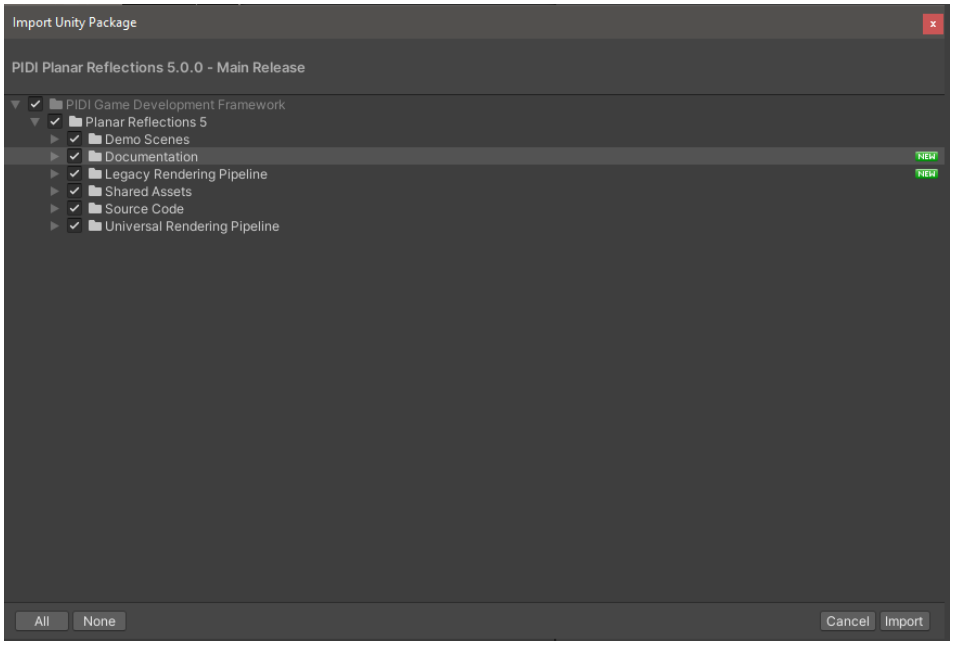
Thank you for buying PIDI Planar Reflections 5. This offline version of the documentation is provided for your convenience, but you are heavily encouraged to read the online version instead as it contains the most up-to-date information about the product, its features and capabilities.

Getting Started

Installation

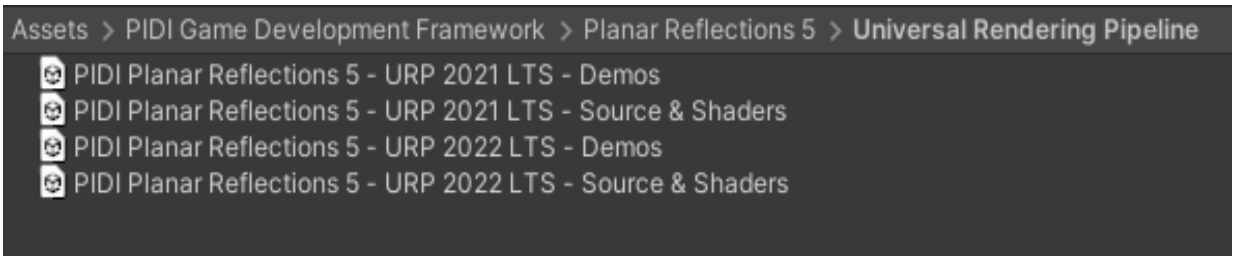
To install the asset, open the Package Manager and from the drop-down menu select My Assets. Then, search for PIDI Planar Reflections 5. Download the asset and prepare to import it into your project. You will see the screen showing all the contents of the asset (in the following picture we show the contents of the Personal Edition):



If you are using the Universal RP edition of the asset, you can simply import all the contents of the package into your project as there is no content from other pipelines available. If you are using the Personal Edition, please read the following section.

Personal Edition and multiple rendering pipelines

If you are using the Personal Edition of the asset, remember to deselect all the folders designed for other pipelines to reduce the amount of content installed into your project and avoid any potential errors. Once you unpack the folder for the rendering pipeline you are using you can simply double click on the most recent unitypackage inside and this will automatically unpack and install all the corresponding files.



Always import the shaders and source code package first, followed by any additional resources you need, and finally install the demo packages. This is necessary to avoid any errors. Also remember that you can only have the contents for ONE pipeline installed at a time since the source code of the asset is different for each one of them.

Every time you update the asset to a newer version, simply import the most recent unitypackage within the folder that matches the rendering pipeline you are using. We recommend you back up older versions of the asset in case that you want to roll back any upgrade / update.

Upgrading from Version 4.x.x

If you are upgrading from Version 4.x.x of Planar Reflections, you just need to remove it entirely from your project before installing version 5.x.x. Once installed, version 5.x.x will, under normal circumstances, replace all references to the previous version without any issues.

