DLSS Upscaling for Unity Documentation

About DLSS

DLSS is an upscaling technique, creating high quality and resolution frames based on lower resolution input. By using this, projects could have drastically lower GPU requirements than without.

Only if your project is limited by GPU performance, DLSS will gain you a higher framerate. If a project is limited by CPU performance, all it will do is make the GPU workload lower. While this may seem like a big limitation, it also means a laptop will use way less battery power when using DLSS!

Current supported Unity Render Pipelines:

Built-in Render Pipeline (BIRP) and Universal Render Pipeline (URP).
High-Definition Render Pipeline (HDRP) already has a native DLSS version.

Current supported platforms*:

- Windows x64 (DX11 and DX12)
- PCVR (BIRP & URP)

**DLSS only works on NVIDIA RTX cards.*

Working on:

- PCVR (HDRP)

The Naked Dev Tools

Upscaling Tools

Each and every upscaler out there has different strong and weak spots, some require specific hardware, some are specifically made for slow or older devices.

Here is the list of all our other upscalers for Unity:

FSR 3 - Upscaling for Unity

FSR 3 is supported on almost every platform and most hardware! Making it the number #1 goto upscaler. However compared with DLSS or XeSS the visual fidelity of FSR 3 is considered to be slightly lower.

XeSS - Upscaling for Unity

XeSS is limited to Windows x64, DX11 and DX12, but works just as well on Intel, AMD and Nvidia GPUs! XeSS's visual fidelity is higher than FSR 3 and in some ways even better than DLSS.

DLSS - Upscaling for Unity

DLSS is limited to Windows x64, DX11 and DX12, and will only work on Nvidia RTX GPUs (20x0 series and up). DLSS's visual fidelity is higher than FSR 3 and in some ways better than XeSS.

SGSR 1 - Upscaling for Unity

SGSR 1 is supported on almost every platform and most hardware! However SGSR 1's visual fidelity is a lot lower than the other upscalers. But it's also way faster than all other upscalers, making it perfect to use on platforms with a high DPI like mobile devices!

SGSR 2 - Upscaling for Unity

SGSR 2 is supported on almost every platform and most hardware! SGSR 2 offers a much higher visual fidelity as SGSR 1 and only costs a fraction more gpu power than SGSR 1. Additionally, SGSR 2 has a fragment version, which allows it to run on ever older hardware, making it perfect to use on mobile devices like Android, iOS and the Nintendo Switch!

Fallback Setup

For the absolute best visual upscaling results we recommend using the following fallback format:

1. DLSS
2. XeSS
3. FSR 3
4. SGSR 2
5. SGSR 1
6. FSR 1

Various Tools

[CACAO - Ambient Occlusion for Unity]

CACAO is an ambient occlusion technique, created for AAA games, to produce the best possible ambient occlusion visuals while also offering the best performance possible.

Quick Start

This chapter is written to add DLSS as fast as possible to your project. However, it is very much recommended to read the Important Information chapter on current DLSS limitations.

Quick Start: BIRP

Step 1: Import the DLSS Package in your project.

Step 2: Import our custom Post-Processing package here and place a Post-process Layer on your main camera then enable DLSS in the Anti-Aliasing settings on the Post-Process Layer.

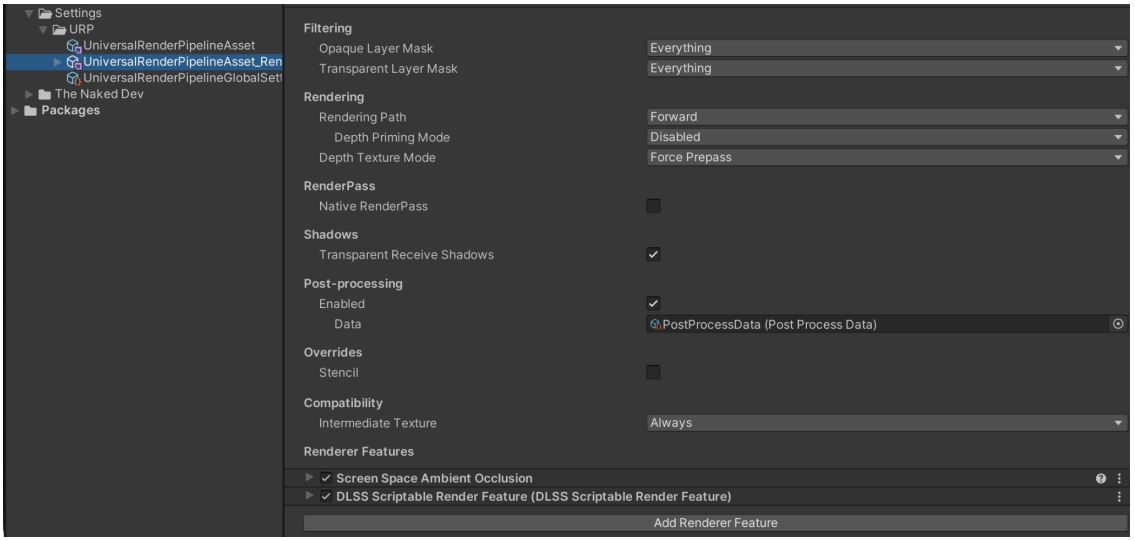
Hit play!

Quick Start: URP

Step 1: Import the DLSS Package in your project.

