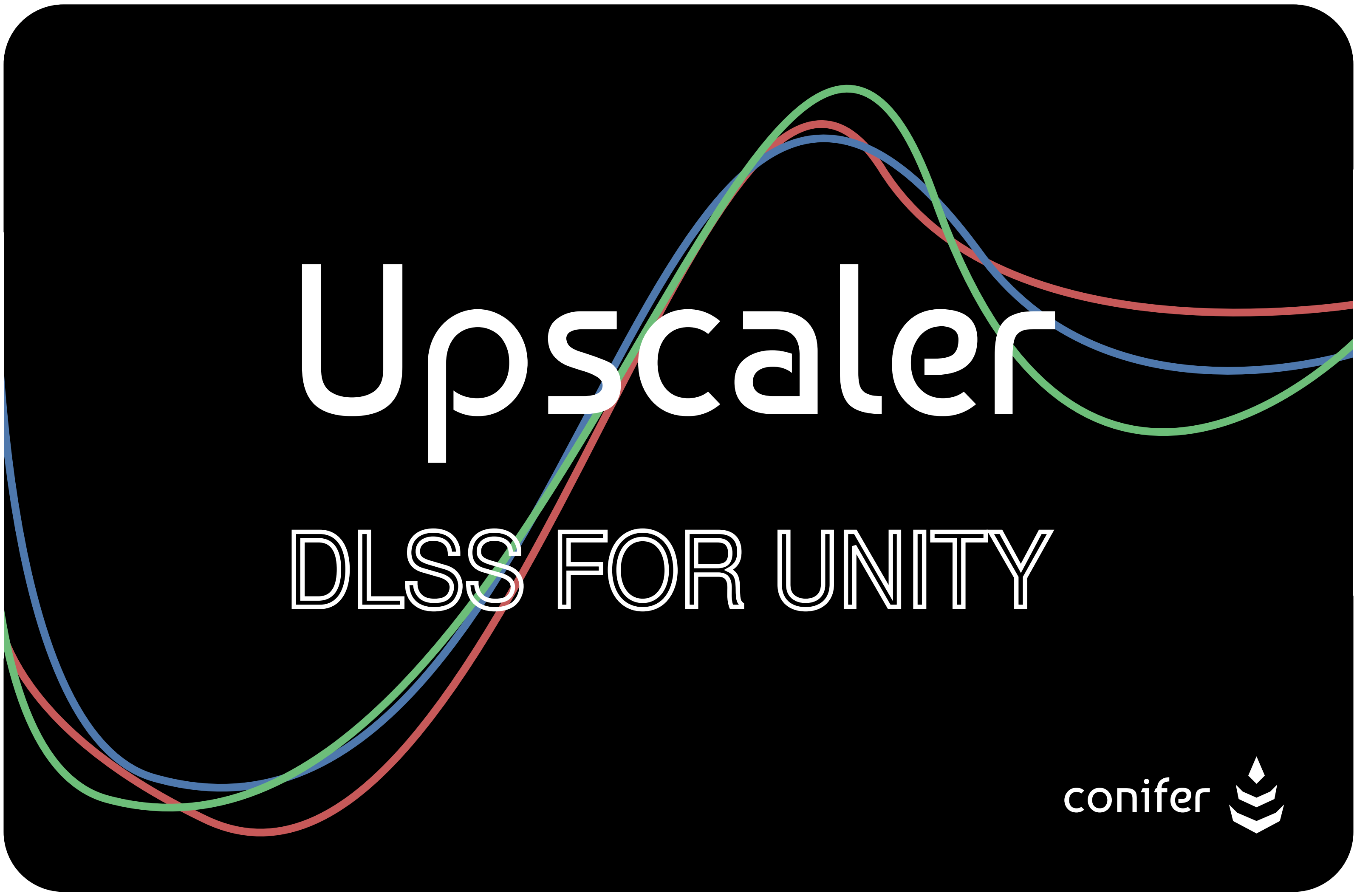


Upscaler Offline Manual (PDF)