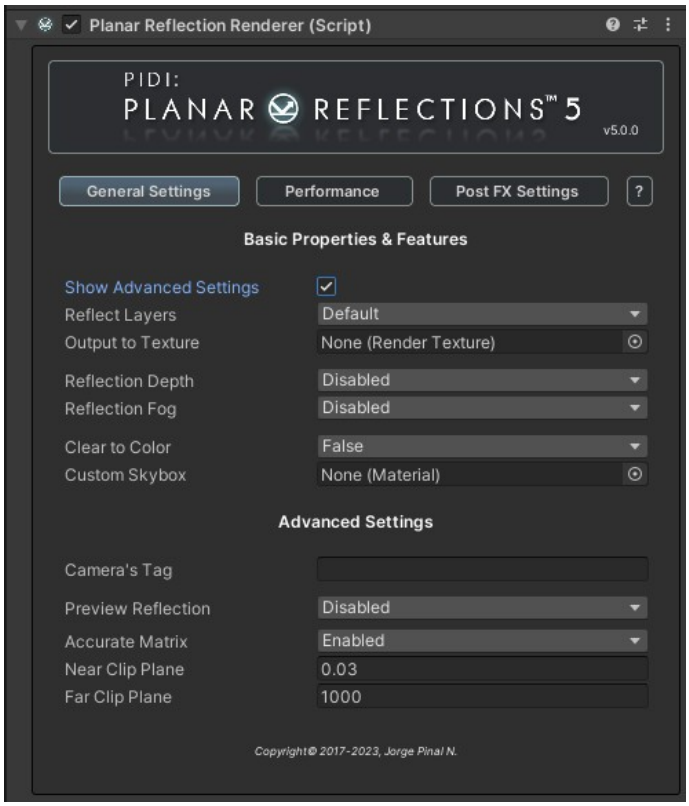
Planar Reflection Renderer

The Planar Reflection Renderer is the script responsible for rendering the real-time reflections and their corresponding depth pass.

General Settings

The General Settings tab contains the essential features and settings of a reflection renderer. The **Reflect Layers** drop out menu allows you to select which rendering layers will be rendered by the reflection. The optional **Output to Texture** and **Output Depth to Texture** slots allow you to specify custom RenderTextures to which the reflection and / or its depth will be rendered to. Here you can also specify whether the Depth and Fog pass of the reflection will be rendered as well as the way the background will be cleared, either to a custom color or to a Skybox.

If the **Show Advanced Settings** toggle is enabled, additional properties of the Reflection Renderer will become available.



A **Camera Tag** can be specified in order to filter out which cameras request a reflection rendering pass. This can be useful to reduce overhead and limit the amount of processing that the Reflection Renderer does, allowing it to ignore unnecessary cameras.

The **Preview Reflection** setting enables or disables a preview mesh that will show, in the Scene View, the way the reflection is being rendered internally. This preview is not visible in-game.

By default, reflectors should use an **Accurate Matrix**, which ensures that the objects behind the reflective surface are not displayed as part of the reflection. This is achieved by modifying the near and far planes of the reflection camera internally, but this may cause conflicts with some screen space effects and other view dependent systems. To solve this, you may need to disable the **Accurate Matrix**, but in turn this will introduce other issues.

While the **Near** and **Far clip planes** of the reflective camera are heavily modified, you can still use these values to limit the range in which the reflections will draw objects, which may help to improve performance in scenes with huge scenes to reflect, such as a lake reflecting a landscape.

Performance Settings

Different settings to fine tune performance are available within the Performance tab. Here you can decide whether shadows should be reflected, adjust the LOD bias for the reflections (so that lower quality meshes are rendered in mirrors and other reflective surfaces) as well as a custom limit for the framerate. While reducing the framerate of the reflections may make them look choppy, this can be a very efficient way to reduce performance costs while remaining unnoticeable if balanced carefully against the game's actual framerate.

The final resolution of the reflection can have a huge impact in performance, especially in scenes with complex shaders and visual effects. The resolution of the reflection can be set manually, or it can be based on the actual screen size the game is being rendered at, to ensure that the reflections keep a consistent quality at different screen resolutions.

Additionally, certain features applied to the output can help ensure a higher quality with a lower performance cost. Mip Maps paired with Anti-aliasing can allow for the reflections to be blurred with PBR enabled shaders as well as improve the look on lower resolution outputs, while HDR reflections will automatically be compatible with Bloom and other post process effects without the overhead of actually computing Post FX on the reflection itself.

Post FX Settings

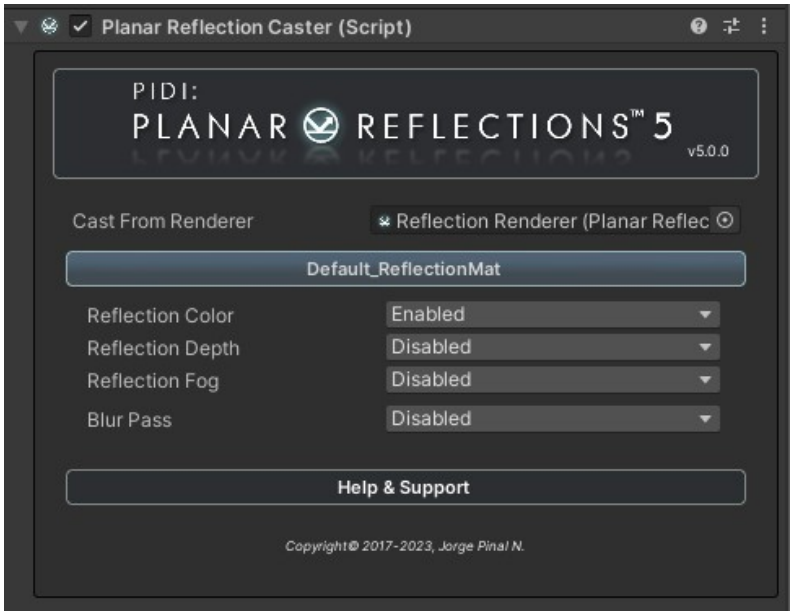
Each Reflection Renderer component can set up support for Post Process FX with their unique volume layer masks. While most Post Process FX will work without issues with the Reflection Renderer (including ambient occlusion, color grading, bloom, depth of field etc.) some of them, especially those that depend on motion vectors or screen-based UVs (such as vignetting) may not work nor look correctly.

