NVIDIA Image Scaling SDK (version 1.0.1)

Programming Guide

Document revision: 1.0.1

Released: 13 December 2021

Table of Contents

Abstract

The NVIDIA Image Scaling SDK Programming Guide provides details on how to integrate NVIDIA scaling and sharpening algorithms in a game or 3D application. The Guide also provides some code snippets and links to a full sample implementation in GitHub. In addition, it describes the best practices and a detailed description of the algorithm.

Revision History

1.0.1	Performance optimizations fp16 coefficients support GLSL support DX12 and Vulkan samples	13 December 2021
1.0.0	Initial release	16 November 2021

1 Introduction

The NVIDIA Image Scaling SDK provides a single spatial scaling and sharpening algorithm for cross-platform support. The scaling algorithm uses a 6-tap scaling filter combined with 4 directional scaling and adaptive sharpening filters, which creates nice smooth images and sharp edges. In addition, the SDK provides a state-of-the-art adaptive directional sharpening algorithm for use in applications where no scaling is required. By integrating both NVIDIA Image Scaling and NVIDIA DLSS, developers can get the best of both worlds: NVIDIA DLSS for the best image quality, and NVIDIA Image Scaling for cross-platform support.

The directional scaling and sharpening algorithm are named NVScaler while the adaptive-directional-sharpening-only algorithm is named NVSharpen. Both algorithms are provided as compute shaders and developers are free to integrate them in their applications. Note that if you integrate NVScaler, you should NOT integrate NVSharpen, as **NVScaler already includes a sharpening pass.**

2 Getting Started

2.1 System Requirements

The following is needed to load and run NVScaler and NVSharpen:

- PC with Windows 10 v1709 or newer
- A DX11, DX12, or Vulkan compatible GPU

The sample app included with SDK can be compiled for Windows or Linux.

2.2 Rendering Engine Requirements

The compute shaders can be integrated with Direct3D and Vulkan, and the application or rendering engine must:

- Support High-level Shader Language (HSLS) model 5.0 or higher, or OpenGL Shader Language (GLSL) 4.50.7 (version 450)
- The HLSL shader integration requires DirectX11, DirectX12, or Vulkan support
- Support a high-quality anti-aliasing technique like TAA or FXAA
- Ability to negatively bias the LOD for textures and geometry
- One each shader call (i.e., each frame), provide:
    - The raw color texture for the frame (in HDR or SDR in display-referred color-space)
    - The output texture with the right dimensions
    - For NVScaler: scale and sharpness values along with configuration and coefficients
    - For NVSharpen: sharpness value and configuration values

To allow for future compatibility and ease ongoing research by NVIDIA, the application should consider integrating NVIDIA Image Scaling SDK compute shaders without modifications.

2.3 Sample Requirements

The sample included with NVIDIA Image Scaling SDK requires the following tools:

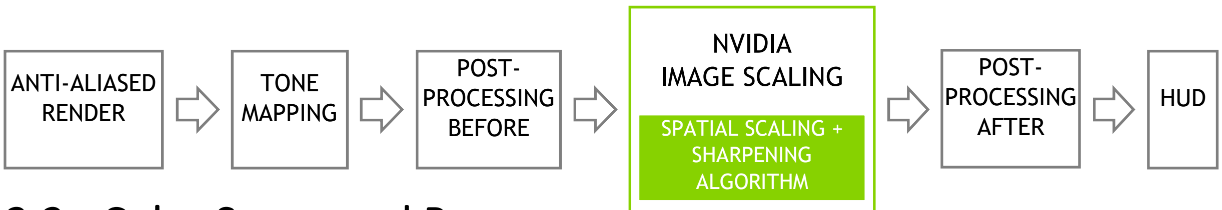
- CMake 3.12 and up
- Visual Studio 2019
- Windows SDK
- Vulkan (If the Vulkan sample is enabled)

3 NVIDIA Image Scaling Shaders Integration

3.1 Pipeline Placement

The call into the NVIDIA Image Scaling shaders must occur during the post-processing phase after tone-mapping. Applying the scaling in linear HDR in-game color-space may result in a sharpening effect that is either not visible or too strong.

Since sharpening algorithms can enhance noisy or grainy regions, it is recommended that certain effects such as film grain should occur after NVScaler or NVSharpen. Low-pass filters such as motion blur or light bloom are recommended to be applied before NVScaler or NVSharpen to avoid sharpening attenuation.