v1.0.0

Table of Contents

1 Using the .unitypackage File......3
2 Using the Unity Editor GUI......3
2.1 Upscaler Settings......3
2.2 Basic Upscaler Settings......4
2.3 Advanced Upscaler Settings......5
3 Understanding the Upscaler API......5
3.1 A Detailed Deepdive......5
3.1.1 The RenderPipeline Class......6
3.1.1.1 Effects on Rendering When Using the URP......6
3.1.1.1.1 URP Upscaling Stage One......7
3.1.1.1.2 URP Upscaling Stage Two......7
3.1.1.2 Effects on Rendering When Using the BRP......8
3.1.1.2.1 BRP Upscaling Stage One......8
3.1.1.2.2 BRP Upscaling Stage Two......8
3.1.2 The Upscaler Class......9
3.1.3 The Jitter Class......9
3.2 Changing Settings......10
3.3 Handling Errors......10
3.4 API Reference......11
4 Things to be Aware of......11
5 GUI Integration......12

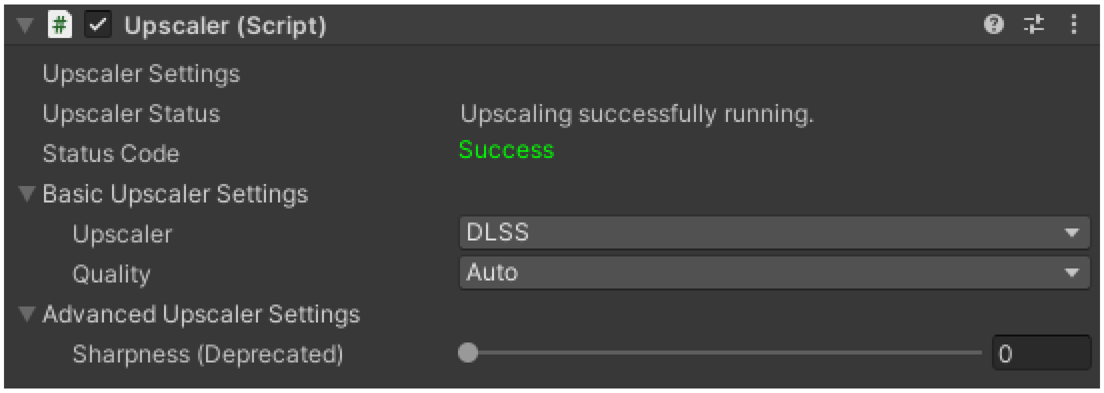
1 Using the .unitypackage File

For a video version of this consult the Quick Start Guide video on www.conifercomputing.com.

After downloading Upscaler's *.unitypackage file from the Asset Store, drag and drop the file onto the Unity Editor. Allow the Unity Editor to import the package. Go to the Assets→Upscaler directory and drag and drop the `Upscaler` script onto your main camera. Press the "Play" button to see the upscaling magic.

Consult section 2 Using the Unity Editor GUI for more information on how to control the upscaling from the Unity Editor. Consult section 3.2 Changing Settings to learn how to control the upscaling from within your scripts. Finally check out 5 GUI Integration in order to add upscaling controls to your game's GUI.

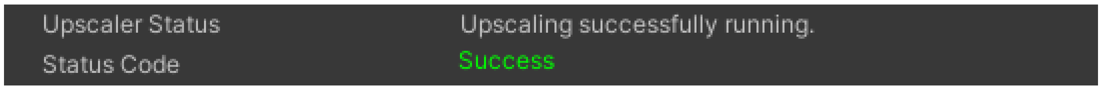
2 Using the Unity Editor GUI



Upscaler Settings
Upscaler Status: Upscaling successfully running.
Status Code: Success
Basic Upscaler Settings
Upscaler: DLSS
Quality: Auto
Advanced Upscaler Settings
Sharpness (Deprecated): 0

When correctly attached to the camera, Upscaler's settings should appear as above in the camera's inspector. This section provides detailed descriptions on the uses of each of the elements in this GUI.