Once Post Process FX are enabled in the Planar Reflection Renderer you can set up with which layer is it going to interact. It is not recommended to use the same post processing volume for your main in-game camera and the reflections as this could cause some overlapping of effects, for example if you apply color grading to your scene you may end up applying color grading to the reflection and then a second pass of the same color grading through the main camera. This applies as well to Tonemapping effects, being useful in some cases to use a HDR enabled reflection instead and let the main camera's tonemapping and bloom filters take care of the rest.

Help & Support

The last tab of the UI includes a small guide so that you can request support for our asset in a more efficient way, detailing steps to contact us and the information we require. It also contains a button that will take you to the latest version of this documentation file.

Planar Reflection Caster



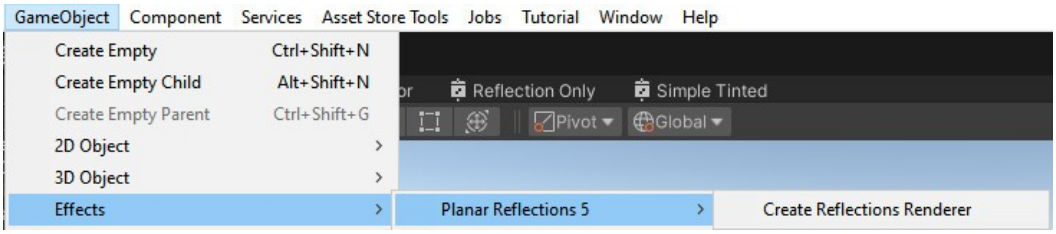
The **Planar Reflection Caster** component is the script that takes a reflection and assigns it to a material. It must always be attached to a Mesh Renderer component, and it will automatically track all of its materials and enable per-material settings to define whether the material uses the Reflection Color texture (**_ReflectionTex**), the Reflection Depth texture (**_ReflectionDepth**) and whether an additional Blur Pass should be used to blur the final output of the reflection. This blur pass should not be confused with the automatic blur applied through PBR shaders. Because of this, we recommend you disable the blur pass in most cases. It is provided as a form of backwards compatibility with shaders from previous versions.

The Reflection Fog pass can also be enabled from here, to simulate and closely match the fog present in the scene within the reflection itself.

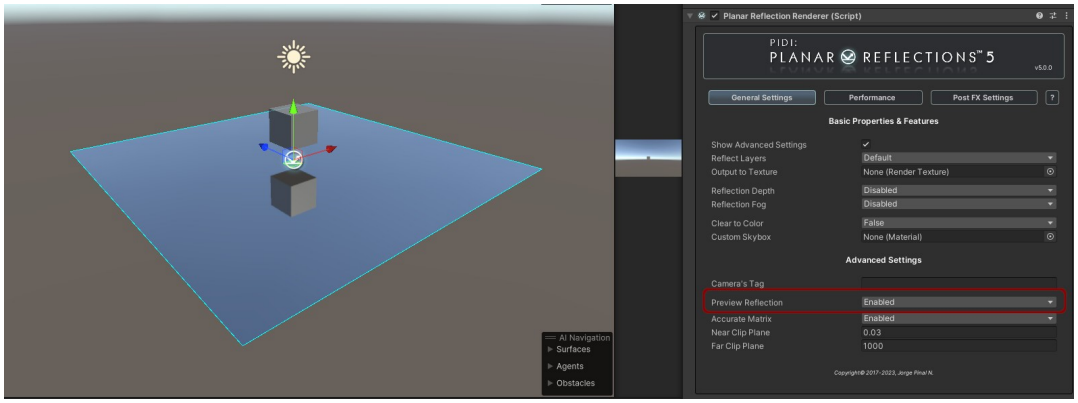
Adding Reflections to a scene

Adding reflections to a scene with PIDI Planar Reflections 5.x is a very simple process that takes only a few minutes. To simplify the workflow of adding reflections in real-time and ensuring that they are as re-usable as possible within a level the process has been split into two main components, the Reflection Renderer which generates a reflection and its depth pass and a Reflection Caster which assigns it to a material and actually displays it in the scene.

