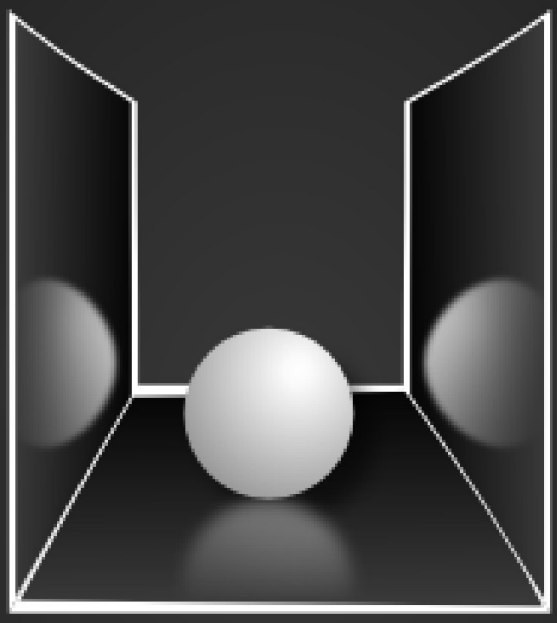

PIDI
GAME DEVELOPMENT FRAMEWORK ™
BY IRREVERENT SOFTWARE™

PIDI

GAME DEVELOPMENT FRAMEWORK ™

BY IRREVERENT SOFTWARE™



PLANAR REFLECTIONS

USER MANUAL

Index

PIDI Planar Reflections
Introduction

Planar reflections are a common resource in game development used for all kinds of reflective surfaces. Most techniques use something called an oblique projection (a perspective projection from the camera in which its sides are not symmetrical, or the clipping planes are not parallel) to cull / crop everything around and behind the reflective surface. However, in most Unity versions prior to Unity 5.5, this technique broke the shadows system. For many games, especially horror and survival games with scenes at night, this is a major flaw that renders those solutions unusable.

Our planar reflections system, which is part of our larger Framework of tools used in all our games, solves this and many other issues by providing several approximations to the oblique projection, while keeping the shadows system intact. It also uses many different optimization techniques that ensure a consistent frame-rate across many different scenes. Besides that, it gives full control over the rendered reflection, its rendering modes, updating frequencies, object pooling, global quality controls, landscape simplification, etc. For newer Unity versions (Unity 5.5 or newer), it offers real oblique projections for physically accurate mirrors along all the other optimization features.

The package also includes a small collection of shaders that showcase different uses for the system in many different conditions, from basic mirrors to reflective walls and a simple water shader. All the code is fully commented and comes with an easy to use interface.

While the system itself is easy to use and the interface provides numerous help boxes, warnings and additional information, we still recommend you to read this small user manual.

1. Adding Reflections to your scene

There are several ways to add dynamic, real-time reflections to your scene with our plugin. Which one you should use depends a lot on the kind of project you are developing and its target devices. In this section we will describe the basic and general approaches you can use for your projects.

Method 1 : The PIDI_PlanarReflection Component

The easiest (and most common) way of adding dynamic planar reflections to your scene is to simply add the **PIDI_PlanarReflection** component to a **GameObject** that has a **MeshRenderer** component and a compatible material (a material with a ReflectionTex property).

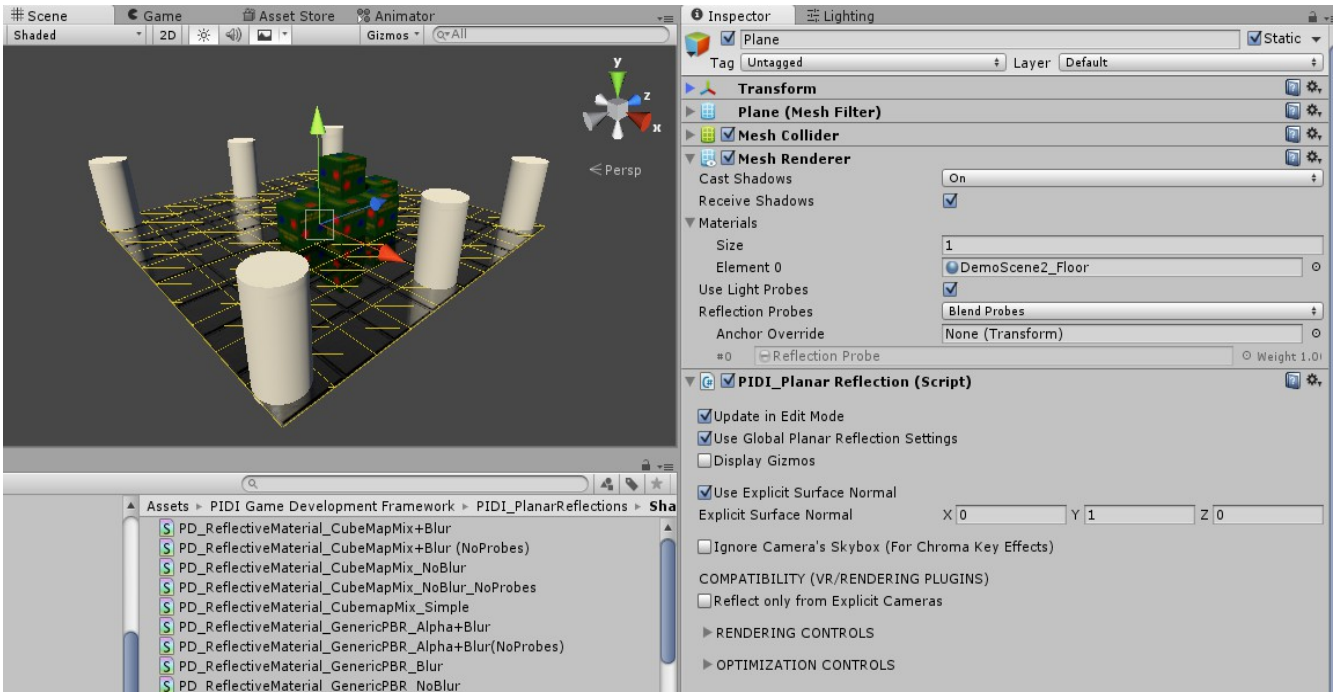


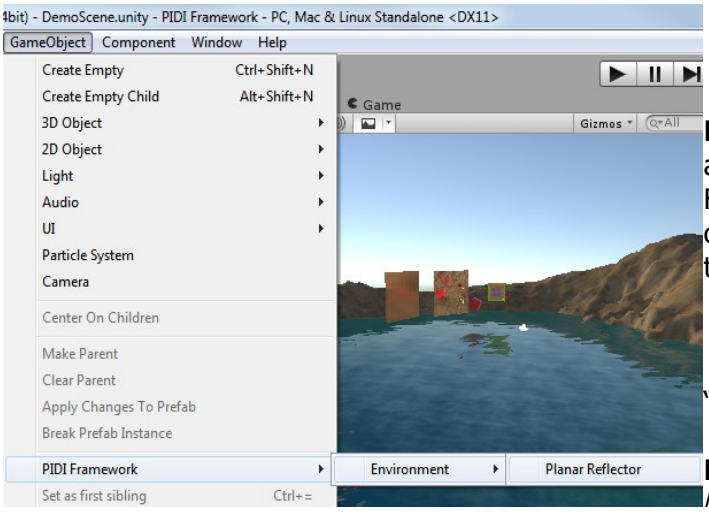
Fig. 1 – Adding the PIDI_PlanarReflection component to an existing GameObject with a MeshRenderer component.

The **Reflection** component will automatically try to use the forward direction of the object as the normal from which the reflections will be calculated. This normal is simply put a straight vector perpendicular to the reflective surface.

While this is useful for most flat objects such as floors and walls, sometimes this behavior is not optimal since the forward direction and the actual normal of the reflective surface may not match. In these cases, simply enable the **Use Explicit Surface Normal** toggle and define the explicit normal direction (in world coordinates) that will be used to generate the reflections.

Method 2 : The Planar Reflector Game Object

In some cases, such as a house with floors of different materials but all in the same plane, you may want to render a reflection that can be shared across all of them. In other cases a single mesh may contain more than one reflective material, making it impossible to add the **PIDI_PlanarReflection** component directly.



For these situations the right solution is to create a **Planar Reflector Object.** These objects render reflections to a custom **RenderTexture** asset which can then be assigned by hand to any number of materials. Furthermore, these **reflector** objects are only visible in the **Scene View** which makes them ideal for complex scenes and a more precise control over the reflections.

To create a Reflector Object, go to the "GameObject/PIDIFramework/Environment" menu on the Unity's Editor top toolbar, and select **Planar Reflector**.

Fig. 2 – Creating a PIDI – Planar Reflector Object

Once the **Reflector** is created, simply assign the **RenderTexture** asset into the **Shared/Static Render Texture** variable and it will immediately start rendering an accurate reflection.