2.1 Upscaler Settings

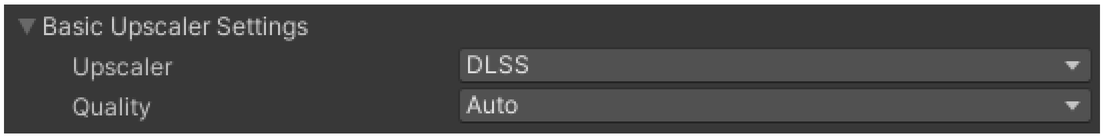


Upscaler Status: Upscaling successfully running.
Status Code: Success

The "Upscaler Status" field will provide a description of Upscaler's current state in plain English. When Upscaler is behaving as expected it will read "Upscaling successfully running." If Upscaler has encountered any type of error this field will change to read "Upscaling encountered an Error. Fell back to the 'None' Upscaler." When this message appears it means that the DLSS upscaler has encountered an error and was unable to continue. Upscaler has automatically reverted settings to use the “None” or dummy upscaler. For more information on how to properly handle and recover from these errors consult section 3.3 Handling Errors.

The “Status Code” field provides a more accurate description of the error. It also will take on the name of the value of the `enum UpscalerStatus` that is represented by `Upscaler.Status`. For a list of all possible values for this field consult section 3.4 API Reference. Internally two of the possible states for this field represent success states, states in which Upscaler is still behaving as desired. They are `Success` and `NoUpscalerSet`. Those will both appear in the color green. Anything that appears in red is an error.

2.2 Basic Upscaler Settings



Basic Upscaler Settings

Upscaler	DLSS
Quality	Auto

The “Upscaler” dropdown menu allows you to select one of the available upscalers. *Currently*, there are just two: “None” and “DLSS”. Selecting “None” will enable the dummy upscaler. This upscaler does nothing and forces the game to render at the output resolution. It ignores the “Quality” setting; ultimately, it behaves as an off switch for Upscaler. Selecting “DLSS” will enable the DLSS on this camera. DLSS passes motion vectors, the depth buffer, and the color buffer through a neural network to quickly perform high quality upscaling. This setting will respect your selection in the “Quality” field.

The “Quality” dropdown menu allows you to select a quality mode to run DLSS at. This setting is ignored if the “Upscaler” option is set to “None”. DLSS has five quality modes, four of which scale by a constant amount, and “Auto” which chooses another quality mode as given by the output resolution. Consult the charts below for more information.

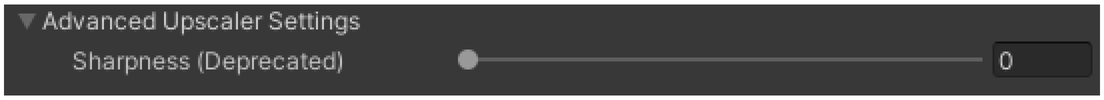
DLSS Quality Mode Effects on Render Resolution

Quality Mode	% less rendered pixels per-axis	% of total pixels rendered
Quality	33.$\overline{3}$%	44.$\overline{4}$%
Balanced	42%	31%
Performance	50%	25%
Ultra Performance	66.$\overline{67}$%	11.$\overline{1}$%

DLSS "Auto" Quality Mode Choice

Output Resolution	Chosen Quality Mode
Less than 1920x1080	DLSS is disabled
Less than or equal to 2560x1440	Quality
Less than or equal to 3840x2160	Performance
Greater than 3840x2160	Ultra Performance

2.3 Advanced Upscaler Settings



The "Sharpness" slider controls the amount of sharpening that DLSS will perform. This setting is ignored when the *Basic Upscaler Settings→Upscaler* field is set to "None". Please note that this setting has been deprecated by NVIDIA. Even when DLSS is active and this slider is set to a non-zero value, it may be true that no sharpening effect is visible. NVIDIA has provided no replacement and has instead recommended rolling a custom sharpness solution. Conifer also recommends rolling a custom sharpness solution, as even when DLSS's sharpening does work, it often produces very poor results.

3 Understanding the Upscaler API

Upscaler's API is made up of two parts that work together: a C# wrapper and a C++ library. The C++ library directly communicates with the graphics API and the DLSS shared library, providing a set of abstracted functions that enable the use of DLSS with any of its supported graphics APIs. The job of the C# layer is to connect the C++ library to Unity and allow C# programmers to manipulate the state of the C++ library.

Programmers are intended to use the C# library exclusively. While it is possible for programmers to interact with the C++ portion of the Upscaler, it should never be necessary. If there is needed functionality that can only be attained by interacting with the C++ library, then it is a bug and should be reported. The details of the C++ library are not documented here.