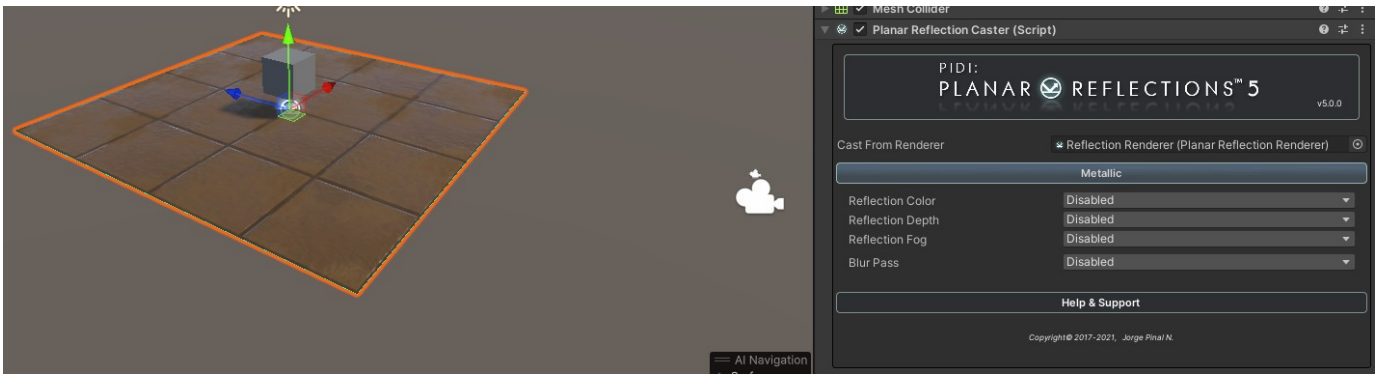
To add a Reflection Renderer to the scene go to the GameObject tab on the top of the Unity Editor, then to Effects, Planar Reflection 5, Create Reflections Renderer.



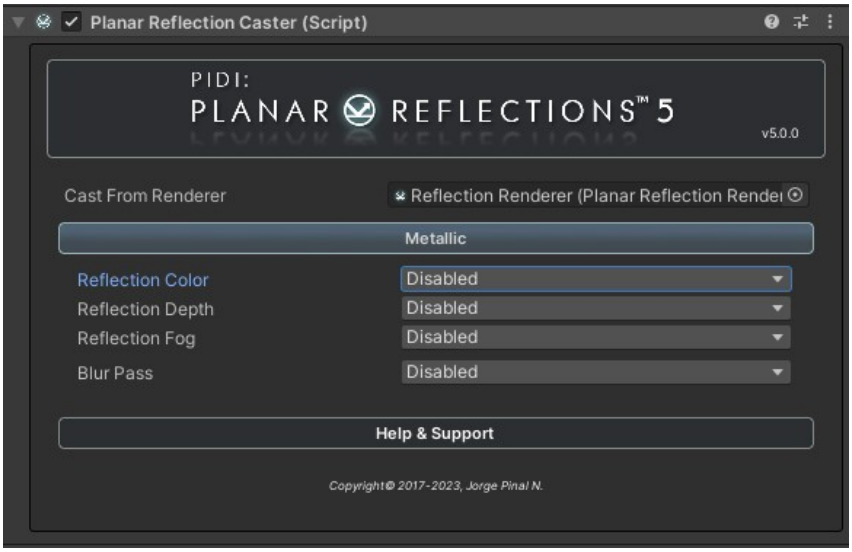
This will create an empty reflection renderer in the middle of the scene. By default, it will not display the preview reflection renderer but you can enable it easily on the Main Settings tab of the Reflection Renderer component.



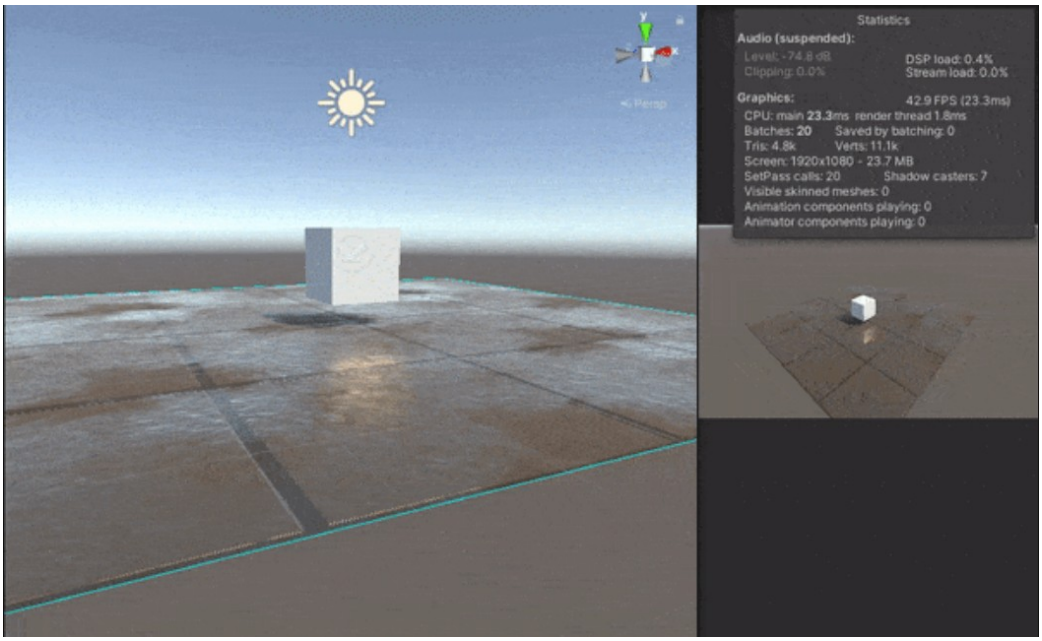
The reflection will not be visible at all on the Game View as it is not being displayed on any mesh at the moment. Let's add a plane to the scene and use one of the demo materials included with the asset on it. In our case, the PIDIPlanarReflections5_PBR_Metallic material (or Universal Rendering Pipeline/Shaders/PBR/Metallic for URP). We will also add a Planar Reflection Caster component to it.