3.2 Color Space and Ranges

NVIDIA Image Scaling shaders can process color textures stored as either LDR or HDR with the following restrictions:

1) LDR

- The range of color values must be in the [0, 1] range
- The input color texture must be in display-referred color-space after tone mapping and OETF (gamma-correction) has been applied

2) HDR PQ

- The range of color values must be in the [0, 1] range
- The input color texture must be in display-referred color-space after tone mapping with Rec. 2020 PQ OETF applied

3) HDR Linear

- The recommended range of color values is [0, 12.5], where luminance value (as per BT. 709) of 1.0 maps to brightness value of 80nits (sRGB peak) and 12.5 maps to 1000nits
- The input color texture may have luminance values that are either linear and scene-referred or linear and display-referred (after tone mapping)

If the input color texture sent to NIS is in HDR format set NIS_HDR_MODE define to either NIS_HDR_MODE_LINEAR (1) or NIS_HDR_MODE_PQ (2).

3.3 Supported Texture Formats

3.3.1 Input and output formats

Input and output formats are expected to be in the rages defined in previous section and should be specified using non-integer data types such as DXGI_FORMAT_R8G8B8A8_UNORM.

3.3.2 Coefficients formats

The scaler coefficients and USM coefficients format should be specified using float4 type such as DXGI_FORMAT_R32G32B32A32_FLOAT or DXGI_FORMAT_R16G16B16A16_FLOAT.

3.4 Resource States, Buffers, and Sampler

The game or application calling NVIDIA Image Scaling SDK shaders must ensure that the textures are in the correct state.

- Input color textures must be in pixel shader read state
    - Shader Resource View (SRV) in DirectX
    - Sample Image in Vulkan
- The output texture must be in read/write state
    - Unordered Access View (UAV) in DirectX
    - Storage Image in Vulkan
- The coefficients texture for NVScaler must be in read state
    - Shader Resource View (SRV) in DirectX
    - Sample Image in Vulkan
- The configuration variables must be passed as constant buffer
    - Constant Buffer View (CBV) in DirectX
    - Uniform buffer in Vulkan
- The sampler for texture pixel sampling
    - Linear clamp SamplerState in DirectX

4 Adding NVIDIA Image Scaling SDK to a Project

4.1 Integration with your framework

The integration instructions in this section can be applied with minimal changes to your own DX11, DX12, or Vulkan application, using HLSL or GLSL.

4.1.1 SDK package files, configuration tools, and constant definitions

Device

- NIS_Scaler.h: shader file, contains NVScaler and NVSharpen implementations
- NIS_Main.hlsl: main HLSL shader example (can be replaced by your own)
- NIS_Main.glsl: main GLSL shader example (can be replaced by your own)

Host Configuration

- NIS_Config.h: Coefficients in fp32 an fp16 formats and configuration declarations

Defines

- **NIS_SCALER**: 1 enable NvScaler, 0 performs fast NvSharpen only (no upscaling)
- **NIS_HDR_MODE**: 0 disabled, 1 Linear, 2 PQ
- **NIS_BLOCK_WIDTH**: pixels per block width. Use GetOptimalBlockWidth query for your platform
- **NIS_BLOCK_HEIGHT**: pixels per block height. Use GetOptimalBlockHeight query for your platform
- **NIS_THREAD_GROUP_SIZE**: number of threads per group. Use GetOptimalThreadGroupSize query for your platform
- **NIS_VIEWPORT_SUPPORT:** 0 disabled, 1 enable input/output viewport support
- **NIS_USE_HALF_PRECISION:** 0 disabled, 1 enable half pression computation
- **NIS_HLSL:** 0 disabled, 1 enable HLSL support
- **NIS_HLSL_6_2:** 0 HLSL v5, 1 HLSL v6.2 forces NIS_HSLS=1
- **NIS_GLSL:** 0 disabled, 1 enable GLSL support

Defines for HLSL with DXC bindings

- **NIS_DXC:** 0 disabled, 1 enable HLSL DXC Vulkan support

Optimal shader settings

To get optimal performance of NVScaler and NVSharpen for current and future hardware, it is recommended that the following API is used to obtain the values for **NIS_BLOCK_WIDTH**, **NIS_BLOCK_HEIGHT**, and **NIS_THREAD_GROUP_SIZE**. These values can be used to compile permutations of NVScaler and NVSharpen offline.

```
enum class NISGPUArchitecture : uint32_t
{
    NVIDIA_Generic = 0,
    AMD_Generic = 1,
    Intel_Generic = 2
};
```

```
struct NISOptimizer
{
    bool isUpscaling;
    NISGPUArchitecture gpuArch;

    NISOptimizer(bool isUpscaling = true,
                 NISGPUArchitecture gpuArch = NISGPUArchitecture::NVIDIA_Generic);
    uint32_t GetOptimalBlockWidth();
    uint32_t GetOptimalBlockHeight();
    uint32_t GetOptimalThreadGroupSize();
};
```