3.1 A Detailed Deepdive

This section is aimed at those who need a deeper understanding of the internals of Upscaler. It does not detail the C++ library, but it does explain the finer points of the C# layer's implementation. It includes information on how using the plugin will effect Unity's graphics pipeline, what each class is intended to do, and the ways that you indirectly influence it.

3.1.1 The `RenderPipeline` Class

The C# layer abstracts the way that the Universal Render Pipeline (URP) and Builtin Render Pipeline (BRP) must interact with the C++ library and with Unity's graphics systems into an abstract `RenderPipeline` class that has `Universal` and `Builtin` as child classes. During the `OnEnable` function of the camera, one of those two child classes are chosen to be instantiated based on the state of `GraphicsSettings.currentRenderPipeline`. The resulting `RenderPipeline` object is then used whenever anything related to interacting with Unity's graphics comes up, including updating DLSS's Unity-managed image resources and updating Unity's `CommandBuffer`.

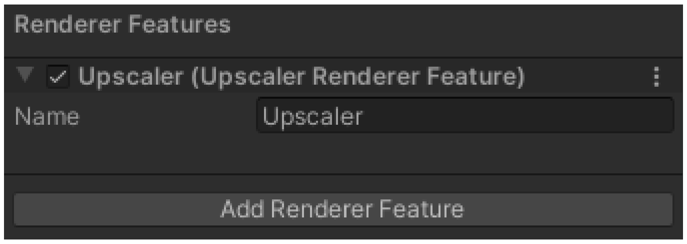
Despite which Unity render pipeline you actually use, you are guaranteed two things about the inputs and outputs of the upscaling process:

1. The texture that will be processed by the upscaler will be the one attached to `Camera.targetTexture` at the beginning of the frame.
2. The depth buffer attached to the camera after upscaling will be at output resolution (upsampled using nearest neighbor scaling).

Furthermore, setting the upscaler to "None" completely removes Upscaler from the graphics pipeline. It will behave exactly as Unity's unmodified graphics pipeline would.

3.1.1.1 Effects on Rendering When Using the URP

The `RenderPipeline` object is actually an instance of the `Universal` class. When instantiated, the `Universal` object uses C#'s reflection API to automatically add itself to the list of `rendererFeatures`. It will appear as below when the application is playing. It will only be visible when the application is playing as it automatically removes itself when the application is stopped.


Renderer Features

Upscaler (Upscaler Renderer Feature)

Name: Upscaler

This Renderer Feature performs two jobs. First, it handles all requests for image resource updates from 3.1.2 The `Upscaler` Class. Second, it handles maintaining the `CommandBuffer`s that are used to prepare and perform upscaling. This will change the default behaviour of the URP. There are two stages during which possibly breaking changes occur. Please read through this section carefully and evaluate these stages' impacts upon your current pipeline.

3.1.1.1.1 URP Upscaling Stage One

The first stage is bound to the `RenderPipelineManager.beginCameraRendering` event. Its job is to ensure that valid and suitable images will be available to DLSS. It performs the following steps:

1. Record the current `Camera.targetTexture`.
2. Set `Camera.targetTexture` and `RenderTexture.active` to the same texture: one that is managed by Upscaler and is suitable for use as a color target when upscaling.

ⓘ This step will not occur if 'None' is the active upscaler.

3.1.1.1.2 URP Upscaling Stage Two

The second stage is the execution of the `UpscalerRenderFeature`'s only `ScriptableRenderPass` (the `UpscalerRenderPass`). This render pass is configured to require `ScriptableRenderPassInput.Motion`. This means that it will force Unity to render motion vectors. This render pass is bound to the `RenderPassEvent.BeforeRenderingPostProcessing + 1` event. Motion vectors are rendered on the `RenderPassEvent.BeforeRenderingPostProcessing` event. This means that it will this stage will occur immediately after motion vectors are rendered.

The second stage is responsible for actually calling the upscaler with the required data. It performs the following steps to do so:

1. Set the color and depth render targets to textures controlled by Upscaler.
2. Copy the depth from the camera's depth texture to the active depth texture (using a custom blit function).
3. Blit the motion vector texture onto a texture controlled by Upscaler.
4. Upscale the color taking into account depth, motion, and current settings.
5. Blit the results of the upscaling step onto the original camera target (recorded in 3.1.1.1.1 URP Upscaling Stage One).