Then, as the last step, assign your Reflection Renderer to the Cast From Renderer slot and enable the Reflection Color feature in order to display the Reflection Color texture over the surface.



With this, the reflection is ready and will be displayed without issues in both the Game and Scene views*. With Planar Reflections 5 you can also use our accurate roughness / smoothness based blur out of the box, integrated within the shader itself, without needing to enable the Blur Pass in the Reflection Caster component and with better performance.



***The Scene View presents a slight delay and may not work if the Game View is not visible when using Unity 2022.3 LTS at release, due to changes in the internal Unity APIs. This does not affect in any way the functionality in the Game View nor at runtime.**

Advanced Topics

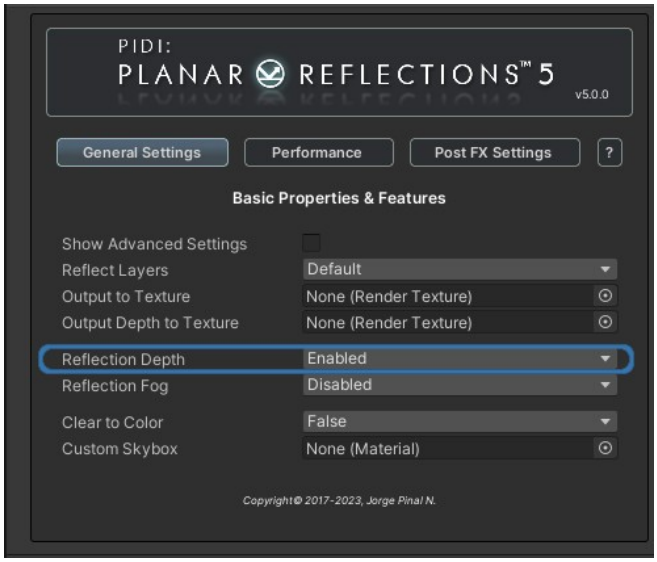
In this section we will cover some advanced uses of the PIDI Planar Reflections 5.x asset, including how to create custom shaders that support the reflections generated by the tool using ShaderLab and ShaderGraph as well as the specific utilities provided for each of them.

We will also go over some optimization tips when using reflections on your scenes, what is a depth pass and a blur pass and how you can integrate them into your own shaders.

Depth Pass

Each Planar Reflection Renderer component can have its own optional depth pass. This depth pass stores the distance between the mirror's surface and the objects reflected on it based on the depth of the virtual camera used to render it. This depth pass can be used to create complex fading effects or, in the case of the included PBR based shaders, to simulate contact reflections (that is, reflections that are sharper the closer an object gets to the reflective surface).

There are two steps to enable depth in your reflections. First, you must enable the depth pass in the Reflection Renderer to ensure that the pass is generated at all:

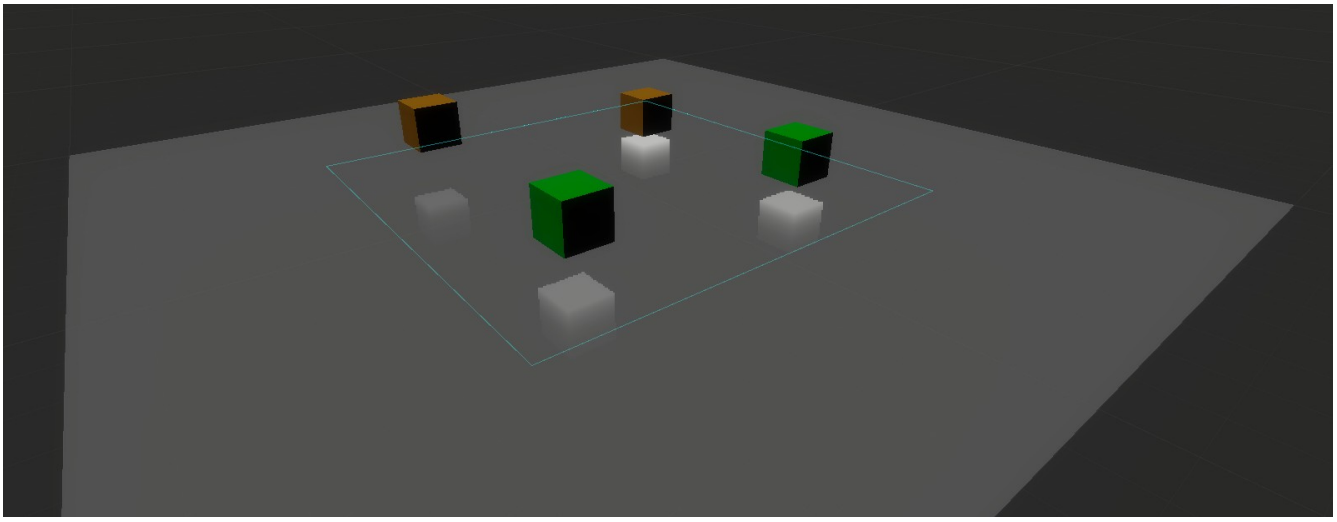


Then you have to enable the depth pass in your Reflection Caster for every material that will use it, so that the corresponding texture is sent to the material.