HDR shader settings

Use the following enum values for setting **NIS_HDR_MODE**

```
enum class NISHDRMode : uint32_t
{
    None = 0,
    Linear = 1,
    PQ = 2
};
```

4.1.2 Integration of NVScaler

Compile the NIS_Main.hlsl shader

NIS_SCALER should be set to **1**, and the isUpscaling argument should set to **true**.

```
bool isUpscaling = true;
NISOptimizer opt(isUpscaling, NISGPUArchitecture::NVIDIA_Generic);
uint32_t blockWidth = opt.GetOptimalBlockWidth();
uint32_t blockHeight = opt.GetOptimalBlockHeight();
uint32_t threadGroupSize = opt.GetOptimalThreadGroupSize();

Defines defines;
defines.add("NIS_SCALER", isUpscaling);
defines.add("NIS_HDR_MODE", hdrMode);
defines.add("NIS_BLOCK_WIDTH", blockWidth);
defines.add("NIS_BLOCK_HEIGHT", blockHeight);
defines.add("NIS_THREAD_GROUP_SIZE", threadGroupSize);
NVScalerCS = CompileComputeShader(device, "NIS_Main.hlsl", &defines);
```

Note: Compilation of the shaders permutations can be performed off-line.

Create NVIDIA Image Scaling SDK configuration constant buffer

```
struct NISConfig
{
    float kDetectRatio;
    float kDetectThres;
    float kMinContrastRatio;
    float kRatioNorm;
    ...
};

NISConfig config;
createConstBuffer(&config, &csBuffer);
```

Create SRV textures for the scaler and USM phase coefficients

```
const int rowPitch = kFilterSize * sizeof(float); // use fp32: float, fp16: uint16_t
const int coefSize = rowPitch * kPhaseCount;
// since we are using RGBA format the texture width = kFilterSize / 4
createTexture2D(kFilterSize / 4, kPhaseCount, DXGI_FORMAT_R32G32B32A32_FLOAT,
D3D11_USAGE_DEFAULT, coef_scaler, rowPitch, coefSize, &scalerTex);
createTexture2D(kFilterSize / 4, kPhaseCount, DXGI_FORMAT_R32G32B32A32_FLOAT,
D3D11_USAGE_DEFAULT, coef_usm, rowPitch, coefSize, &usmTex);

createSRV(scalerTex.Get(), DXGI_FORMAT_R32G32B32A32_FLOAT, &scalerSRV);
createSRV(usmTex.Get(), DXGI_FORMAT_R32G32B32A32_FLOAT, &usmSRV);
```

Note: It is also possible to specify an fp16 format for the coefficients such as DXGI_FORMAT_R16G16B16A16_FLOAT. Use the fp16 coefficients coef_scaler_fp16 and coef_usm_fp16.

Create Sampler

```
createLinearClampSampler(&linearClampSampler);
```

Update NVIDIA Image Scaling SDK configuration and constant buffer

Use the following API call to update the NVIDIA Image Scaling SDK configuration

```
bool NVScalerUpdateConfig(NISConfig& config,
    float sharpness,
    uint32_t inputViewportOriginX, uint32_t inputViewportOriginY,
    uint32_t inputViewportWidth, uint32_t inputViewportHeight,
    uint32_t inputTextureWidth, uint32_t inputTextureHeight,
    uint32_t outputViewportOriginX, uint32_t outputViewportOriginY,
    uint32_t outputViewportWidth, uint32_t outputViewportHeight,
    uint32_t outputTextureWidth, uint32_t outputTextureHeight,
    NISHDRMode hdrMode = NISHDRMode::None
);
```

NVScalerUpdateConfig returns true if the configuration was successful and false if the configuration could not be set. The input texture sizes should be less or equal than the output texture sizes. The algorithm does not check for viewport inconsistencies.

When viewports are required compile the shader with NIS_VIEWPORT_SUPPORT = 1. The use of viewports might impact performance when the viewport size is closed to the texture sizes. To improve performance, consider adjusting the dispatch dimensions to accommodate the viewport size. Ideally the outputViewport size should be in the same ratio as the input and output texture sizes.