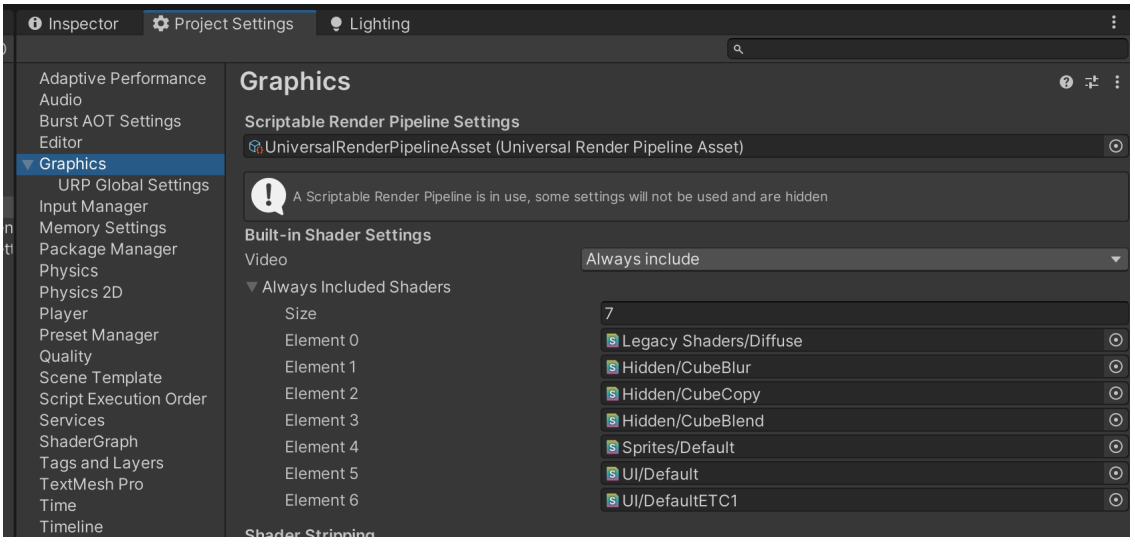
Step 2: Add DLSS_URP.cs script to your main camera.

Step 3: Add the “DLSS Scriptable Render Feature” to your Universal Renderer Data files. (Add it to all the URP data files you will use in your project, for example if you use different ones per quality setting).



Hit play!

Note: If you can’t add the DLSS_URP component to your Main Camera, make sure you have a Scriptable Render File in the Scriptable Render Pipeline Settings, DLSS uses this to check which render pipeline is active

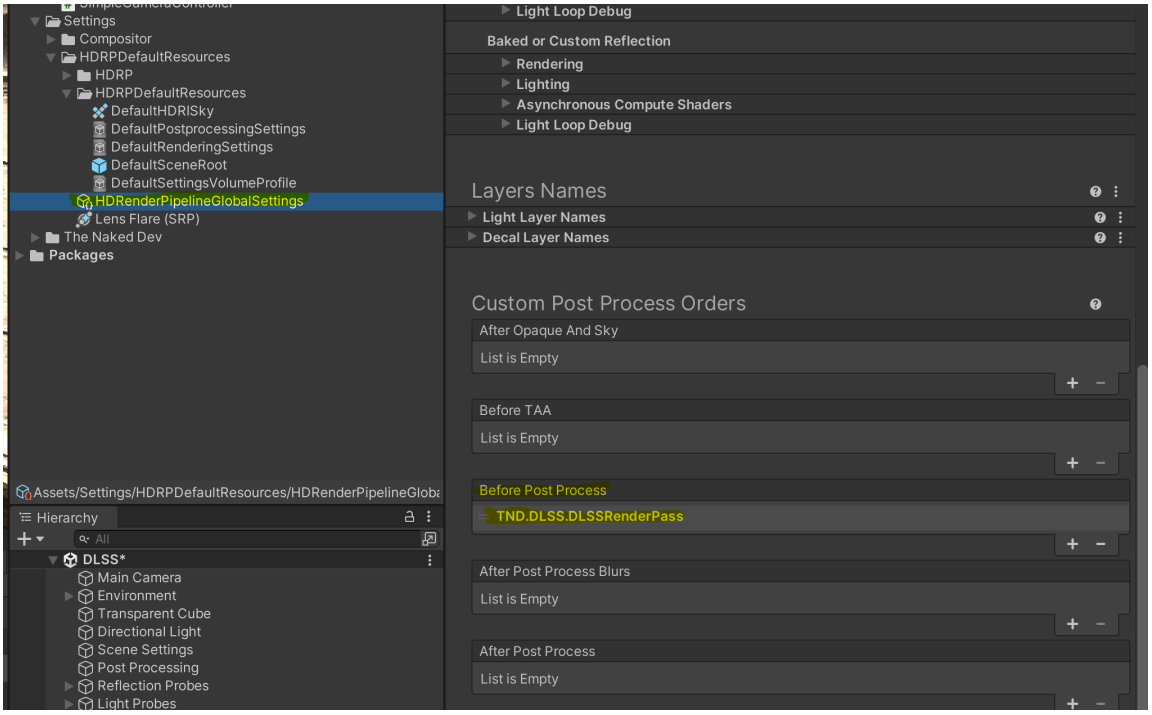


Quick Start: HDRP

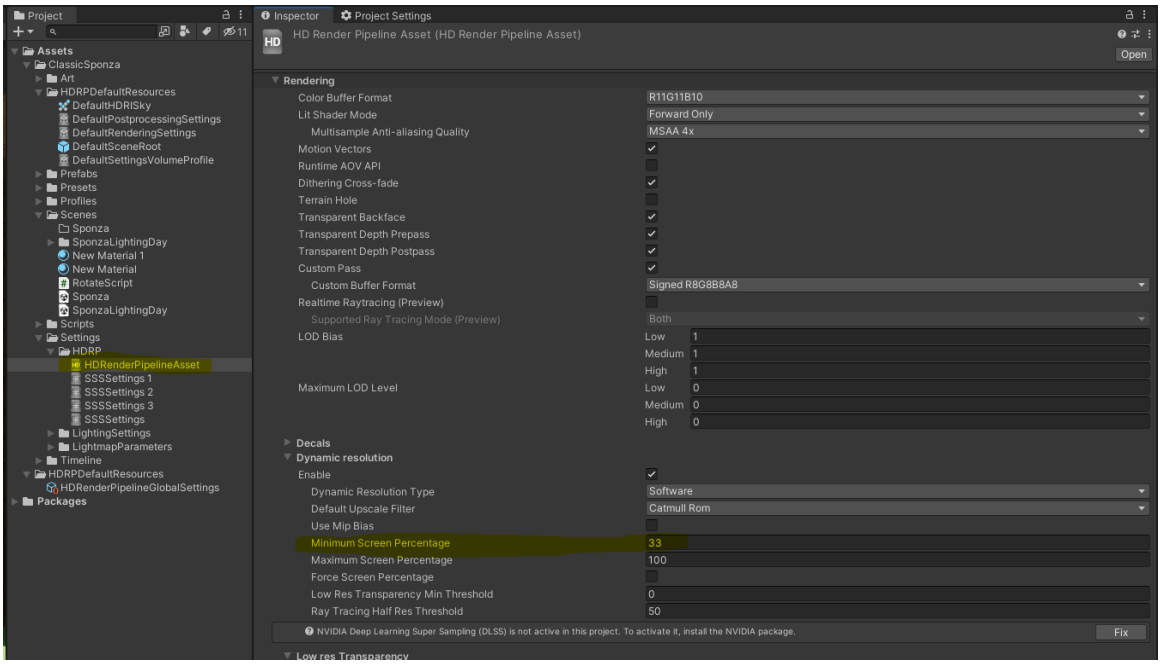
Step 1: Import the DLSS Package in your project.