6. Set the `Camera.targetTexture` and the `RenderTexture.active` to the same texture: the original camera target (recorded in 3.1.1.1.1 URP Upscaling Stage One).

ⓘ This step will not occur if 'None' is the active upscaler.

3.1.1.2 Effects on Rendering When Using the BRP

The `RenderPipeline` object is actually an instance of the Builtin class. This class performs two jobs. First, it handles all requests for image resource updates from 3.1.2 The `Upscaler` Class. Secondly, It handles maintaining the two `CommandBuffer`s that are used to upscale and copy out the results. The `Builtin` object hooks into the `OnPreRender`, and `OnPostRender` `MonoBehaviour` events. It uses these events to perform each of its two stages. This will change the default behaviour of the BRP. Please read through this section carefully and evaluate these stages' impacts upon your current pipeline.

3.1.1.2.1 BRP Upscaling Stage One

Stage one is bound to `OnPreRender` and is responsible for preparing the upscaler for evaluation. It ensures that valid and suitable images will be available to DLSS by performing the following steps:

1. Record the current `Camera.targetTexture`.
2. Set the `Camera.targetTexture` and the `RenderTexture.active` to the same texture: one that is managed by Upscaler and is suitable for use as a color target when upscaling.

ⓘ This step will not occur if 'None' is the active upscaler.

3.1.1.2.2 BRP Upscaling Stage Two

Stage two is bound to `OnPostRender`. It is responsible for actually evaluating the upscaler. It performs the following steps:

1. Copy motion vectors to an Upscaler controlled texture.
2. Upscale the color taking into account depth, motion, and current settings.
3. Set the `Camera.targetTexture` and the `RenderTexture.active` to the same texture: the original camera target (recorded in 3.1.1.2.1 BRP Upscaling Stage One).
4. If the camera target was `null`

    1. Blit the results of the upscaling step onto `BuiltinRenderTextureType.CameraTarget`.
5. If the camera target was **not** `null`
    1. Copy the results of the upscaling step onto `BuiltinRenderTextureType.CameraTarget`.
6. Blit the rendered depth buffer onto the current camera depth buffer.

ⓘ This step will not occur if 'None' is the active upscaler.

3.1.2 The `Upscaler` Class

The `Upscaler` class handles interacting directly with the Unity camera itself. This includes:

1. Handling the camera's desired output resolution (based on the `width` and `height` properties of `Camera.targetTexture` if it exists or `Camera.pixelWidth` and `Camera.pixelHeight` failing that).
2. Handling the camera's HDR rendering requirements (based on Camera.allowHDR).
3. Handling the camera's MSAA rendering requirements (based on Camera.allowMSAA).
4. Integrating the usage of the `RenderPipeline` object into the camera's messages.

It also has duties that do not include the camera directly. For example:

1. Tracking current settings values in the GUI.
2. Detecting settings changes from both the API and GUI.
3. Determining when it is required for each of DLSS's image resources to be updated.
4. Determining when it is required that the `CommandBuffer`s be updated.

3.1.3 The `Jitter` Class

DLSS, like any good temporal antialiasing solution, requires that sub-pixel jittering be applied to the camera. This is handled by our static helper class `Jitter`. When the amount of upscaling to be applied to the image changes, the `Jitter` class constructs an array of jitter points based on the Halton Sequence. In accordance with DLSS's programming manual, the length of this array is dictated by the formula below where `upscalingFactor` represents the output resolution divided by the input resolution and is always greater than or equal to 1.

When jitter is actually being applied to the camera, the function `Camera.ResetProjectionMatrix` is called. The next Halton value in the sequence is then fetched from the array, transformed into the `(-0.5, 0.5)` range, and added to the camera's projection matrix as to offset the camera by that many pixels. Information about how the camera was just jittered is then sent to the C++ library.

⚠ If you call `Camera.ResetProjectionMatrix`, or you modify the camera's projection matrix manually, you may interfere with the expected operation of Upscaler.