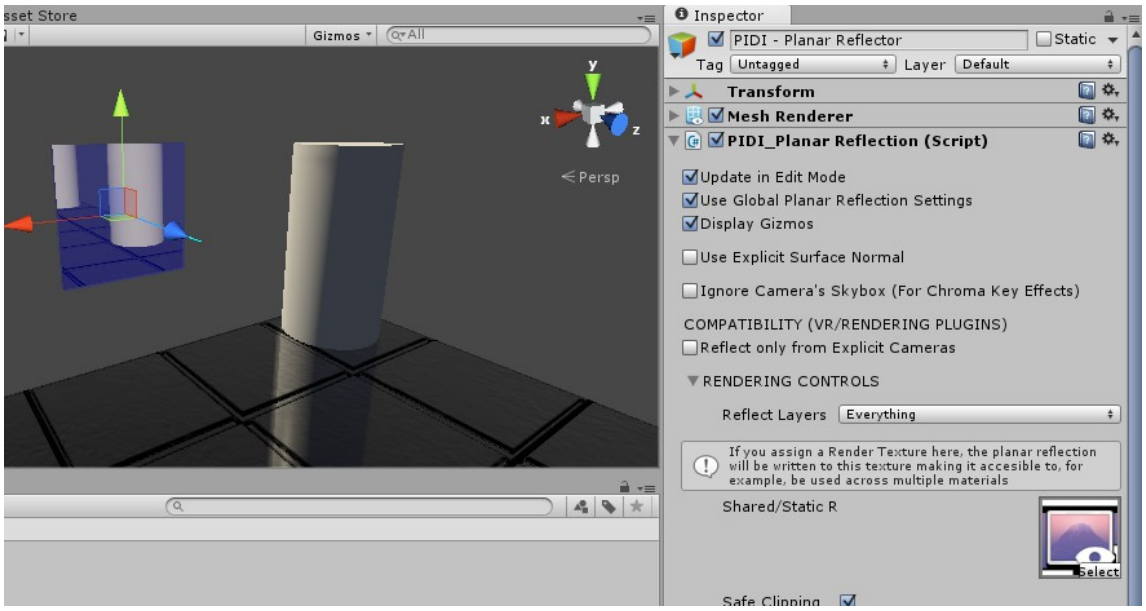


Fig. 3 – PIDI – Planar Reflector in action, with a custom RenderTexture asset assigned

2. Understanding the PIDI_PlanarReflection Component
The Inspector

The Planar Reflection Component has a custom inspector that allows you to easily manipulate its settings and configure it according to your needs. The parameters are split in three groups : **General Controls, Rendering Controls** and **Optimization Controls.**

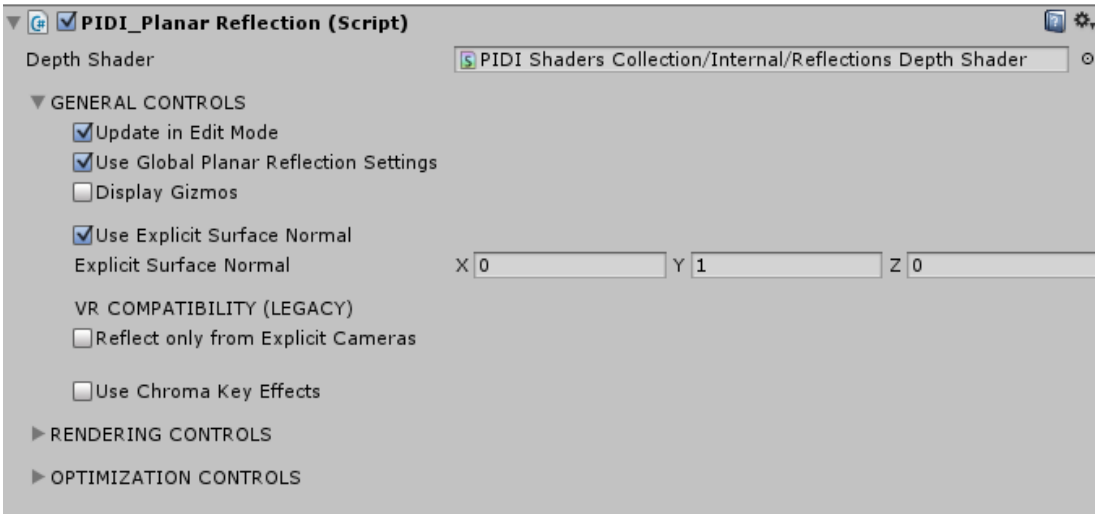


Fig 4 – PIDI_PlanarReflection component, General Control settings.

In the **General Controls** section you can find the following settings :

- **Depth Shader :** The shader to use when performing depth based calculations for the reflections. By default, it is set to the internal depth shader included with the package.
- **Update in Edit Mode :** Allows the reflections to render and be fully enabled in the Scene View while working inside the Unity Editor.
- **Use Global Planar Reflection Settings :** Defines whether this reflection will be affected by the global reflection settings (**v_maxResolution** which limits the maximum resolution of any reflection and **v_forceQualityDowngrade** which down-scales all reflections in the scene).
- **Display Gizmos :** Shows this component's gizmos
- **Use Explicit Surface Normal :** Use a Vector as the global direction of the surface normal, from which the reflections will be generated.
- **Reflect only from Explicit Cameras (LEGACY) :** When enabled, this reflection will only work on cameras called "ExplicitCamera" as defined in the VR Compatibility section of this manual. **This method is now outdated and has been replaced by the Forced Updates method (also described in the VR Compatibility section).**
- **Use Chroma Key effects :** When enabled, modifies the rendering workflow of the reflections to adapt them to the Chroma Key shaders, giving them a precise color background that will be removed by the shader.

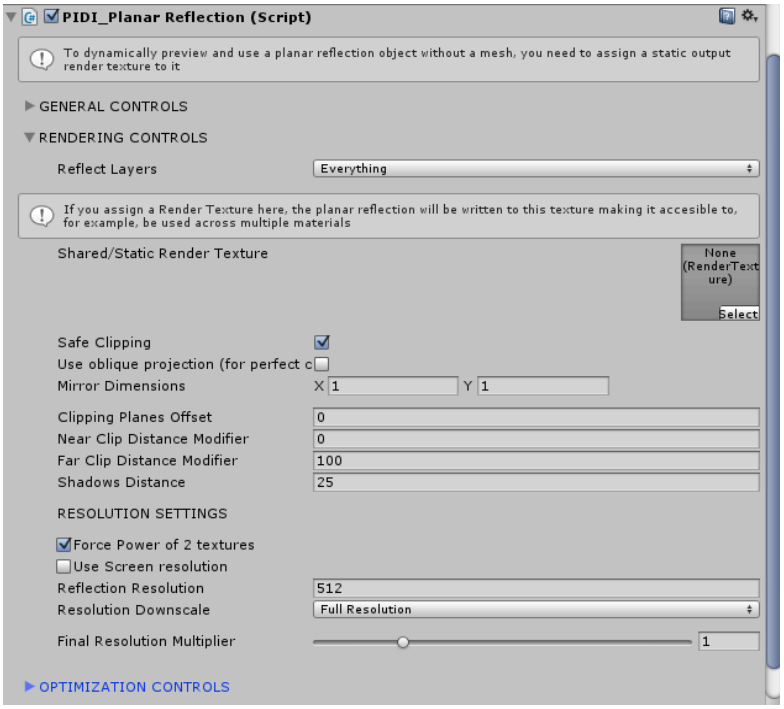


Fig 5 – PIDI_PlanarReflection component, Rendering Control settings.

In the **Rendering Controls** section you can find the following settings :

- **Reflect Layers :** Contains a layer mask that defines which layers will be reflected and which ones will be ignored.
- **Shared/Static Render Texture :** An optional RenderTexture asset to which the reflections will be rendered.
- **Safe Clipping :** Uses additional calculations to generate a more accurate projection matrix (only in Unity versions prior to Unity 5.5)
- **Use oblique projection :** Uses a real oblique projection matrix (the reflection's rendering matches the reflective surface) which makes the reflections much more accurate. In Unity versions older than 5.5 this setting will break the reflection's shadows.
- **Mirror Dimensions :** When using safe clipping, type in the approximate mirror size (for small mirrors use a size of 1-1).
- **Clipping planes offset :** Offsets the reflection to put it closer / further away from the reflective surface.
- **Near/Far Clip Distance Modifier :** Final offset applied to the near/far clip plane of the reflection camera.
- **Shadows Distance :** The maximum distance to which the shadows will still be rendered in the reflections.
- **Force power of 2 textures :** Forces the reflection to be rendered in a power of 2 RenderTexture (128,256,512,1024 etc)
- **Use screen resolution :** Renders the reflection at the full screen resolution (**Best**)
- **Reflection resolution :** The custom resolution of the reflection
- **Resolution downscale :** Allows to render the reflection at a fraction of the original resolution
- **Final Resolution Multiplier :** Allows a more precise control over the final resolution.

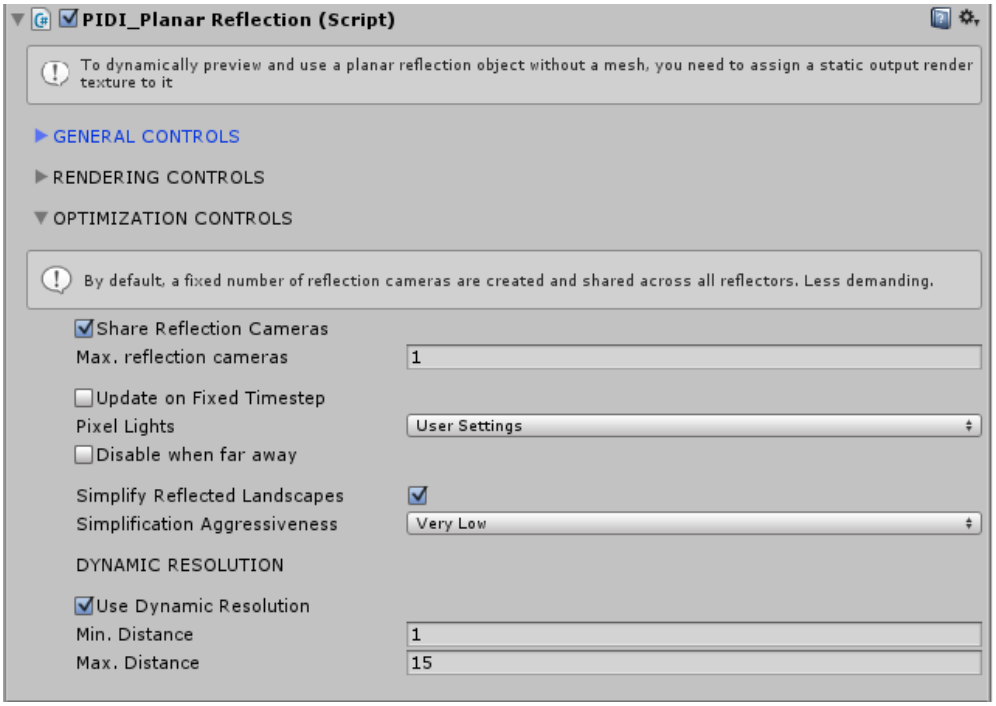


Fig 6 – PIDI_PlanarReflection component, Optimization Control settings.