Step 2: Edit the HDRP source files, seen chapter HDRP source files

Step 3: Add the TND_DLSS_DLSSRenderPass to the HDRenderPipelineGlobalSettings in “Before Post Process”.



Step 4: Enable Dynamic Resolution in HDRP settings and set the “Minimum Screen Percentage to 33. Keep the Default Upscale Filter to Catmull Rom. NOTE: if you use multiple RenderPipeline Asset files for different Quality settings, make sure you edit them all!



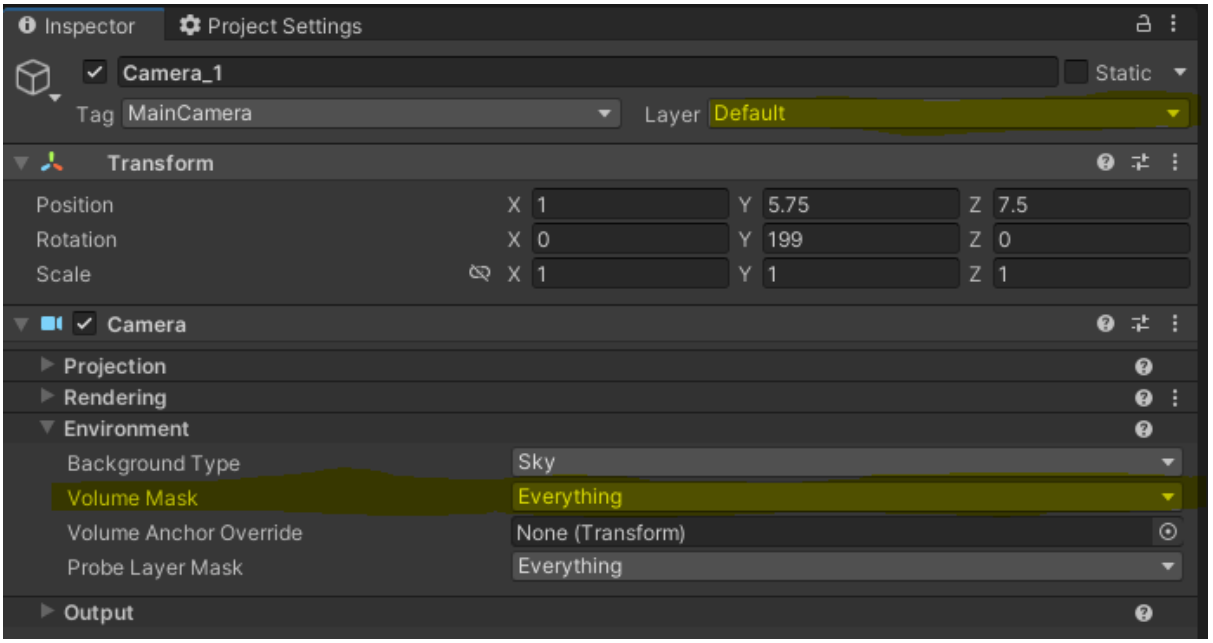
Step 5: Add DLSS_HDRP.cs script to your main camera.

Step 6: Press the “I have edited the source files” button.
Note: If you get compilation errors after you press the button, you have not properly edited the source files.

Hit play!

The next steps are optional, because DLSS will do these steps for you if you skip them.

Optional Step 6: Make sure the “Volume Mask” of your camera includes the layer of your camera. If you skip this step, DLSS will automatically do this for you.



Optional Step 7: Enable “Allow Dynamic Resolution” on the Camera. If you skip this step, DLSS will automatically do this for you. (DLSS will also automatically disable Unity’s DLSS if enabled”

Demo Scenes

With the ChangeQuality.cs script you can toggle between quality modes by pressing the spacebar.

Download Demo Projects here

Note, the HDRP demo has been created in Unity 2021.3.33 LTS, and HDRP 12, the demo project might break on older or newer Unity versions because of the changes in the HDRP source.

Important Information

Multiple Cameras

DLSS Upscaling for Unity doesn't currently support multiple cameras using DLSS, it may even make Unity crash if you try. Generally it is best practice to limit the use of multiple cameras at the same time due to degrading performance.

Temporal Anti-Aliasing (TAA)

DLSS has a very good built-in AA solution. This internal AA is not optional. There is no need to add any other AA to the camera, since this will only add more blur and ghosting. Adding different types of Anti-Aliasing will also decrease the upscaling quality.

Mipmaps - BIRP Only

When DLSS downscales the source rendering, it is recommended to add a negative mipmap bias to all textures in your scene for improved quality. Textures will look very washed out otherwise. In the Inspector chapter, we explain how we created a way to update all mipmap biases on the textures automatically.

Post Processing Effects (BIRP)

For every upscaling technique, is it very important to know how to use Post Processing effects. Some effects will need to be added before DLSS, others afterwards. Check out our demo's for examples.

To make this happen in the most efficient way we have edited a version of Unity's Post Processing, which you can download here, that fully inserts DLSS to the Post Processing layer allowing it to support DLSS as is intended.

Inspector

Quality

Off: DLSS is off
Native AA: 1.0x scaling (Native, AA only!)
Ultra Quality: 1.2x scaling
Quality: 1.5x scaling
Balanced: 1.7x scaling
Performance: 2.0x scaling
Ultra Performance: 3.0x scaling

Example: If the native resolution is 1920x1080, selecting the **Performance** option will change the rendering resolution to 960x540, which will then be upscaled back to 1920x1080. Changing the quality setting will update the “m_scaleFactor” variable.

