S. Government End Users Notice.
*

* Original code by Microsoft Corporation was obtained from
*  https://github.com/Microsoft/DirectX-Graphics-Samples/tree/master/Samples/Desktop/D3D12Multithreading
*  and licensed under MIT license (below)
*  NVIDIA license applies to changes made relative to original code.

* Copyright (c) Microsoft. All rights reserved.
* This code is licensed under the MIT License (MIT).
* THIS CODE IS PROVIDED *AS IS* WITHOUT WARRANTY OF
* ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING ANY
* IMPLIED WARRANTIES OF FITNESS FOR A PARTICULAR
* PURPOSE, MERCHANTABILITY, OR NON-INFRINGEMENT.
```

9.5.4 xml

https://pugixml.org/license.html

```
* This library is available to anybody free of charge, under the terms of MIT License:
* Copyright (c) 2006-2018 Arseny Kapoulkine
* Permission is hereby granted, free of charge, to any person obtaining a copy of this
* software and associated documentation files (the “Software”), to deal in the Software
* without restriction, including without limitation the rights to use, copy, modify, merge,
* publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons
* to whom the Software is furnished to do so, subject to the following conditions:
* The above copyright notice and this permission notice shall be included in all copies or
* substantial portions of the Software.
* THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
* INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR
* PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
* FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
* OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
* DEALINGS IN THE SOFTWARE.
* This means that you can freely use pugixml in your applications, both open-source and
* proprietary. If you use pugixml in a product, it is sufficient to add an acknowledgment
* like this to the product distribution:
* This software is based on pugixml library (http://pugixml.org). pugixml is Copyright (C) *
2006-2018 Arseny Kapoulkine.
```

9.5.5 npy

https://github.com/llohse/libnpy

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

```
* MIT License

* Copyright (c) 2021 Leon Merten Lohse

* Permission is hereby granted, free of charge, to any person obtaining a copy
* of this software and associated documentation files (the "Software"), to deal
* in the Software without restriction, including without limitation the rights
* to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
* copies of the Software, and to permit persons to whom the Software is
* furnished to do so, subject to the following conditions:

* The above copyright notice and this permission notice shall be included in all
* copies or substantial portions of the Software.

* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
* IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
* FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
* AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
* LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
* OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
* SOFTWARE.
```

9.5.6 stb

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

```
/*
* ALTERNATIVE A - MIT License
* Copyright (c) 2017 Sean Barrett
* Permission is hereby granted, free of charge, to any person obtaining a copy of
* this software and associated documentation files (the "Software"), to deal in
* the Software without restriction, including without limitation the rights to
* use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies
* of the Software, and to permit persons to whom the Software is furnished to do
* so, subject to the following conditions:
* The above copyright notice and this permission notice shall be included in all
* copies or substantial portions of the Software.
* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
* IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
* FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
* AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
* LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
* OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
* SOFTWARE.
*/
```

9.5.7 DirectX-Graphics-Samples

https://github.com/Microsoft/DirectX-Graphics-Samples/tree/master/Samples/Desktop/D3D12Multithreading

```
/*
* Copyright (c) Microsoft. All rights reserved.
* This code is licensed under the MIT License (MIT).
* THIS CODE IS PROVIDED *AS IS* WITHOUT WARRANTY OF
* ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING ANY
* IMPLIED WARRANTIES OF FITNESS FOR A PARTICULAR
* PURPOSE, MERCHANTABILITY, OR NON-INFRINGEMENT.
*/
```

9.6 Linux driver compatibility

The Linux driver no longer adheres to a strict minimum driver version compatibility model that can be queried by `NVSDK_NGX_Parameter_SuperSampling_MinDriverVersionMajor`.

Any NVIDIA Linux driver released starting 22 March 2022 (for instance, NVIDIA Linux driver 510.60.02) will only support DLSS snippets version >= 2.4.0.

Any NVIDIA Linux driver released prior to that date will only support DLSS snippets version < 2.4.0.

Applications can support both old and new drivers by shipping with two snippets (one < 2.4.0, and one >= 2.4.0). In that case the application must have the paths to both snippets in `NVSDK_NGX_PathListInfo` (first – to the older snippet, then – to a newer snippet). The driver will access those snippets in order and use the first snippet it is able to load.

Windows versions of DLSS snippets (including those running on Steam Play Proton) are not affected by this restriction.