In the **Optimization Controls** section you can find the following settings :

- **Share reflection Cameras :** A shared pool of cameras will be used instead of generating one reflection camera for each reflective surface and each camera that looks at it.
- **Max. reflection cameras :** The size of said pool of cameras.
- **Update on Fixed Timestep :** Updates the reflection only a certain amount of times per second.
- **Pixel Lights :** The amount of pixel lights that will be used in the reflected rendering.
- **Disable when far away :** If enabled, it will ignore cameras further away than a given max distance.
- **Simplify reflected Landscapes :** Reduces the LOD of any terrain that is reflected.
- **Simplification aggressiveness :** How aggressive will be the LOD applied.
- **Use Dynamic Resolution :** If enabled, the resolution of the reflection will be dynamically adjusted according to the distance to the current camera. This allows the reflection to reduce its quality when it is far away from the camera and become much clearer when it is closer.
- **Min/Max Distance :** When the camera is closer to the reflective surface than min. distance, the reflection will be rendered at full resolution. When it is further away than Max. distance, the reflection will be rendered at quarter resolution.

2. Understanding the PIDI_PlanarReflection Component
Additional features and general recommendations

When using the PIDI_PlanarReflections component you must be aware of several things :

- All reflections are rendered using the Forward pipeline, which may have a performance cost. It is recommended to adjust the number of pixel lights per-reflection to achieve the best performance.
- Every reflection will re-draw all the visible geometry again for each camera that is looking at the reflective surface. This is a normal behavior for Planar Reflections but must be considered when designing the scenes. For a better performance, making extensive use of the Reflected Layers feature is recommended, limiting the dynamic reflections only to the necessary layers.
- Currently, reflection cameras do not support occlusion culling as this produced undesired flickering and conflicts with Unity's own occlusion system. For a better performance, reduce the drawing distance of the reflection.
- If shaders with support for depth testing are used (**Reflection Depth** parameter) the reflection will render a depth pass, making it slightly more expensive to draw.

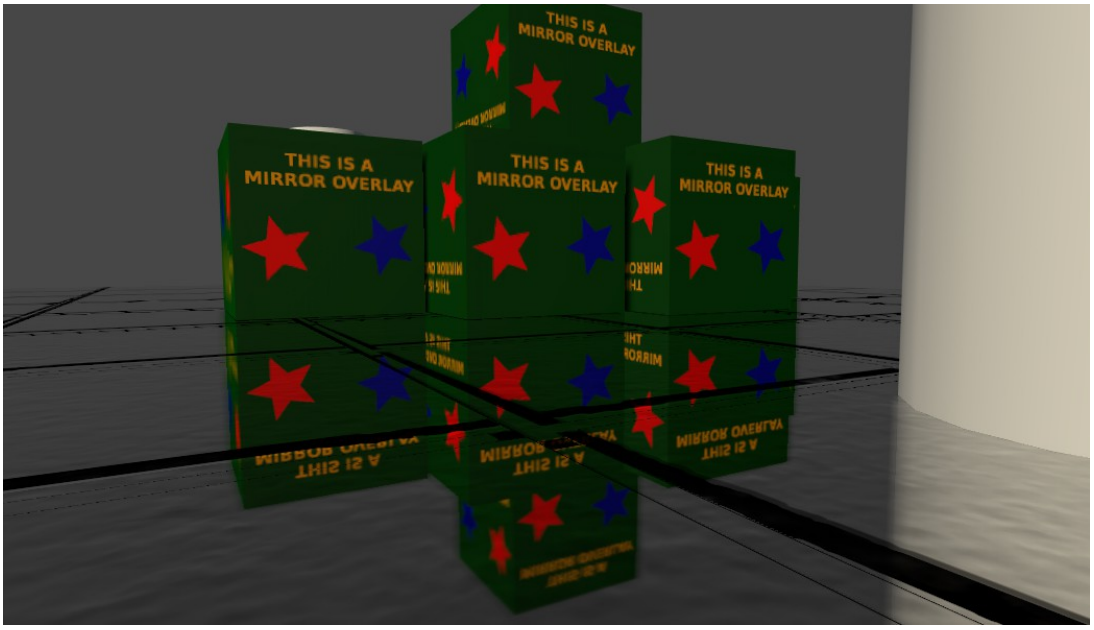


Fig. 7 – Reflection's depth in action. Objects closer to the reflective surface have sharper reflections

The depth calculations of the reflections component are handled internally, without requiring you to do anything to enable them. You just need to make sure that the PIDI_PlanarReflection component has the included depth shader assigned. To disable depth calculations simply change the shader to one without a ReflectionDepth parameter.

Reflection Depth is not supported on Planar Reflector objects since all their calculations are made in a per-surface basis and cannot be shared.

3. Setup for VR Projects

PIDI - Planar Reflections has been successfully integrated into numerous VR projects. From mobile VR projects to fully fledged ArchViz apps for desktop, our tool can be added to your VR project with little to no effort.

VR is, however, an evolving platform that is constantly changing and improving. While we do our best effort to support and adapt to the new drivers, software and hardware used by VR developers, support for every device is not guaranteed.

So far, PIDI – Planar Reflections has been successfully used with the following devices :

- **Google Cardboard**
- **Google Daydream**
- **Google VR**
- **Oculus**
- **Oculus Rift**
- **Microsoft Holo Lens.**

Other devices, including Steam's VIVE, are not officially supported

Additionally, even though the included shaders as well as the tool itself support Single-pass stereoscopic rendering with little to no distortions, a Multi-pass setup with a 2 eyed camera rig (1 camera for each eye) is recommended for maximum accuracy in the reflections.

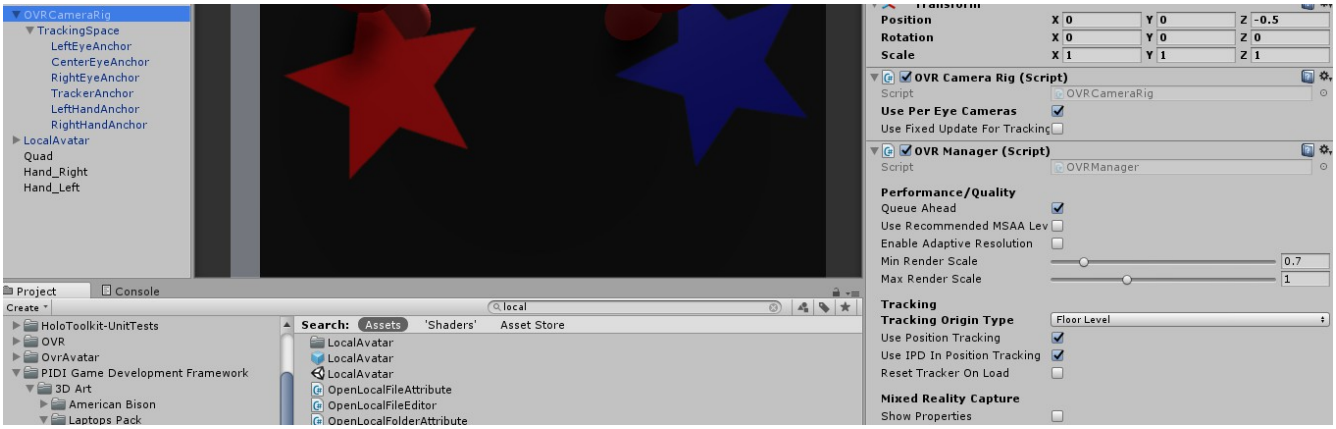


Fig. 8 – Example Camera rig for Oculus VR. Adaptive resolution must be turned off and "Use Per Eye Camera" should be enabled

In Oculus Rift devices you must remember to disable adaptive resolution as this feature causes conflicts with several features from our reflections tool.

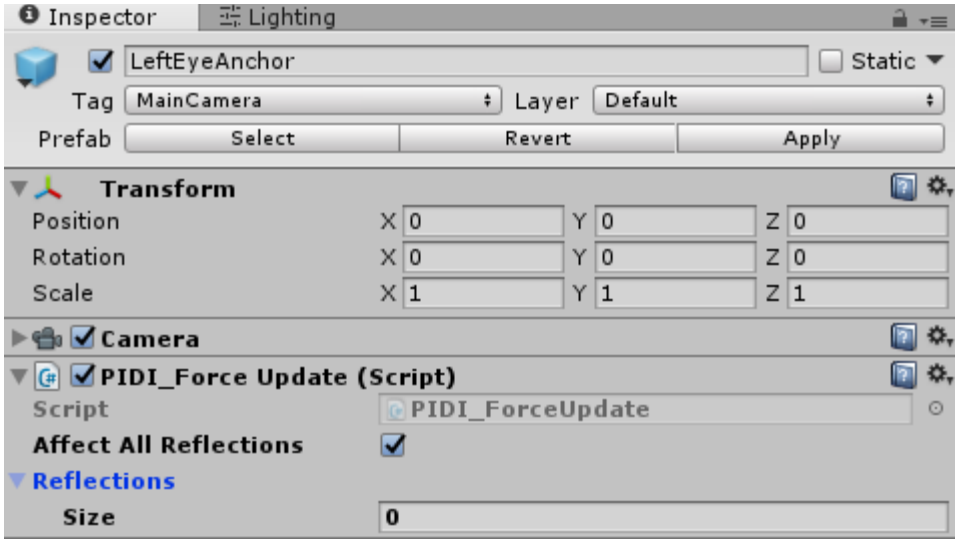


Fig. 9 – Adding the PIDI_ForceUpdate component to the left eye's camera in an Oculus VR Camera Rig

Once your initial setup is ready, simply add the PIDI_ForceUpdate component to each eye's camera and either turn on the **Affect All Reflections** flag or manually add the reflections you want to work with your cameras to the **Reflections** list.

After these easy steps, you are ready to add as many reflections as you need to your VR project.