3.2 Changing Settings

Upscaler allows you to change the settings through C# scripts using its API. Below is an example script that shows how one can get the Upscaler object, then use the API it provides to change upscaling settings on-the-fly.

Obtain an instance of the Upscaler class from the main camera. It is highly recommended that you ensure that the main camera actually exists. The example below works, but is not error protected at all. Use it with caution.

```
Upscaler.Upscaler upscaler = Camera.main!.GetComponent<Upscaler.Upscaler>();
```

Use `Upscaler.upscalingMode` to change the upscaler that is being used.

```
upscaler.upscalerMode = Upscaler.Upscaler.UpscalerMode.DLSS;
```

Use `Upscaler.qualityMode` to select the quality that that upscaler should use.

```
upscaler.qualityMode = Upscaler.Upscaler.QualityMode.Auto;
```

When you are in the Unity Editor, these values will change with the Editor UI.

3.3 Handling Errors

Errors can occur when requesting a settings change. By default, when this happens the default error handler will reset the current upscaler to "None". This will remove the `Upscaler` plugin from the pipeline ensuring that `Upscaler` does not cause any rendering errors. If this is not the behaviour that you want, you can choose to override the error handler. You may want to instead revert back to the last settings that worked, print the error to the console, or something else.

$$1 \leq upscalingFactor = \frac{outputResolution}{inputResolution}$$

$$Length = upscalingFactor.x \cdot upscalingFactor.y \cdot 8$$

First, to override the error handler replace the empty lambda with the function that you wish to respond to errors in the code below.

```
Upscaler.Upscaler upscaler = Camera.main!.GetComponent<Upscaler.Upscaler>();
upscaler.ErrorCallback = (status, message) => { };
```

Next, if you wish to perform any settings changes within your error handler, refer to section 3.2 Changing Settings. If you do not perform any settings changes then the active upscaler will be set to “None” to prevent fatal errors.

3.4 API Reference

- Include documentation for every function, class, enum, etc. that can be used by users
- Include code blocks and images where necessary.

4 Things to be Aware of

The following is a list of advice both from Conifer and from NVIDIA as given in their DLSS developer guide.

1. Disable all forms of anti-aliasing including but not limited to MSAA, TAA, FXAA, and SMAA.
2. Disable dynamic resolution. Upscaler does not support dynamic resolution.
3. Change the texture mip level bias when changing the quality mode using the formula below.

$$mipLevelBias = nativeBias + \log_2\left(\frac{renderResolution.x}{outputResolution.x}\right) - 1.0$$

4. Use DLSS for rendering directly to the screen OR rendering to an image that will be post processed before presented directly to the screen. Do NOT use DLSS to render things such as:
    1. The view through portals.
    2. The view of a security camera onto a monitor.

3. To upscale textures.
4. To render shadows.
5. To render a depth pass.
6. To render a reflection pass.

Or any number of possible non direct-to-screen views.

5. The Upscaler plugin is incompatible with RenderDoc. While this is unavoidable due to RenderDoc being incompatible with DLSS itself (as of RenderDoc v1.31 and likely long into the future), an alternative is available. Use NVIDIA's Nsight Graphics for graphics debugging. You can connect to Unity's Editor by launching it from Nsight Graphics. Upscaler will be able to load successfully, and will not interfere with any captures. Nsight Graphics is available for both Windows and Linux and was used to help create this plugin.

   This incompatibility is not something that Conifer plans on working to fix.
6. Read the DLSS developer guide for more information.

5 GUI Integration

Consult the DLSS UI Developer Guidelines PDF for more information about integration with in-game GUIs.

Upscaler breaks these guidelines when 'Auto' is the selected quality mode. NVIDIA recommends disabling DLSS at resolutions lower than 1920x1080. Upscaler's 'Auto' mode will select the 'Quality' mode in this case. This was done because we believe that it is important to respect user's options. If you want DLSS to disable itself at lower resolutions to follow NVIDIA's guidelines you must implement this yourself when integrating Upscaler with your GUI.

This document will be kept up-to-date with the latest releases of Upscaler. The document version number will be the same as the Upscaler version number. If any errors in this documentation are found please report them to <CONTACT INFORMATION>. If you have any further questions, please reach out to <CONTACT INFORMATION>.