FallBack - BIRP only

The current Anti-Aliasing will be changed to the fallback option when DLSS is not supported.

Anti-Ghosting

0 - 1

All Temporal Anti-Aliasing solutions add some ghosting, this can be minimised by using the Anti-Ghosting slider.

Auto Texture Update - BIRP only

Off - On

As previously explained, it is recommended to update the MipMap Bias of all used textures. In Unity, the only way to do this is by script, texture by texture. This is less than ideal, so we added a feature to automatically update all textures currently loaded in memory. In real-world projects, we saw no noticeably extra CPU cost.
Note: It seems URP and HDRP already automatically do mipmap biassing, so here we disabled this feature by default.

Mip Map Update Frequency

0.0 - Infinite

This settings determines how often the Auto Texture Update checks for new loaded textures to update the Mipmap Bias for.

Mipmap Bias Override - BIRP only

0.0 - 1.0

When using DLSS, and changing the MipMap bias, it is possible that there will be additional texture flickering. If you notice too much texture flickering, try lowering this setting until you have the desired balance between quality and texture stability. If you have no texture flickering, keep this to 1 for best quality.

Public API

When using the DLSS_URP.cs or DLSS_HDRP.cs camera components, you can call the following API functions on those components. When using the custom PostProcessing package for BIRP, you can change the values of the Post-processing Layer just like you'd normally would when changing settings on it.

Generic

public bool OnIsSupported()
Use this function to check if DLSS is supported on the current hardware. It's recommended to use this function, before enabling DLSS .

public bool OnSetQuality(DLSSQuality value)
Use this function to change the DLSS quality settings.

BIRP Custom Post Processing Layer

To change any of the settings of the Post-process Layer you'll need a reference to it, afterwards you can address the upscaler settings like this:

```
using TND.DLSS;

public class ChangeQuality : MonoBehaviour
{
    void Start()
    {
        _postProcessingLayer.dlss.qualityMode = DLSS_Quality.Quality;
    }
}
```

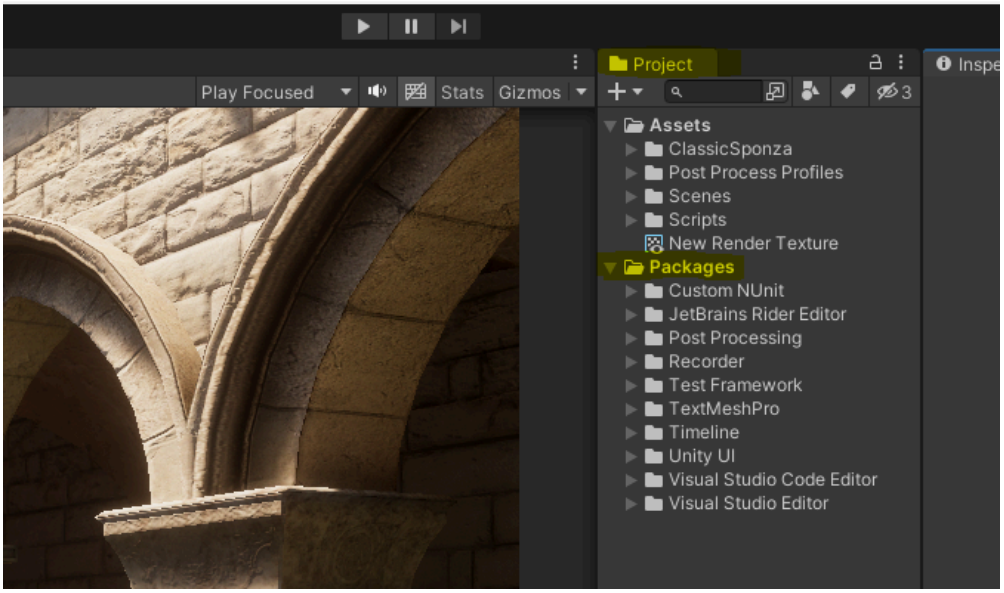
Editing Unity Post Processing package

Download

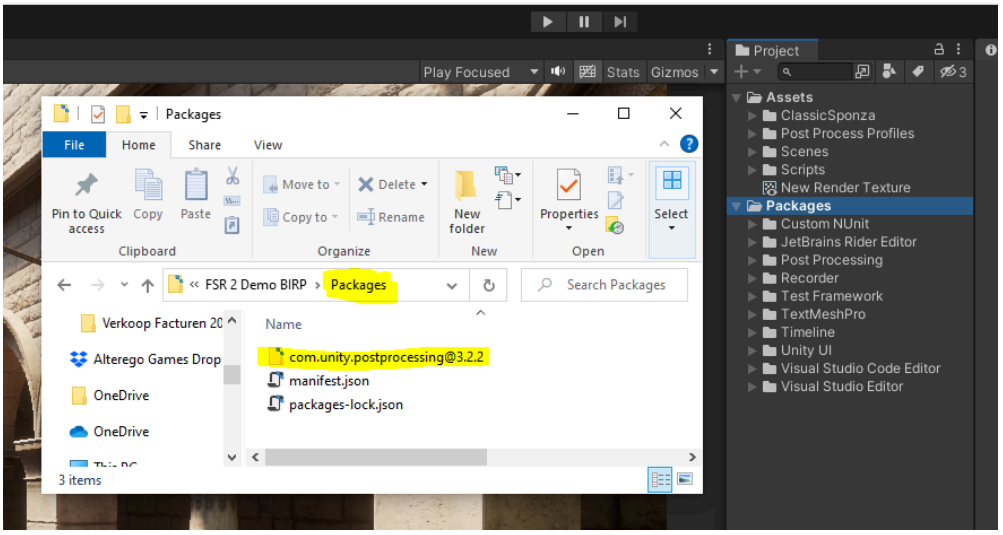
First of all, download our edited Unity Post Processing package:
[Download Link](#) or add https://github.com/DominicdeGraaf/Unity-Post-Processing-Stack.git to the Unity Package Manager

Use this edited package only when you're planning to use BIRP and want to be able to use [Unity's Post Processing Stack V2.](#)