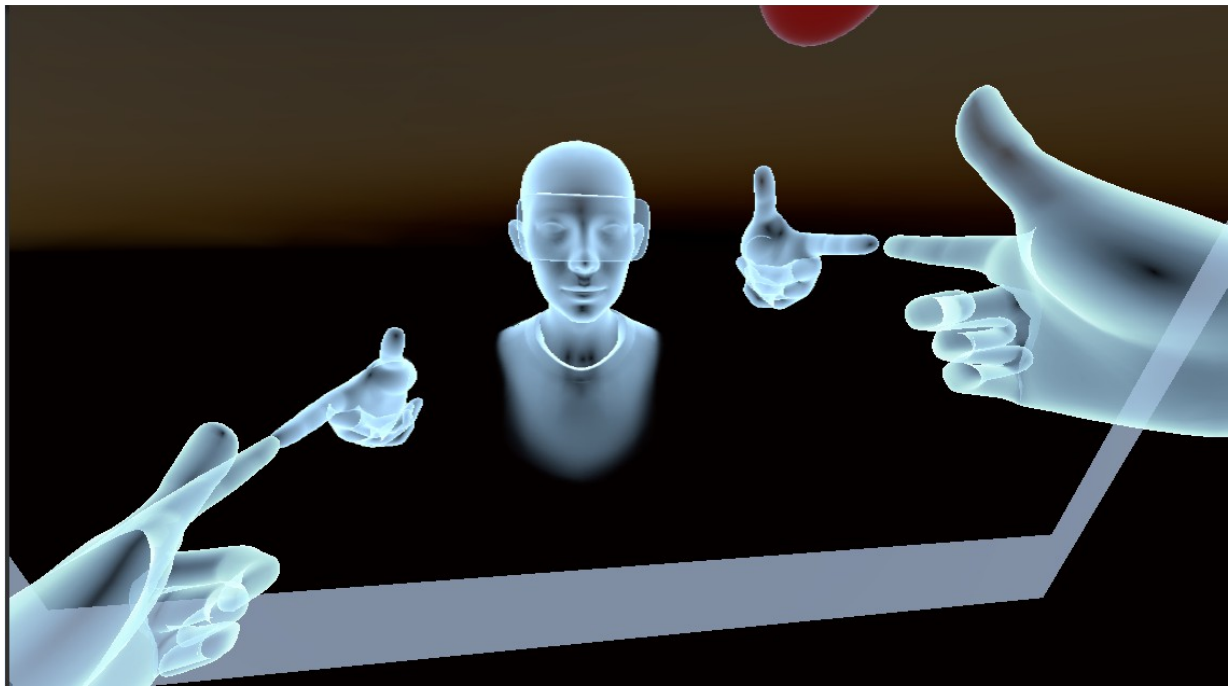


Fig. 10 – Accurate Planar Reflections inside an Oculus VR project

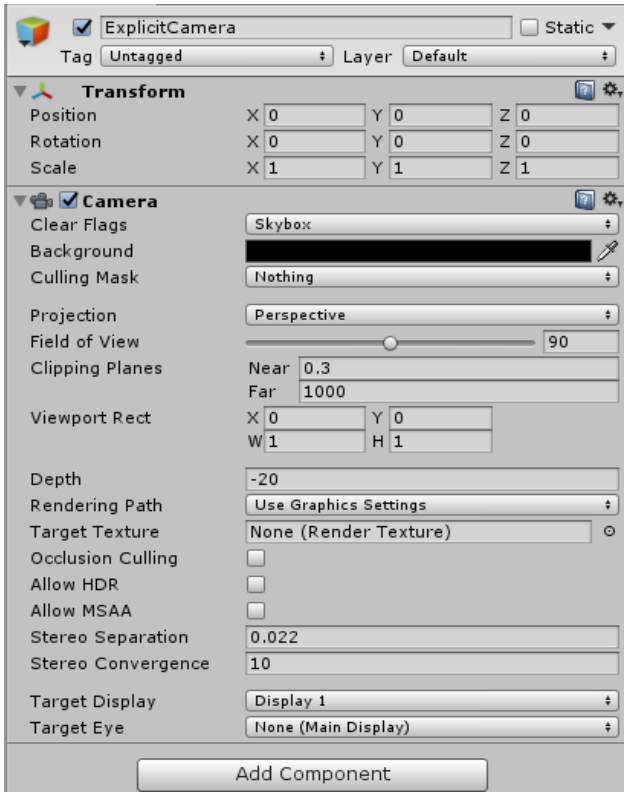
4. Legacy Solutions

Prior to version 1.5 of this tool there were 2 different workarounds designed to improve compatibility across devices and mostly for VR projects. Both methods have been replaced by the new PIDI_ForceUpdate component and are now outdated.

For new projects we recommend using the new PIDI_ForceUpdate component, but for older projects which were already using the old methods and cannot update we provide here the full legacy documentation of both the **HoloLens_ForcedUpdate** and **Explicit Cameras** workflows.

Method 1 : Explicit Cameras

Create a copy of the camera(s) that act as the main display, and add each copy as a child object of their respective originals. Each camera that is causing conflict should have a copy of itself as a child object. Name each one of those copied cameras "ExplicitCamera".



Remove all the components of the ExplicitCamera object, except the Camera component. Make sure that all the values from the Camera component in the ExplicitCamera match the ones in the parent, conflicting Camera.

Then, set Target Eye to None (if you are have enabled VR), the Culling Mask to Nothing, Clear Flags to Skybox, Depth a value lower than all the other cameras (usually -99), disable Occlusion Culling, HDR, MSAA and any other effects as well.

Finally, in all the PIDI_PlanarReflection components in your scene, enable the Reflect only from Explicit Cameras flag.

This will ensure that, during Play Mode, there are untouched and unmodified cameras to calculate the reflections from, in the same place and with the same FOV as the conflicting ones.

Fig. 11 – An Explicit Camera object correctly configured

Warning : This method will allow the tool to work with most plugins such as SEGI. HOWEVER, the reflected image will be rendered using ONLY Unity's standard rendering features (ignoring all external effects and plugins).

Method 2 : HoloLens_ForcedUpdate component.

Some devices and configurations prevent the main camera and even the explicit cameras from detecting the PIDI_PlanarReflection objects, interrupting the normal updating process. This produces an issue of "floating" reflections, reflections that move across the surface when the camera rotates or moves without being updated, giving the impression of a flat image panning across the reflective surface.

This issue has been reported by some users working with HoloLens devices. To solve it, version 1.3 introduces the HoloLens_ForcedUpdate component which, despite its name, can be used for any situation where this problem arises. For issues related to Oculus Rift Devices, we recommend to turn off the "Enable Adaptive Resolution" flag on your Oculus Rift enabled camera.

Using it is very simple. Set every reflection's "Reflect only from Explicit Cameras" flag to true and delete all the ExplicitCamera objects from the scene.

Then, add the HoloLens_ForcedUpdate component to the main camera (**the camera that is actually rendering the game**). Then all the PIDI_PlanarReflection objects to the "Reflections" list of the HoloLens_ForcedUpdate component.

Fig. 12 – The HoloLens_ForcedUpdate

The HoloLens_ForcedUpdate component will force the reflections to update every single frame by using the OnPreRender() function on the main camera.

This fix will solve most issues with most devices, with the drawback of a higher performance cost since some optimization features get disabled (the reflection is updated every single frame regardless of if it is visible or not). All other performance optimizations work as usual, reducing the performance hit as much as possible even when using this fix.

5. PIDI Reflection Shaders

In this section we will cover the basics of all the shaders included on this package as well as a small tutorial that will teach you how to add reflections support to your own custom shaders.

While the plugin is compatible with Mobile devices, most of the included shaders require support for at least OpenGL 3.0. If you are developing for Mobile devices, check the device's capabilities before using the included shaders.

The included shaders can be divided in 5 different groups : Water shader, Mirror Shaders, Generic, Chroma Key and full PBR surface shaders.

Water Shader

SIMPLE WATER, REFLECTION+REFRACTION
(PIDI SHADERS COLLECTION/WATER/SIMPLE/REFLECTION+REFRACTION)

There is one water shader included with this package, the **Simple - Reflection+Refraction** water shader. While part of the **Simple** branch of water shaders, has several important features that allow it to be easily extended into more complex shaders.

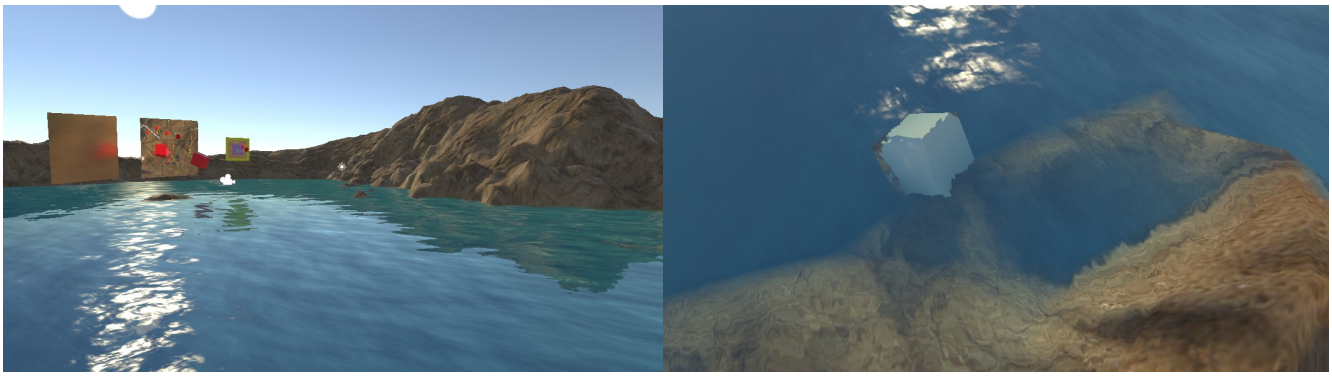


Fig. 13 - Full real-time reflection & refraction support, with fresnel based transitions and color.