Once you've done this, your shaders can access the property called "_ReflectionDepth" and use its Red Channel, which now contains a default depth texture, and use it for all sorts of effects.

Please remember that not all platforms support depth textures and that some pipelines (like Universal RP) need to have the depth rendering feature enabled from their general settings.

For shader programming purposes, when depth is disabled in either the Reflection Renderer or the Reflection Caster it is set to black by default. The included nodes for ShaderGraph provide useful outputs for the reflection's depth that are ready to use.

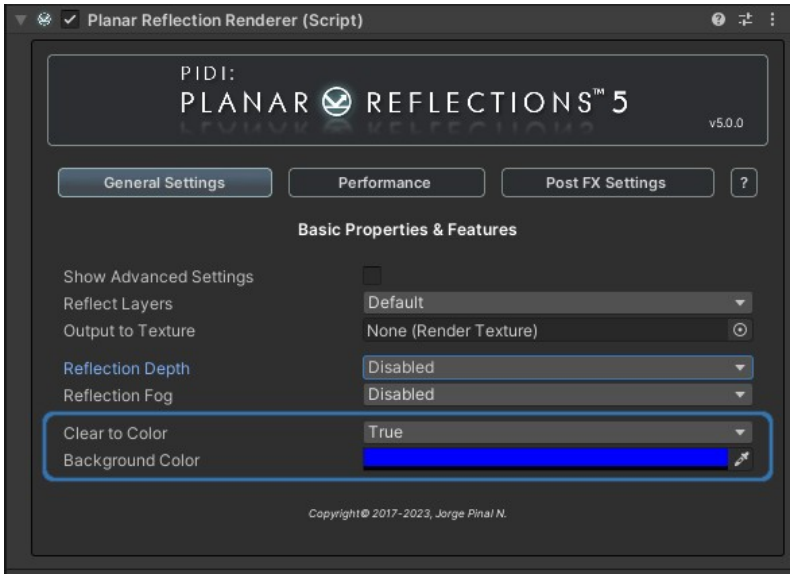


Depth pass debug

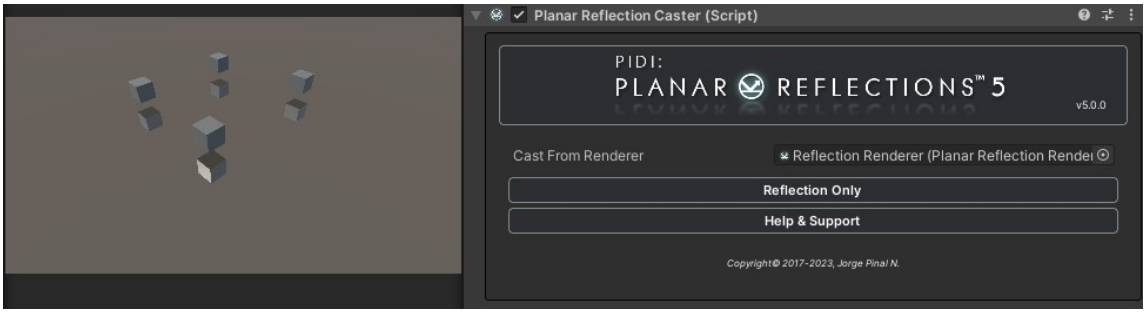
Reflection Only Mode

Starting with PIDI Planar Reflections 5 a new shader has been added that renders only the reflections and not the rest of the reflective surface. This is particular useful in cases where compositing is necessary.

First, set the Clear to Color flag to true, assign a Background color with an alpha value of zero:



Then assign the corresponding Reflection Only shader to the Reflection Caster:



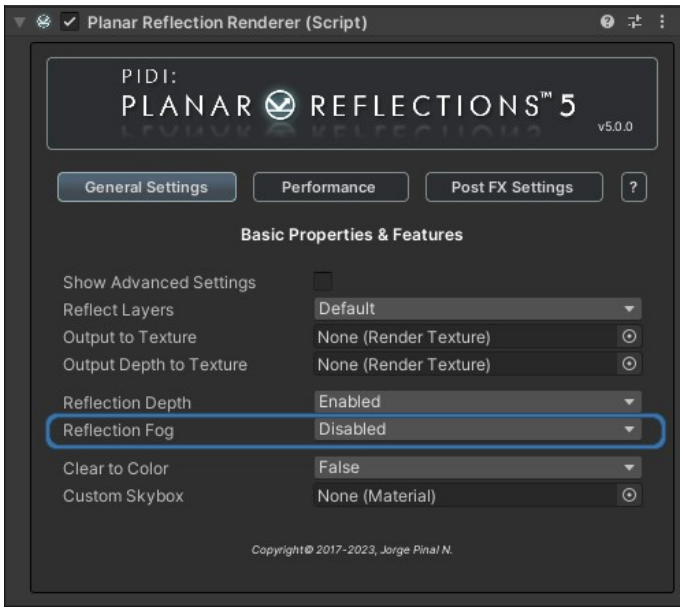
For extra precision in the Legacy (Built-in) rendering pipeline, you can enable the use of Reflection Depth in both the Renderer and the Caster.