Step 1: Locate the Packages folder in your project, press Right-Mouse Button on it and select "Show in Explorer"



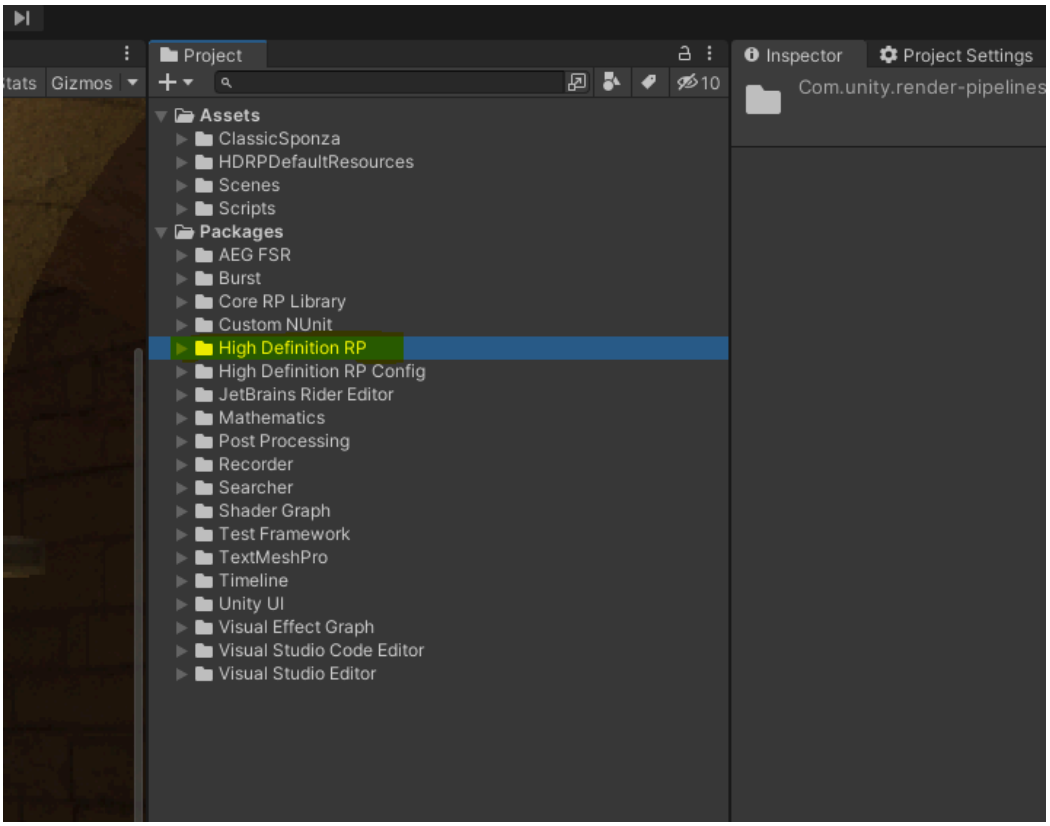
Step 2: Unzip the downloaded package into the Packages folder



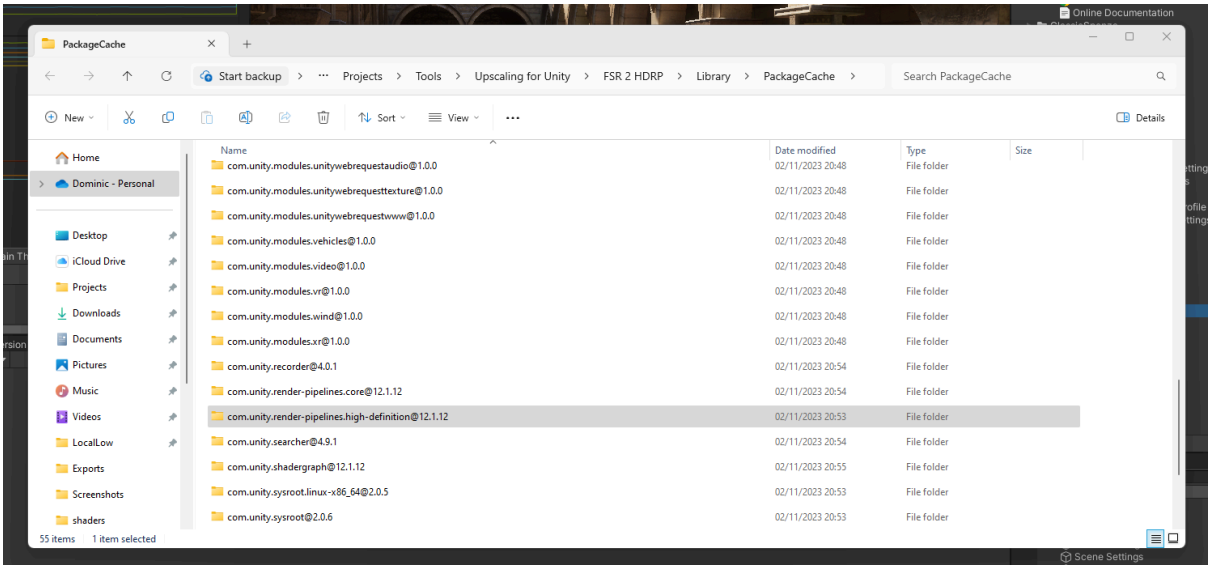
All done, now you should be able to use the single camera setup in BIRP.