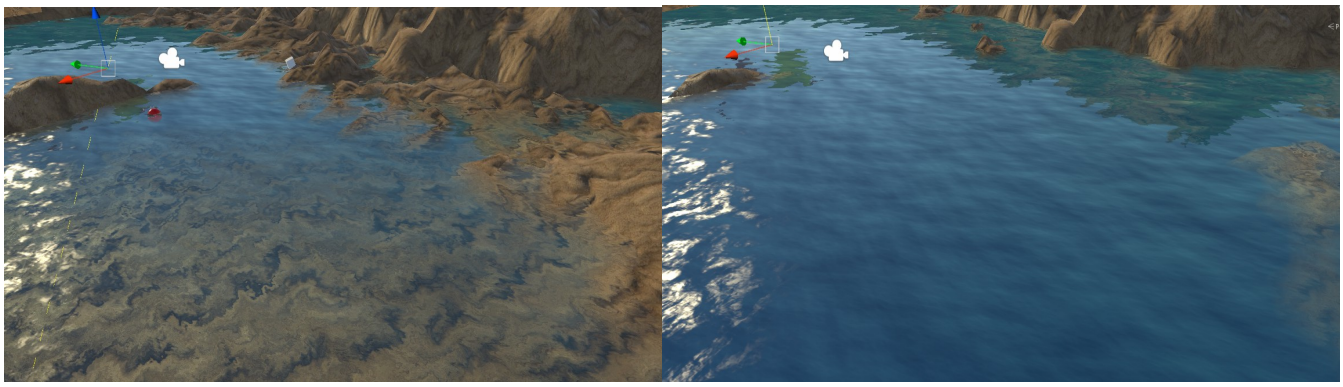


Fig. 14 - Depth based soft transparency and underwater / depths color. Transparency based on GrabPass to avoid alpha drawing overhead.

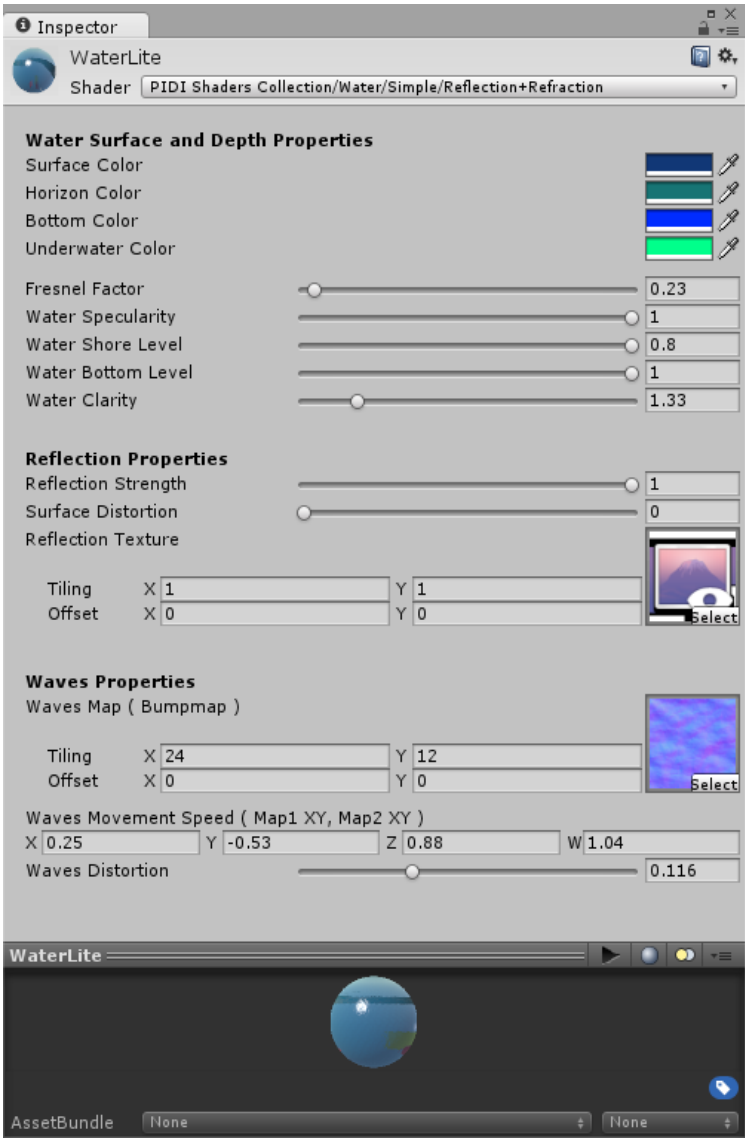


Fig. 15 – The water shader and all its parameters.

The only texture required by the shader is the Waves Normal map. A default Normal map is provided with this package.

The reflection texture can be any Render Texture and is automatically assigned if you create a plane with this material and add the **PlanarReflection** component to it.

The **Shore, Bottom** and **Clarity** levels all control the depth based rendering of the transparency and colors of the water. Change these values to move the shore line, the deep end of the water and how clear or colored is the water near the shorelines.

While the water material is easily configurable to look great in both Gamma and Linear color spaces, Linear is recommended for best and most coherent results.

2 – Mirror Shaders

There are several variants included under the mirror shaders section: a simple mirror that just renders the reflection without any further effects; a PBR mirror with support for fogginess and blurred reflections; a mirror with blur support only; mirror with fog support only and mirrors with distortions for broken reflections.

SIMPLE REFLECTION SHADER
(PIDI SHADERS COLLECTION/PLANAR REFLECTION/MIRROR/SIMPLE)

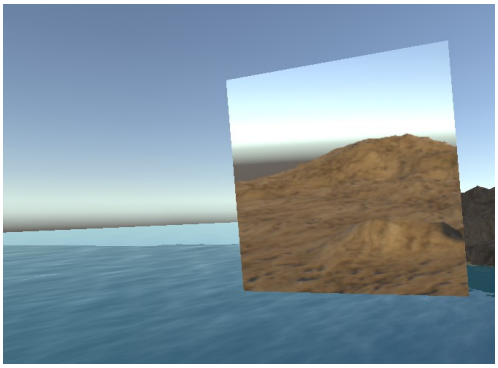


Fig. 16 - A very basic planar reflection shader.

The most basic reflective shader available, it takes a render texture with the reflection from a PIDI_PlanarReflection object and applies a color to it.

FOGGED MIRROR SHADER
(PIDI SHADERS COLLECTION/PLANAR REFLECTION/MIRROR/FOG+BLUR)

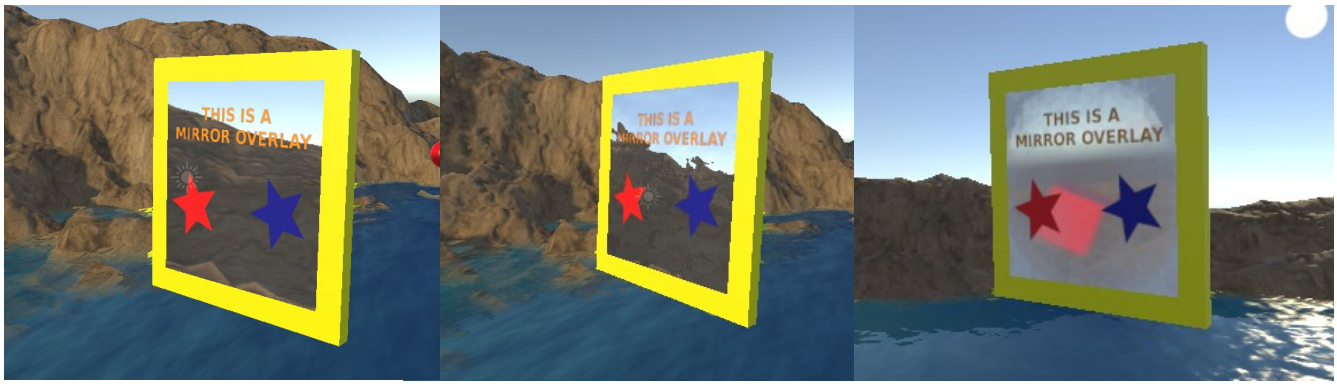


Fig. 17 - The shader supports texture overlays, PBR lighting, bump mapping and distorted/fogged/blurred reflections

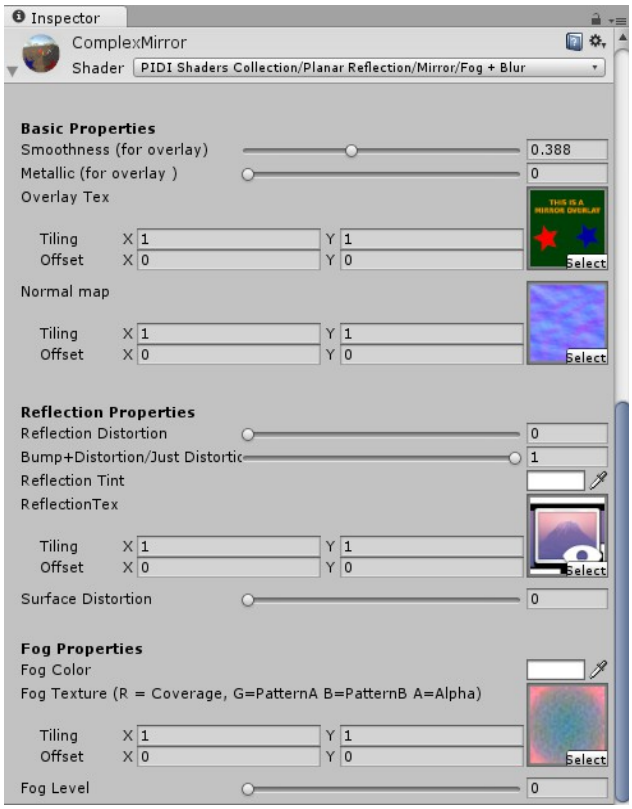


Fig. 18 – The parameters on the Mirror+Fog+Blur Shader

The shader takes an overlay/decal texture which will be rendered on top of the reflection and fog and will react to the scene's lighting. The alpha of the texture is used to show the reflection through.

A Normal map can be assigned as well, to distort the reflections, be used as a normal map or both. The Bump+Distortion/Just Distortion slider controls this transition.

The strength of the distortion effect is controlled with the Reflection Distortion slider.

A color tint can be applied to the reflection through the Reflection Tint color.

The reflection texture can be assigned automatically by the **PIDI_PlanarReflection** object.

The fog color modifies the final color of the fog in the mirror while the fog texture controls how it spreads. The red channel of the fog texture controls the spreading of the fog. When the Fog Level is changed, the fog will appear spreading from the highest to the lowest red values in the fog texture.

In the example texture this is in a radial gradient towards the center. This is further multiplied by the green and blue channels to give some variation to the spreading. The alpha channel of the fog texture is used to control precisely which parts of the mirror will not have any fog. Internally, the shader automatically blurs the reflection in the fogged zones, making the blur as strong as the fog.