Update the constant buffer whenever the input size, sharpness, or scale changes

```
NISUpdateConfig(m_config, sharpness,
                0, 0, inputWidth, inputHeight, inputWidth, inputHeight,
                0, 0, outputWidth, outputHeight, outputWidth, outputHeight,
                NISHDRMode::None);

updateConstBuffer(&config, csBuffer.Get());
```

A simple DX11 NVScaler dispatch example

```
context->CSSetShaderResources(0, 1, input); // SRV
context->CSSetShaderResource (1, 1, scalerSRV.GetAddressOf());
context->CSSetShaderResource (2, 1, usmSRV.GetAddressOf());
context->CSSetUnorderedAccessViews(0, 1, output, nullptr);
context->CSSetSamplers(0, 1, linearClampSampler.GetAddressOf());
context->CSSetConstantBuffers(0, 1, csBuffer.GetAddressOf());
context->CSSetShader(NVScalerCS.Get(), nullptr, 0);

context->Dispatch(UINT(std::ceil(outputWidth / float(blockWidth))),
                  UINT(std::ceil(outputHeight / float(blockHeight))), 1);
```

4.1.3 *Integration of NVSharpen*

If your application requires upscaling and sharpening **do not use NVSharpen** use NVScaler instead. Since NVScaler performs both operations, upscaling and sharpening, in one step, it performs faster and produces better image quality.

Compile the NIS_Main.hlsl shader

NIS_SCALER should be set to **0** and the optimizer isUscaling argument should be set as **false**.

```
bool isUpscaling = false;
NISOptimizer opt(isUpscaling, NISGPUArchitecture::NVIDIA_Generic);
uint32_t blockWidth = opt.GetOptimalBlockWidth();
uint32_t blockHeight = opt.GetOptimalBlockHeight();
uint32_t threadGroupSize = opt.GetOptimalThreadGroupSize();

Defines defines;
defines.add("NIS_DIRSCALER", isUpscaling);
defines.add("NIS_HDR_MODE", hdrMode);
defines.add("NIS_BLOCK_WIDTH", blockWidth);
defines.add("NIS_BLOCK_HEIGHT", blockHeight);
defines.add("NIS_THREAD_GROUP_SIZE", threadGroupSize);
NVSharpenCS = CompileComputeShader(device, "NIS_Main.hlsl”, &defines);
```

Note: Compilation of the shaders permutations can be performed off-line.

Create NVIDIA Image Scaling SDK configuration constant buffer

```
struct NISConfig
{
    float kDetectRatio;
    float kDetectThres;
    float kMinContrastRatio;
    float kRatioNorm;
    ...
};

NISConfig config;
createConstBuffer(&config, &csBuffer);
```

Create Sampler

```
createLinearClampSampler(&linearClampSampler);
```

Update NVIDIA Image Scaling SDK configuration and constant buffer

Use the following API call to update the NVIDIA Image Scaling SDK configuration. Since NVSharpen is a sharpening algorithm only the sharpness and input size are required. **For upscaling with sharpening use NVScaler** since it performs both operations at the same time.

```
bool NVSharpenUpdateConfig(NISConfig& config, float sharpness,
    uint32_t inputViewportOriginX, uint32_t inputViewportOriginY,
    uint32_t inputViewportWidth, uint32_t inputViewportHeight,
    uint32_t inputTextureWidth, uint32_t inputTextureHeight,
    uint32_t outputViewportOriginX, uint32_t outputViewportOriginY,
    NISHDRMode hdrMode = NISHDRMode::None
```

```
);
```

Update the constant buffer whenever the input size or sharpness changes.

```
NVSharpenUpdateConfig(m_config, sharpness,
                      0, 0, inputWidth, inputHeight, inputWidth, inputHeight,
                      0, 0, NISHDRMode::None);

updateConstBuffer(&config, csBuffer.Get());
```

A simple DX11 NVSharpen dispatch example

```
context->CSSetShaderResources(0, 1, input);
context->CSSetUnorderedAccessViews(0, 1, output, nullptr);
context->CSSetSamplers(0, 1, linearClampSampler.GetAddressOf());
context->CSSetConstantBuffers(0, 1, csBuffer.GetAddressOf());
context->CSSetShader(NVSharpenCS.Get(), nullptr, 0);

context->Dispatch(UINT(std::ceil(outputWidth / float(blockWidth))),
                  UINT(std::ceil(outputHeight / float(blockHeight))), 1);
```

4.2 Sample Code

A sample code is provided with the NVIDIA Image Scaling SDK. The sample apps are very simple examples of how to integrate NVScaler or NVSharpen in your application.