Editing Render Pipeline source files

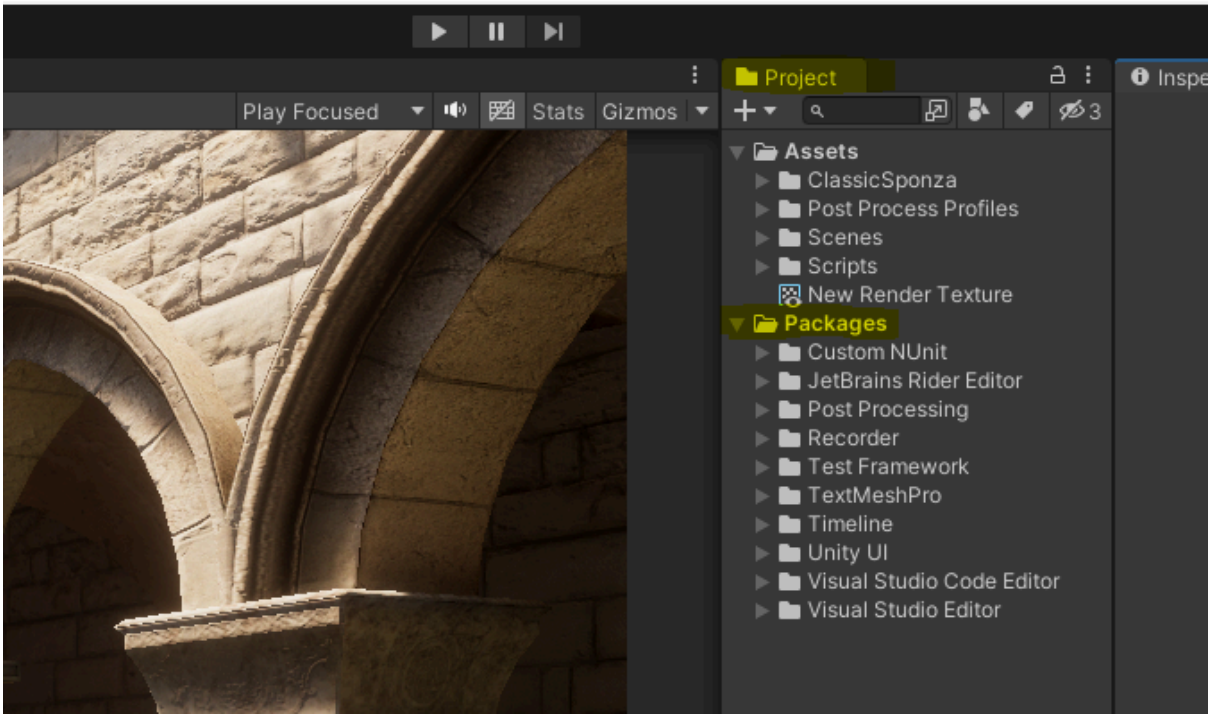
Step 1: Locate the Packages folder in your project, unfold it and press Right-Mouse Button on the “High Definition RP” and select “Show in Explorer”



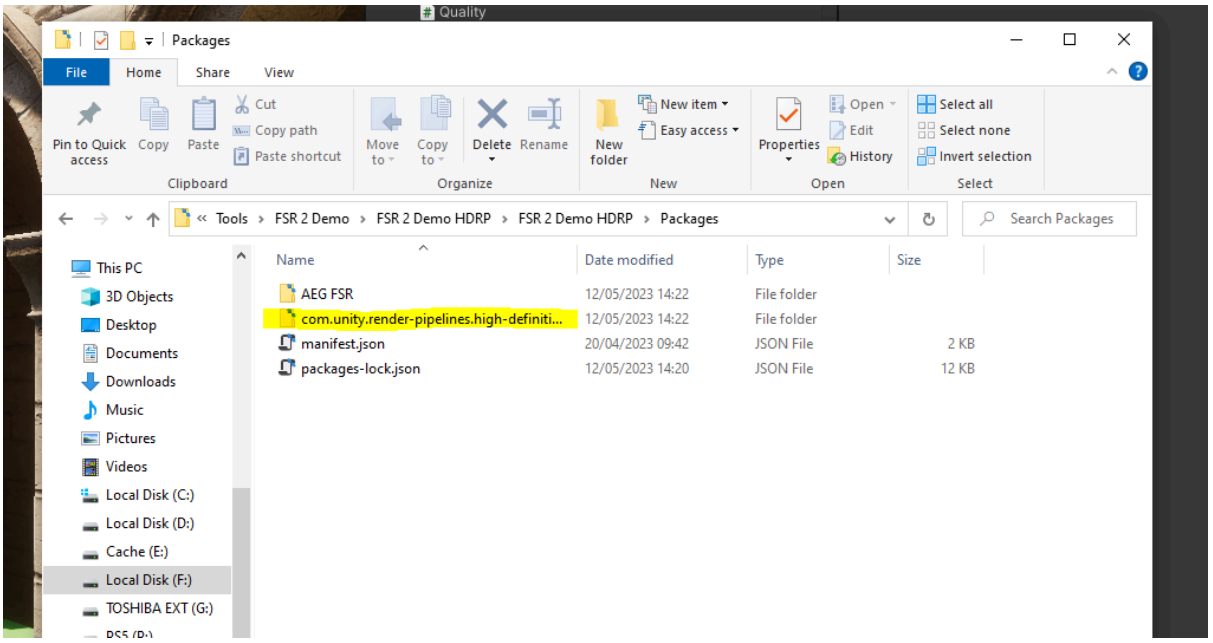
Step 2: Copy the com.unity.render-pipelines.high-definition@xx.x.xx



Step 3: Locate the Packages folder in your project, press Right-Mouse Button on it and select “Show in Explorer”



Step 4: And paste the previously copied folder into this Packages folder!



Now the source files of the High-Definition Render Pipeline are editable!

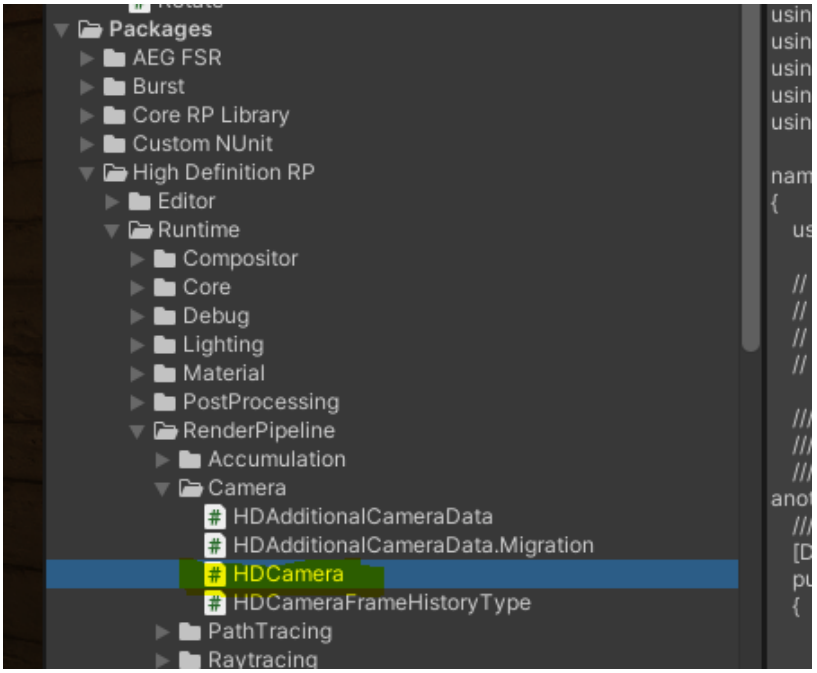
HDRP source files

To make FSR 3 work in HDRP we need to edit two files. If you are unsure on how to edit Render Pipeline Source files, read about it here.