Additional variants with support for only blur, only fog or only the overlay are also provided.

BROKEN MIRROR SHADER
(PIDI SHADERS COLLECTION/PLANAR REFLECTION/MIRROR/BROKEN)

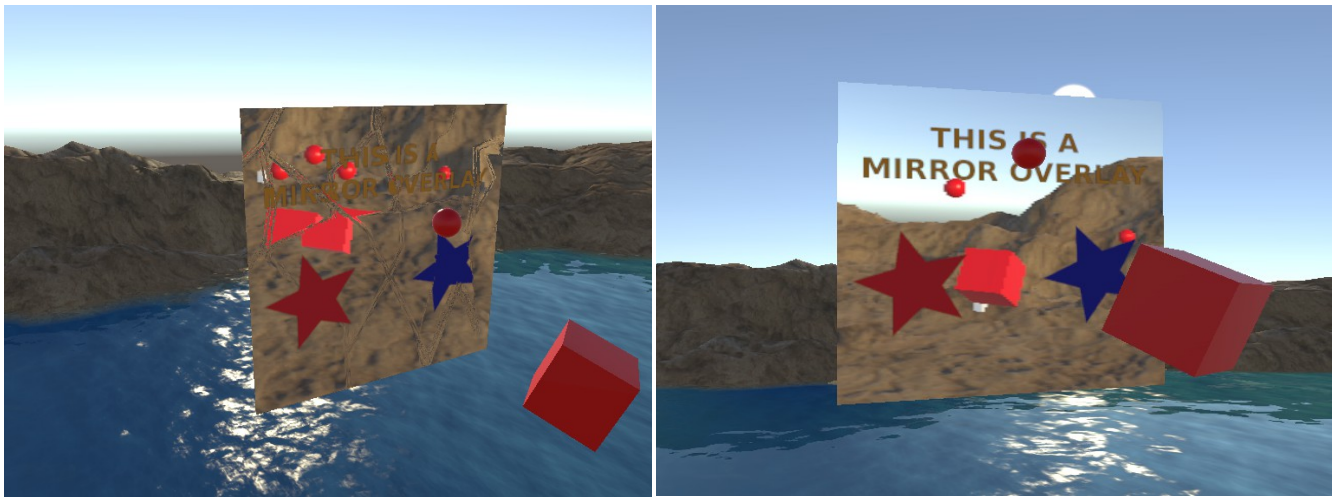


Fig. 19 - On the left, the mirror with the broken reflection effect strength set to 1. On the right, with the strength set to 0

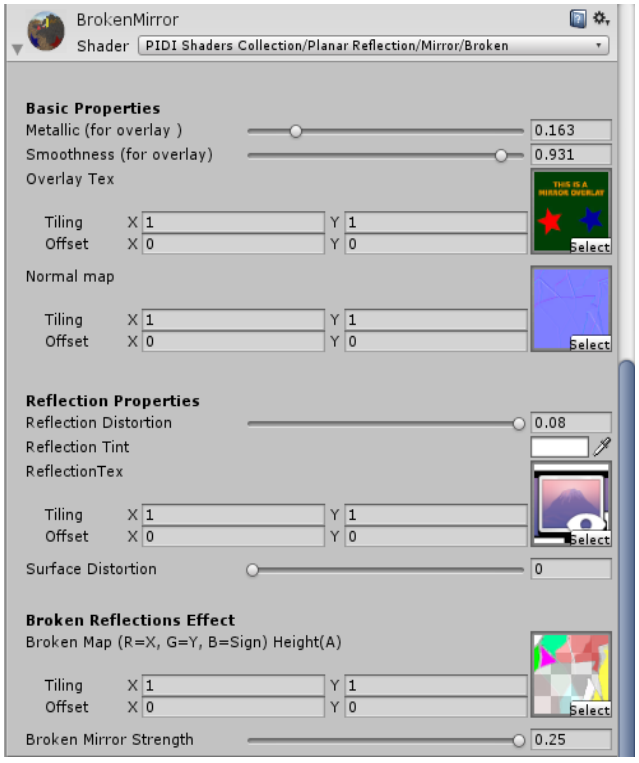


Fig. 20 – Parameters on the broken mirror shader

The shader takes an overlay/decal texture which will be rendered on top of the broken effect while being slightly distorted by it.

The Normal map is used to make a more believable effect and should match the broken pattern.

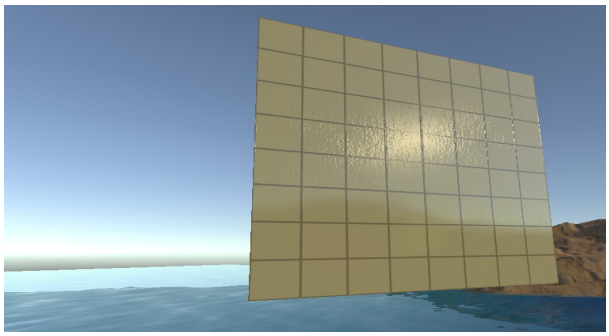
The Reflection distortion is applied to the reflection texture to make it deform along the breaking patterns and the normal map.

The broken map is a texture that controls the full breaking effect. The alpha channel acts as the height map for the distortion, the red and green channels control the displacement of the reflection along the X/Y axis and the blue channel controls the sign of the direction of the movement (B>0.5 = positive, B< 0.5 = negative)

3. GENERIC REFLECTIVE SHADERS

Four variants of generic reflective shaders are included with this package. These generic shaders are meant for regular PBR materials that should display a slight real-time reflection (such as marble floors, plastics, etc). The variants are : **Simple Surface, Simple Surface+ Blur, Depth+Blur, Alpha Surface** and **Alpha Surface+Blur**.

GENERIC REFLECTIVE SHADERS
(PIDI SHADERS COLLECTION/PLANAR REFLECTION/GENERIC)



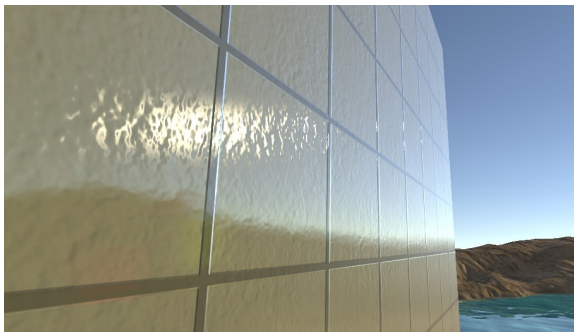
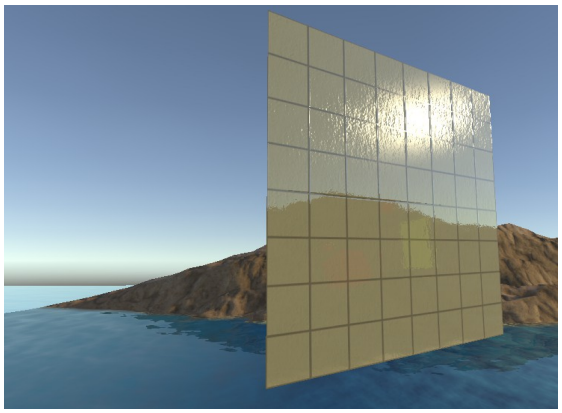


Fig. 21 - Above, the generic reflective surface with blurred view dependent real-time reflections



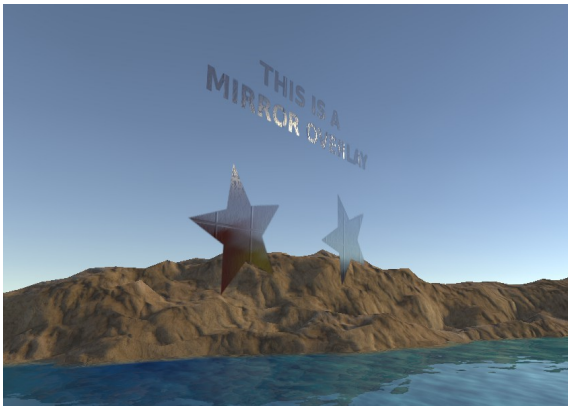
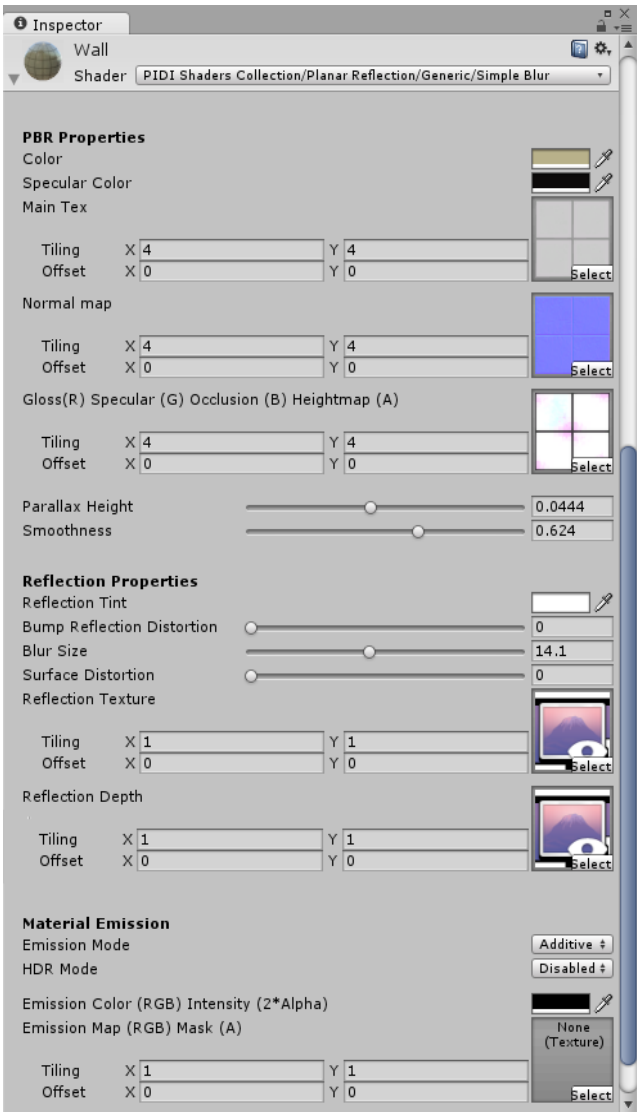


Fig. 22 - On the left, the generic reflective surface without blurred reflections (distortion only) and on the right, the transparent surface with blurred reflections.