To compile the samples:

```
$> cd samples
$> mkdir build
$> cd build
$> cmake ..
```

Open the solution with Visual Studio 2019. Right-click the sample project and select "Set as Startup Project" before building the project.

For building the Vulkan sample

```
$> cd samples
$> mkdir build
$> cd build
$> cmake .. -DNIS_VK_SAMPLE=ON
```

5 Best Practices

5.1 Mip-Map Bias

The application should set the mip-map bias (also called the texture LOD bias) to a value lower than 0. This improves the overall image quality as textures are sampled at the display resolution rather than the lower render resolution in use with NVScaler. NVIDIA recommends

```
MipLevelBias = NativeBias + log2(Render XResolution / Display XResolution) + epsilon
```

Note: Carefully check texture clarity when NVScaler is enabled and confirm that it matches the texture clarity when rendering at native resolution with 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 NVScaler is enabled, the bias may be too large or too small compared to the default leading too poor image quality. In such case, adjust the “epsilon” value for the MipLevelBias calculation.

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

6 Appendix

6.1 Notices

6.1.1 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.

6.1.2 License

6.2 Third-party Software

6.2.1 Dear ImGui

6.2.2 GLFW

6.2.3 DirectX Shader Compiler

```
    University of Illinois at Urbana-Champaign

    http://llvm.org

Permission is hereby granted, free of charge, to any person obtaining a copy of
this software and associated documentation files (the "Software"), to deal with
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:

    * Redistributions of source code must retain the above copyright notice,
      this list of conditions and the following disclaimers.

    * Redistributions in binary form must reproduce the above copyright notice,
      this list of conditions and the following disclaimers in the
      documentation and/or other materials provided with the distribution.

    * Neither the names of the LLVM Team, University of Illinois at
      Urbana-Champaign, nor the names of its contributors may be used to
      endorse or promote products derived from this Software without specific
      prior written permission.

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
CONTRIBUTORS 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 WITH THE
SOFTWARE.

==============================================================================
Copyrights and Licenses for Third Party Software Distributed with LLVM:
==============================================================================
The LLVM software contains code written by third parties.  Such software will
have its own individual LICENSE.TXT file in the directory in which it appears.
This file will describe the copyrights, license, and restrictions which apply
to that code.

The disclaimer of warranty in the University of Illinois Open Source License
applies to all code in the LLVM Distribution, and nothing in any of the
other licenses gives permission to use the names of the LLVM Team or the
University of Illinois to endorse or promote products derived from this
Software.

The following pieces of software have additional or alternate copyrights,
licenses, and/or restrictions:

Program             Directory
-------             ---------
Autoconf            llvm/autoconf
                    llvm/projects/ModuleMaker/autoconf
Google Test         llvm/utils/unittest/googletest
OpenBSD regex       llvm/lib/Support/{reg*, COPYRIGHT.regex}
pyyaml tests        llvm/test/YAMLParser/{*.data, LICENSE.TXT}
ARM contributions   llvm/lib/Target/ARM/LICENSE.TXT
md5 contributions   llvm/lib/Support/MD5.cpp llvm/include/llvm/Support/MD5.h
miniz               llvm/lib/Miniz/miniz.c llvm/include/miniz/miniz.h
llvm/lib/Miniz/LICENSE.txt
```

6.2.4 tinyEXR

```
License
3-clause BSD

tinyexr uses miniz, which is developed by Rich Geldreich richgel99@gmail.com, and licensed
under public domain.

tinyexr tools uses stb, which is licensed under public domain: https://github.com/nothings/stb
tinyexr uses some code from OpenEXR, which is licensed under 3-clause BSD license.
```

6.2.5 stb image

```
This software is available under 2 licenses -- choose whichever you prefer.
------------------------------------------------------------------------------
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.
------------------------------------------------------------------------------
ALTERNATIVE B - Public Domain (www.unlicense.org)
This is free and unencumbered software released into the public domain.
Anyone is free to copy, modify, publish, use, compile, sell, or distribute this
software, either in source code form or as a compiled binary, for any purpose,
commercial or non-commercial, and by any means.
In jurisdictions that recognize copyright laws, the author or authors of this
software dedicate any and all copyright interest in the software to the public
domain. We make this dedication for the benefit of the public at large and to
the detriment of our heirs and successors. We intend this dedication to be an
overt act of relinquishment in perpetuity of all present and future rights to
this software under copyright law.
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 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.
```

6.2.6 d3dx12.h

```
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.
```