NOTE: If you're already using one of our other upscalers for HDRP and already changed the source files, you don't have to do it again!

NOTE 2: Please take note that different HDRP versions require different edits

HDCamera.cs



UNITY 2022.3 LTS or older (with HDRP 13.X.X)

In the script, locate:

```
internal bool UpsampleHappensBeforePost()
{
    return IsDLSSEnabled() || IsTAAUEnabled();
}
```

And replace the whole function with:

```
public bool tndUpscalerEnabled = false;

internal bool IsTNDUpscalerEnabled()
{
    return tndUpscalerEnabled;
}

internal bool UpsampleHappensBeforePost()
{
    return IsTNDUpscalerEnabled() || IsDLSSEnabled() || IsTAAUEnabled();
}
```

UNITY 2023.1 or newer (and certain 2022.3 with HDRP 14.0.x)

In the script, locate:

```
internal DynamicResolutionHandler.UpsamplerScheduleType
UpsampleSyncPoint()
{
   if (IsDLSSEnabled())
   {
      return
HDRenderPipeline.currentAsset.currentPlatformRenderPipelineSettings.dyna
micResolutionSettings.DLSSInjectionPoint;
   } else if (IsTAAUEnabled())
   {
       return DynamicResolutionHandler.UpsamplerScheduleType.BeforePost;
   } else
   {
       return DynamicResolutionHandler.UpsamplerScheduleType.AfterPost;
   }
}
```

And replace the whole function with:

```
public bool tndUpscalerEnabled = false;

internal bool IsTNDUpscalerEnabled() {
    return tndUpscalerEnabled;
}

internal DynamicResolutionHandler.UpsamplerScheduleType
UpsampleSyncPoint() {
   if(IsDLSSEnabled()) {
         return
HDRenderPipeline.currentAsset.currentPlatformRenderPipelineSettings.dyna
micResolutionSettings.DLSSInjectionPoint;
   } else if(IsTAAUEnabled()) {
         return
DynamicResolutionHandler.UpsamplerScheduleType.BeforePost;
   } else if(IsTNDUpscalerEnabled()) {
         return
DynamicResolutionHandler.UpsamplerScheduleType.BeforePost;
   } else {
         return
DynamicResolutionHandler.UpsamplerScheduleType.AfterPost;
  }
}
```

UNITY 6 or newer (with HDRP 17.0.x)

In the script, locate the function:

```
internal DynamicResolutionHandler.UpsamplerScheduleType
UpsampleSyncPoint()
```

Add “ || IsTNDUpscalerEnabled() “ to the IsFSR2Enabled if statement.

```
 if(IsFSR2Enabled() || IsTNDUpscalerEnabled()) {
         Return
HDRenderPipeline.currentAsset.currentPlatformRenderPipelineSettings.dyna
micResolutionSettings.FSR2InjectionPoint;
   }
```

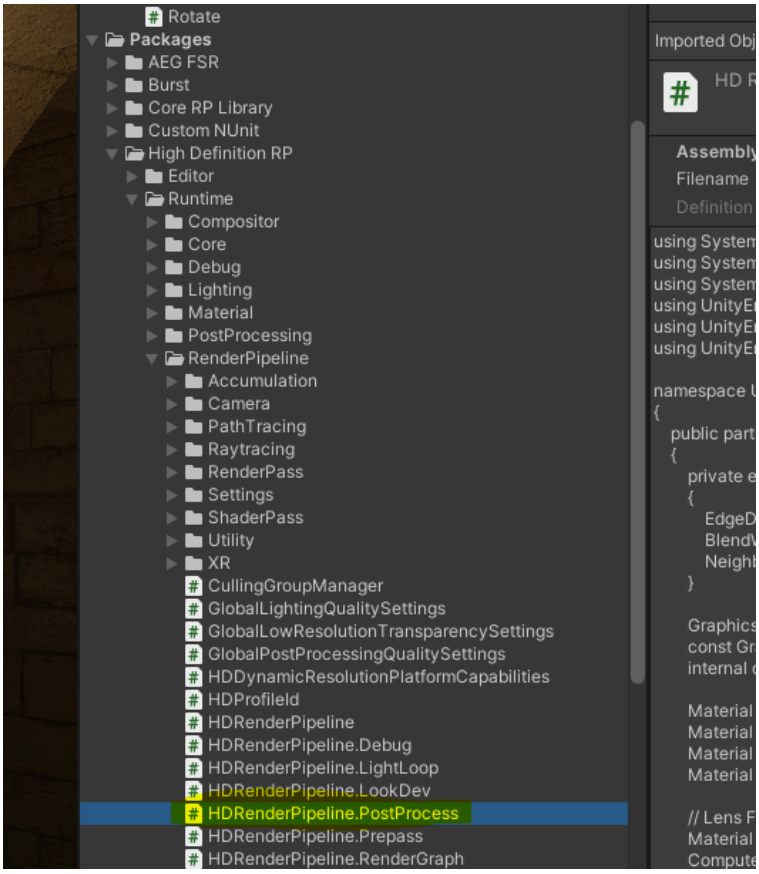
And add the following lines just above the function

```
public bool tndUpscalerEnabled = false;

internal bool IsTNDUpscalerEnabled() {
    return tndUpscalerEnabled;
}

internal DynamicResolutionHandler.UpsamplerScheduleType
UpsampleSyncPoint() {
    …
    …
    …
    …
    …
    …
    …
    …
```