The generic branch of shaders are based on the Standard Specular shader pipeline.

They take a **Main texture** to be displayed on the surface, a **Normal map** that will also distort the reflection, a **GSOH Map** (R=Gloss, G = Specularity, B = Occlusion, A = Height)

The **Parallax Height** value controls a basic Parallax mapping effect.

The **Reflection Tint** adds some coloring to the final reflection.

Bump Reflection Distortion is the amount of distortion that the bump mapping will apply over the real-time reflections.

Surface Distortion allows the reflections to be affected by the curvature of surfaces that are not entirely flat

Both the **ReflectionTex** and **ReflectionDepth** are provided by the Planar Reflections component automatically.

The reflections can be distorted and blurred (depending on the shader variant) and the alpha channel of the main texture can control the transparency of the material (in Alpha enabled variants)

Fig. 23 – Generic Surface+Blur shader parameters

Additionally, all the generic surface shader have full support for an emission channel to be used and give full control over how the emission will mix with the reflections.

In additive mode the reflection and the emission channels will be added to one another and mixed. In masked mode the emission channel will cover the reflections, hiding them.

Besides these parameters there is also a HDR mode that allows you to use overbrightness in your emission channel which is very useful for HDR based effects such as advanced bloom.

4. Cube map Mix / Chroma keyed shaders

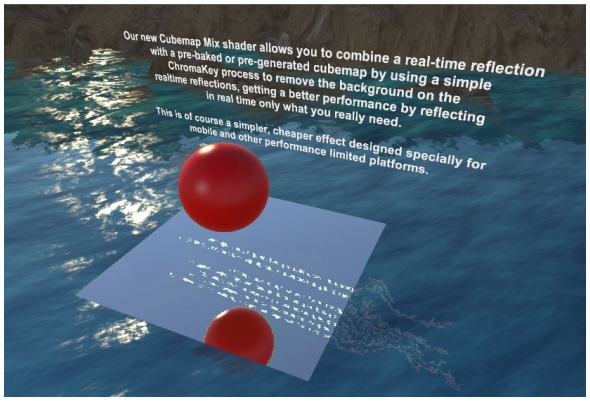


Fig. 24 – Chroma key shader in action. The ball and the text are rendered in real-time while the skybox and surrounding area are rendered through a static cubemap

In version 1.4 we have included a new kind of shader designed for performance limited platforms such as mobile, which combines a static cubemap with a real-time reflection whose background color is dynamically removed with a chroma key effect. While the effect is still expensive due to the shader operations, it is far less demanding than rendering unnecessary geometry several times (once for the main scene, once for each reflection).

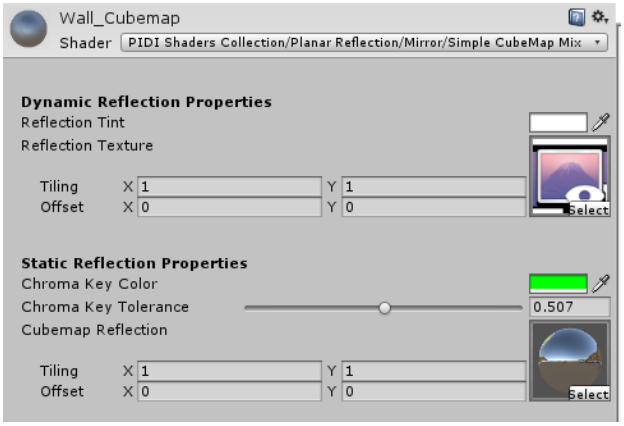


Fig. 25 - Chroma Key shader parameters

To use these shaders effectively, enable the "Use Chroma Key Effects" flag on the PIDI_PlanarReflection component and define a background color that will be removed. Then, using the Chroma Key Tolerance value, control the amount of background that will be removed.

5. Full PBR Shaders

Nine variants of PBR reflective shaders are included with this package. These shaders are meant for regular PBR materials that should mimic Unity's Standard shader as close as possible. The variants are :

- **Metallic**
- **Metallic + Depth (Blur)**
- **Metallic + Depth (Fade)**
- **Specular**
- **Specular + Depth (Blur)**
- **Specular + Depth (Fade)**
- **Roughness**
- **Roughness + Depth (Blur)**
- **Roughness + Depth (Fade)**.

PBR REFLECTIVE SHADERS
(PIDI SHADERS COLLECTION/PLANAR REFLECTION/PBR)

This collection of shaders is based on the Standard shader included with Unity and all the inputs match those of the Standard Unity shader, with just a few differences.

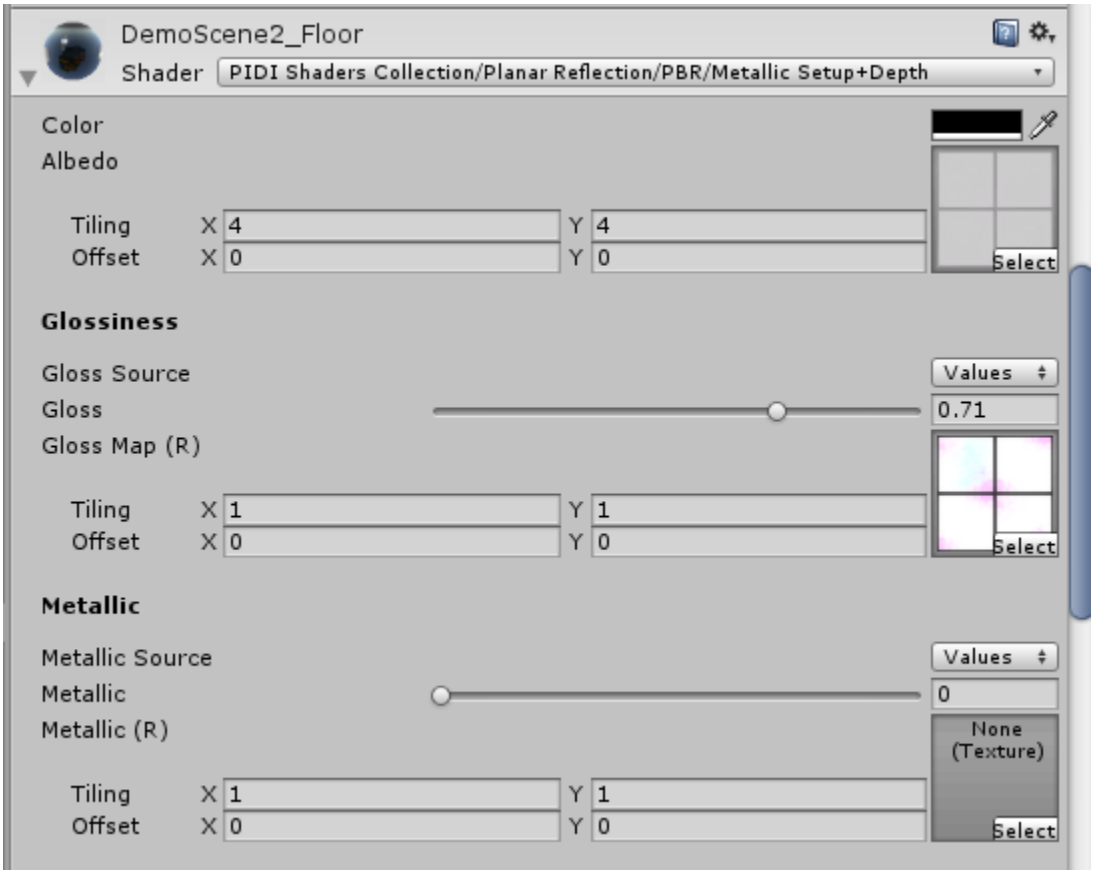


Fig. 26 – An example PBR setup shader (Metallic workflow)

For some values such as the metallic, gloss (smoothness), roughness and specular, the shader allows you to select the source of these inputs. The source can be either the direct values as shown in the shader's interface or the corresponding texture. By using the dropdown menu you can select which source to use

On top of the standard shader properties and the default reflection properties such as the reflection tint, distortion and blur, the Depth versions of these shaders also include a variable called "Depth Blur Power" in the Depth variants and "Depth Fade Power" in the Depth (Fade) variants. For complex scenes that require high quality materials that

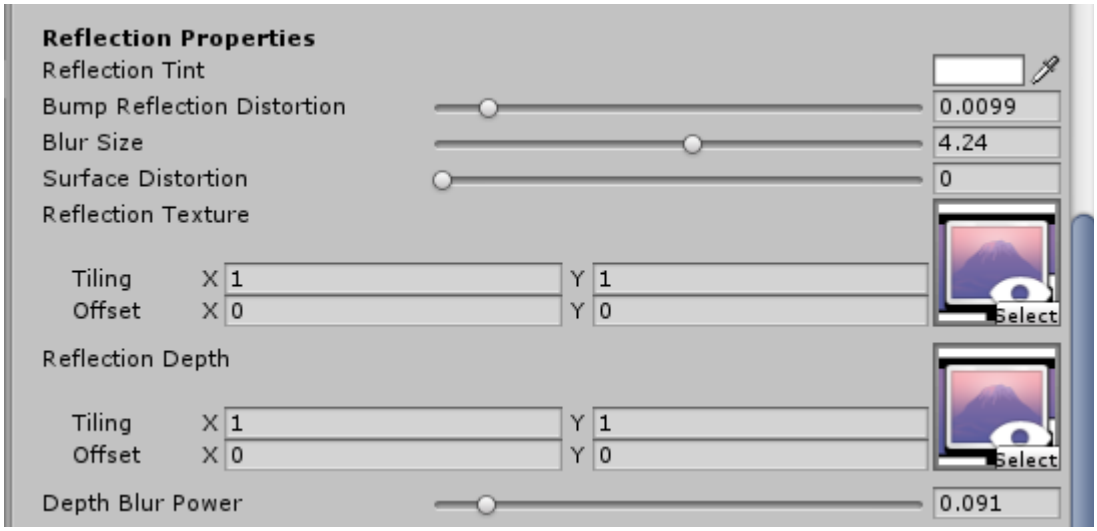


Fig. 27 – The depth power slider

This Depth power variable controls how the depth will affect the reflections, either by controlling the amount of blur applied depending on the distance between the reflected objects and the surfaces or by fading out the far away objects in the case of the Depth Fade shaders.