Reflections + Fog

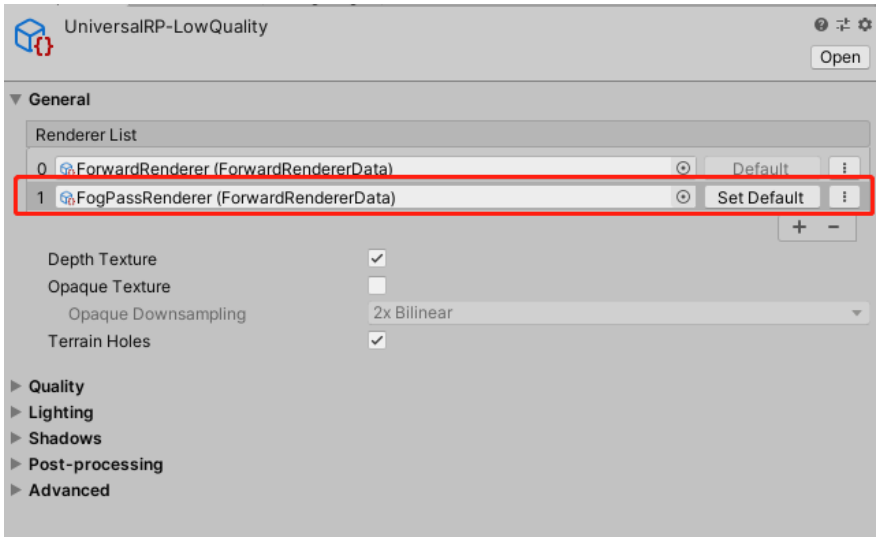
Fog in Unity doesn't always work in the way that is expected when used together with Planar Reflections. The main reason for this is the way fog is calculated using the camera's clip space, which in Planar Reflections is heavily modified due to the use of an oblique projection. Please test thoroughly whether fog is working well enough with reflections in your game before attempting to follow the steps in this guide, as there are several trade-offs when using our implementation of reflection-based fog.

In the Legacy pipeline, while in Deferred mode + Post FX fog is reconstructed or calculated from the start based on world positions, which means that it works mostly without any issues in the reflections. But for any other cases and for URP this is not possible, which means that fog has to be calculated and added back into the reflections in a different way. As a solution for this problem, PIDI Planar Reflections 5 includes a brand-new Fog Pass that generates an additional texture including world-position-based fog as seen through the reflection. This fog reads and matches all parameters set up in the Lighting settings of the scene by default, no additional work required.

Enabling fog support in the Legacy pipeline is as easy as turning on the setting in the Reflection Renderer and Reflection Caster scripts.



In URP however, due to limitations in the rendering pipeline, a special renderer is required and has to be added to the Scriptable Render Pipeline asset that your project uses. You can find this renderer inside the Fog Pass folder included with the asset.



As this is a Beta feature there are some limitations to what this fog renderer can do. The main limitation is that all objects in the fog pass are rendered as opaque objects.

Legacy (Built-in) Shaders

While several different shaders are provided for the Legacy Rendering Pipeline that cover a wide variety of uses including PBR materials, water shaders and basic mirror-like shaders, there may be a need to create your own for the specific project you are working on.

A CGInclude file is provided with the asset that contains several helpful functions that should simplify the integration of the asset with custom made shaders. Helpers for getting the reflection UVs as well as for generating a fully PBR adjusted reflection color (with and without contact depth) are provided.

half2 screen2ReflectionUVs(float4 screenPosition)

This function turns the screen position (in Unity shaders IN.screenPosition) into a set of usable UVs to unwrap the reflection data (both color and depth).

half4 PBRBasedBlur(half4 albedo, half smoothness, half4 screenPosition, half maxBlur, half3 viewDir, half3 normal)

&

half4 PBRBasedBlurDepth(half4 albedo, half smoothness, half4 screenPosition, half maxBlur, half3 viewDir, half3 normal)

This function takes several properties from the PBR shader (the same that will be sent to the corresponding PBR outputs) such as albedo, smoothness and normal, as well as the reflectionUVs, a maxBlur value (to define how much will the reflection blur on very rough surfaces) and a view direction to apply a simple fresnel effect. It can be used as follows:

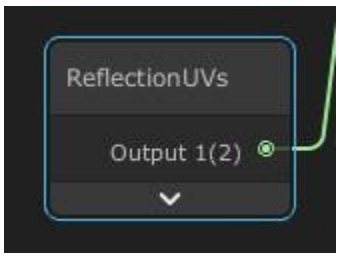
```
half4 e = tex2D( _EmissionMap, mainUV );
e.rgb *= _EmissionColor.rgb * 16 * _EmissionColor.a;

#if !_USE_DEPTH
half4 reflectionColor = PBRBasedBlur( o.Albedo, o.Smoothness, IN.screenPos, 32, IN.viewDir, o.Normal );
#else
half4 reflectionColor = PBRBasedBlurDepth( o.Albedo, o.Smoothness, IN.screenPos, 32, IN.viewDir, o.Normal );
#endif
o.Emission = e.rgb + lerp( reflectionColor, reflectionColor * (1-e.a), _EmissionMode );
```

This is an example of how to add accurate reflections to a PBR shader using the Specular workflow in the Legacy RP (it is based around the Specular shader included with the asset). Using the functions included in the CGInclude file as well as a shader_feature called USE_DEPTH to define whether a depth pass will be used or not (which can save some performance at the shader level by reducing the amount of texture operations).