HDRenderPipeline.PostProcess



All versions pre Unity 6 (with HDRP 15.X.X) or older

In the script locate:

```
source = CustomPostProcessPass(renderGraph, hdCamera, source,
depthBuffer, normalBuffer, motionVectors,
m_GlobalSettings.beforePostProcessCustomPostProcesses,
HDProfileId.CustomPostProcessBeforePP);
```

And replace the line with:

```
source = CustomPostProcessPass(renderGraph, hdCamera, source,
depthBuffer, normalBuffer, motionVectors,
m_GlobalSettings.beforePostProcessCustomPostProcesses,
HDProfileId.CustomPostProcessBeforePP);
if (hdCamera.IsTNDUpscalerEnabled())
{
    SetCurrentResolutionGroup(renderGraph, hdCamera,
ResolutionGroup.AfterDynamicResUpscale);
}
```

UNITY 6 or newer (with HDRP 17.0.x)

In the script locate:

```
source = BeforeCustomPostProcessPass(renderGraph, hdCamera, source,
depthBuffer, normalBuffer, motionVectors,
m_CustomPostProcessOrdersSettings.beforePostProcessCustomPostProcesses,
HDProfileId.CustomPostProcessBeforePP);
```

And replace the line with:

```
source = BeforeCustomPostProcessPass(renderGraph, hdCamera, source,
depthBuffer, normalBuffer, motionVectors,
m_CustomPostProcessOrdersSettings.beforePostProcessCustomPostProcesses,
HDProfileId.CustomPostProcessBeforePP);
if (hdCamera.IsTNDUpscalerEnabled())
{
    SetCurrentResolutionGroup(renderGraph, hdCamera,
ResolutionGroup.AfterDynamicResUpscale);
}
```

VR

The latest update of DLSS - Upscaling for mobile has added support for PCVR for BIRP and URP. If you do run into any issues, please get in contact!

Current Limitations:

For now only Multi-pass is supported. Single-pass support is still under investigation.

FAQ

Q: Does DLSS offer free performance?
A: Almost, in many cases it does. However DLSSdoes use a small bit of GPU performance, so when using Quality mode the gains can be little. However if your project is CPU bound, you will not likely see much performance gains, just a lower GPU usage.

Q: Will DLSS work for every kind of project?
A: No, in projects that are CPU bound, DLSS will probably only hinder performance. However because DLSS includes an industry leading AAA quality Anti-Aliasing, your project's fidelity might improve over Unity's standard anti-aliasing.

Q: DLSS is not working or I am having an issue, help!
A: If you encounter any issues, you can contact us by emailing to info@thenakeddev.com, joining our discord in the "Unity Tools" channel or on the Unity Forum.

Q: Can I use older/newer DLSS versions?
A: Yes you can! The nvngx_dlss.dll file located in the editor installation can be replaced with any DLSS version downloadable from here:
https://www.techpowerup.com/download/nvidia-dlss-dll/
Updating this .dll file may also fix any black screens that you're getting.

Q: Where is the Frame Generation?
A: I'm looking into the possibilities of adding DLSS 3 Frame Generation feature into Unity.

Known Issues & Limitations

General

- Sometimes DLSS will provide a black screen. This is a known issue and is fixed by updating the nvngx_dlss.dll file in the Unity editor installation folder as described here.

BIRP

-

URP

- Ideally most Post Processing is rendered before DLSS, due to the nature of each different URP version, this is not possible as of yet. In future updates this might be added for better quality Post Processing.

HDRP

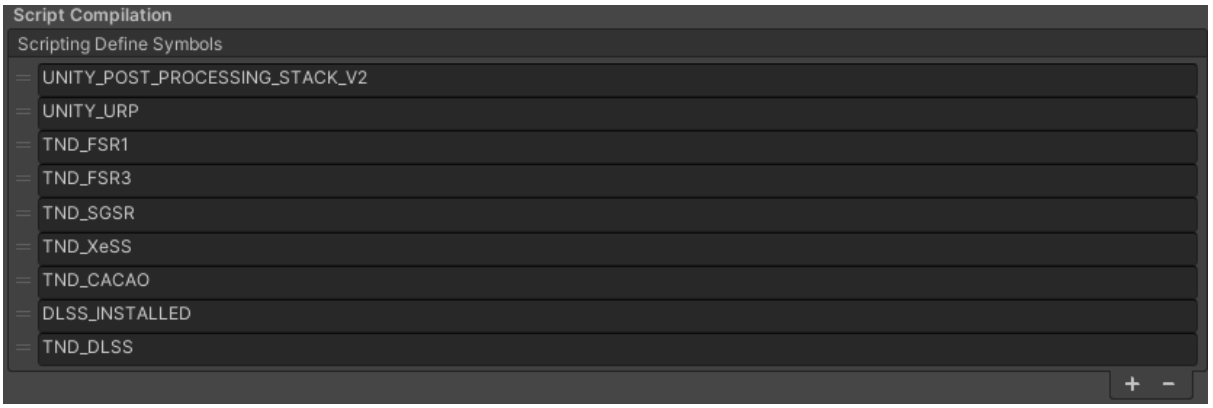
- Sometimes to make DLSS work correctly, the “Dynamic Resolution Type” in the HDRenderPipelineAsest file should be set to “Software” instead of “Hardware”
- Screen Space Reflections can show some artefacts on PBR settings.

Uninstall

Step 1: Delete the “DLSS” folder in“The Naked Dev” Folder

Optional Step 2 (BIRP ONLY): Delete the Custom Post-Processing Package folder from the “Packages” folder

Optional Step 3: If you are still getting compile errors after deleting the asset, go to the Scripting Define Symbols in the Player settings and manually delete the define symbols starting with “**TND_**”, “**DLSS_INSTALLED**” and “**UNITY_URP/UNITY_BIRP/UNITY_HDRP**”.



If you are still getting compile errors, make sure you are not referencing the upscaling asset in a component of your own.

Support

If you encounter any issues, you can contact us by emailing to info@thenakeddev.com, joining our discord in the “Unity Tools” channel or on the Unity Forum.