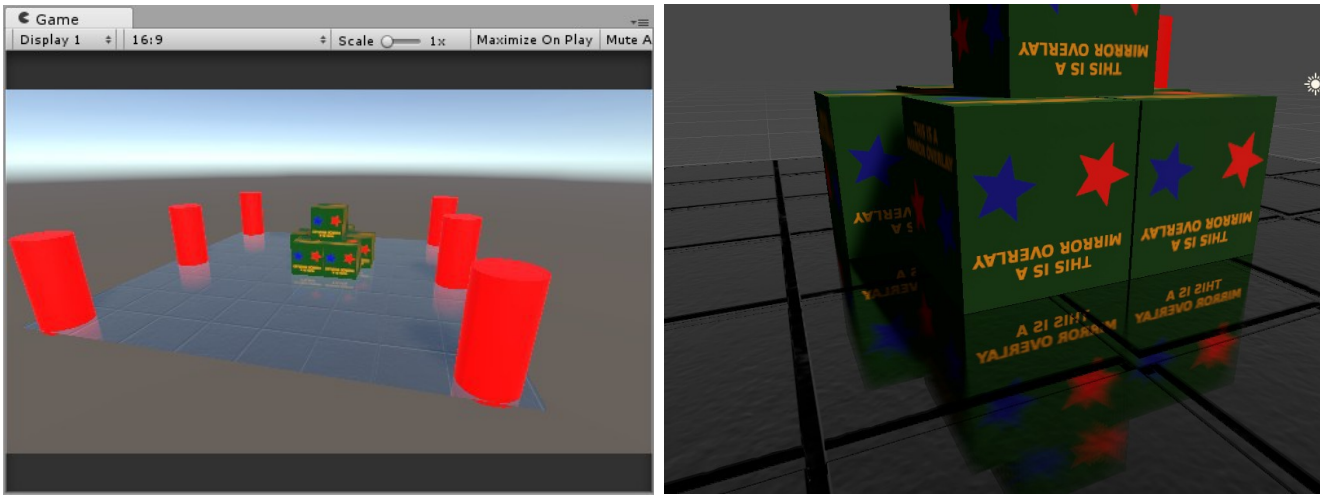


Fig. 28 – On the left, a depth fade shader with a Depth Power value of 0.5, showing how the objects fade out as they get further away from the reflective surface. On the right, a depth blur shader where the objects' reflections get blurrier the further away they are from the reflective surface.

6. Adding Reflections to a Custom Shader

Adding reflections support to a custom shader or any existing shader is very easy. There are just a few bits of code that you need to add to be ready to take full advantage of our tool.

The minimum implementation possible requires a 2D parameter called "_ReflectionTex" which will receive the output reflection texture from the PIDI_PlanarReflection component.

```
Shader "PIDI Shaders Collection/Planar Reflection/Mirror/Simple" {
    Properties {
        _ReflectionTint("Reflection Tint", Color) = (1,1,1,1) //The tint to color the reflection with
        _ReflectionTex ("ReflectionTex", 2D) = "white" {} //The real time  reflection texture
    }
```

This reflection texture will then be unwrapped using screen coordinates. The standard code for screen coordinates is the following :

```
            //We calculate the screen UV coordinates ( and ensure IN.screenPos.w is never 0 )
            float2 screenUV = IN.screenPos.xy / max( IN.screenPos.w, 0.0001 );
```

Once the texture has been unwrapped and its value stored in a half4 variable the reflection can be used without any issue. In most cases the reflection will be used through the Emission channel of the shader, but this is entirely optional and depends on the effect you want to achieve.

```

            //Our final color will be just the reflection multiplied by the ReflectionTint color.
            half4 c = tex2D ( _ReflectionTex, screenUV ) * _ReflectionTint;
            o.Emission = c.rgb*_ReflectionTint;
```

If you need complex examples of the different uses you can give to the reflections and how to integrate them to shaders we recommend you to read the source code of the included shaders as they will provide a good learning material.

If your shader is intended for VR apps and Single-pass stereoscopic support, we recommend you to read the Unity manual for your specific Unity version to find the correct method to convert standard screen space UV coordinates to the needed VR format.

```
        _ReflectionDepth("Reflection Depth", 2D) = "white"{}//Reflection depth

```

If you need to use the reflection's depth for any purpose (as measured from the reflective surface) you just need to add a 2D parameter called "_ReflectionDepth" to your shader and the PIDI_PlanarReflection component will assign it at runtime in the same way as the normal reflection. Then unwrap this texture by using the tex2Dproj function and use the length of the resulting color vector as your depth value. This depth will go from black (closest to the surface) to color (red for X axis, green for Y and blue for Z) depending on the distance to the reflective surface.

7. Scripting Reference

PIDI_PlanarReflection.cs - Public variables

Name	Definition	Type
v_reflectionPool	The static reflection pool of cameras to be shared across all reflections.	ReflectionCamera[]
v_renderingPath	The rendering path to be used by the reflection camera (OBSOLETE)	RenderingPath
v_staticTexture	Optional RenderTexture asset to be used by this reflection	RenderTexture
b_displayGizmos	Show the object's gizmos	bool
b_updateInEditMode	Update the reflection on Edit Mode and make it visible on the SceneView	bool
b_useFixedUpdate	Update the reflection at a fixed framerate instead of every frame	bool
b_ignoreSkybox	Ignores the original skybox and renders a background color instead (used by the Chroma Key shaders)	bool
v_backdropColor	The background color that will be removed by the Chroma Key Effect	Color
b_useGlobalSettings	If enabled, this component will be affected by the global reflection settings	bool
b_useExplicitCameras	If enabled, the component will render reflections only for those cameras called "ExplicitCamera". (OUTDATED)	bool
b_useExplicitNormal	Use a custom normal vector as the surface's normal instead of the forward direction of the transform component.	bool
v_explicitNormal	Custom normal vector to use as the surface's normal	Vector3
b_forcePower2	Forces the output RenderTexture to be a power of 2 texture	bool
b_useScreenResolution	Renders the reflection at a resolution based on the current screen resolution	bool

PIDI_PlanarReflection.cs - Public variables

Name	Definition	Type
b_useDynamicResolution	Adjust the reflection's resolution dynamically, according to the distance between the view and the surface.	bool
v_dynRes	The downscale factor applied to the reflection's resolution	int
v_resMultiplier	Final reflection's resolution multiplier	float
v_minMaxDistance	If the camera is closer than v_minMaxDistance.x the reflection will be rendered at full resolution. If it is further away than v_minMaxDistance.y it will be rendered at a quarter resolution.	Vector2
b_useReflectionsPool	Use a reflection's pool instead of creating a reflection camera for each surface and each point of view.	bool
v_poolSize	The amount of reflection cameras in the shared pool	int
v_disableOnDistance	Disable this reflection when the camera is further away than the specified distance. Ignored if v_disableOnDistance < 0	float
v_resolution	The reflection's resolution	Vector2
v_pixelLights	The amount of pixel lights that can affect the rendered reflection (since the reflection is rendered in the Forward pipeline)	int
v_reflectLayers	The layers that will be rendered by the reflection	LayerMask
b_simplifyLandscapes	Reduces the LOD of any terrain that is drawn by the reflection camera	bool
v_agressiveness	How much will the terrain LOD be reduced	int
v_shadowDistance	The drawing distance of the shadows as rendered in the reflection	float
v_clippingOffset	Offset from the reflective surface to the actual position of the reflection camera from which the reflection is rendered	float
v_nearClipModifier	Modifier to further adjust the near clipping plane (which is copied from the camera that is looking at the reflective surface)	float
v_farClipModifier	Modifier to further adjust the far clipping plane (which is copied from the camera that is looking at the reflective surface)	float

PIDI_PlanarReflection.cs - Public variables

b_safeClipping	Use an approximated oblique projection to cull anything behind the mirror	bool
b_realOblique	Use an accurate oblique projection (breaks the reflected shadows in Unity versions prior to Unity 5.5)	bool
v_mirrorSize	The approximate mirror size (usually set to 1,1) to be used by the safe clipping approximated oblique projection	Vector2

PIDI_PlanarReflection.cs - Global variables

v_maxResolution	The maximum resolution that any given reflection in the scene can have. Controled globally	int
v_forcedQualityDowngrade	The downscale factor applied to the resolution that any given reflection in the scene can have. Controled globally	int

ReflectionCamera – Struct

reflector	The camera that will render the actual reflection	Camera
source	The source camera that is looking at the reflective surface	Camera
owner	The owner component of this reflection camera	PIDI_PlanarReflection

8. Final Notes

The tool has been thoroughly tested on a Windows 7 machine with the following specs :

System : Windows 10 64 bits
Processor : Intel Core i3 5200U locked at 75% speed.
RAM : 8GB
Graphics : Intel HD 5000

Our benchmarks uses 4 real-time reflectors, 1024x1024 resolution each, inside the Unity Editor on the demo scene and we tested it with and without optimization controls enabled.

No optimization : **100~FPS** With Optimization : **260~FPS**

The tool is aimed at PC game development but it has also been used successfully on many mobile devices (even while using Google VR).

The performance of the tool depends entirely on the complexity of the scene and the amount of objects that will be reflected.

While this software has been thoroughly tested across all Unity versions from Unity 5.2 to Unity 2018.1 and extensive optimizations and fixes have been included to avoid any bugs, the software is provided as is and without any explicit warranties. We strongly encourage you to backup all your work before making use of our tool or any other plugin from any provider.

If you have any troubles with our software or need any assistance, don't hesitate in contacting us at support@irreverent-software.com and we will get back at you with help.

Thank you very much for purchasing our software, we hope our tools and the PIDI Game Development Framework will help you make awesome games!

The Irreverent Software Team.