Universal RP Shaders

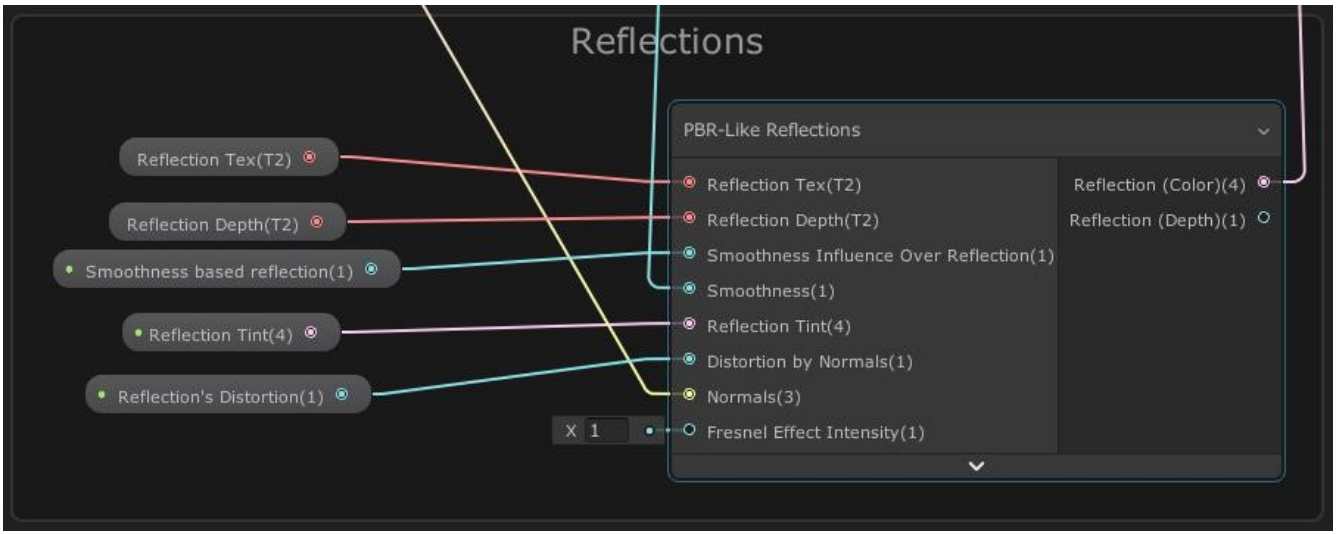
In the Universal RP you can easily create custom shaders that support real-time reflections by using the included ShaderGraph sub-graphs that handle reflection UVs (for simple reflections) and full PBR based workflows. These graphs can be edited and extended as you see fit in order to adapt to your project's purposes.

ReflectionUVs node



This node automatically generates a Vector2 type output containing ready to use UVs to use on either the Reflection color or Reflection depth textures.

PBR Like Reflections node



The PBR-Like reflection node generates a PBR based reflection that takes into account the smoothness of the material, its normal and a simple fresnel effect. It takes as inputs the **Reflection Tex** and **Reflection Depth** textures, a value to define how much the smoothness of the material will affect the sharpness and intensity of the reflection, the smoothness of the material as a float value, a color for the reflection's final tint, a value controlling how much the normals of the material will distort the reflection and a vector3 containing the normal data of the material (the same value that will be sent to the Normal output of the shader), and finally a value specifying the intensity of the fresnel effect.

This sub-graph has all the necessary operations inside to unwrap, process, blur and output both the reflection color and depth in a ready to use way that will greatly simplify the process of integrating PIDI Planar Reflections 5.x in your own ShaderGraph based shaders.

Optimization Tips

Rendering a reflection in real-time involves in most cases re-rendering your scene once for every single camera that views the reflection and for every reflection in the scene. This can cause performance to drop significantly if too many reflections are added to a scene and many of them are visible at once. While the actual cost of rendering the scene for a reflection is usually smaller than the cost of rendering the scene through the main camera there are some additional steps that you can take to improve performance even more.

Control Shadows & LOD
Use a lower LOD bias for your Reflection Renderer so that the reflections use lower quality models, reducing their performance impact. You can also change the Max. LOD value to ensure that only low poly versions of your models are reflected since, in most cases, players will not notice these smaller differences. Additionally, you can disable shadows in the reflection unless completely necessary to reduce the amount of draw calls generated by every mirror in scene.

Lower the reflection's resolution
If your game uses heavy shaders and post-process FX or has a lot of overdraw then reducing the resolution of the reflections may be helpful. Using a slight blur pass on them may also reduce the sharp edges and make them appear softer and more natural. On top of this, you can also set the reflection's resolution to depend on the main screen's resolution so that players using older devices and switching to lower resolutions get lower quality reflections automatically.

Limit the reflection's framerate
Lastly, you can limit the reflection's framerate to gain some additional performance. This however is not recommended unless your game runs at a very high framerate by default as with a lower framerate the reflections may seem too choppy producing an unpleasant effect in the scene.