Blend Modes

Documentation for version 3.10 (Unity 2019.1 and later); legacy versions docs: https://goo.gl/CXo6RN

Thank you for purchasing Blend Modes! You now have the ability to apply 25 different blend modes to UI graphics and text, sprites, meshes, particle systems, trails, cameras and other Unity objects. This documentation will help you to get started using the plugin.

I really hope you'll like Blend Modes! If you feel like it, please leave a review on the Asset Store, that helps a lot.

Introduction

Blend modes (or mixing modes) are used to determine how two layers are blended into each other. The default blend mode used for most of the objects in Unity is simply to hide the lower layer with whatever is present in the top layer (so called normal blend mode). Using different blend modes, you can achieve a wide range of effects and create a truly unique look for you game.

Blend Modes is a plugin for Unity, which consists of a shader pack and editor extensions that provides an easy way to apply blend mode effects to the game objects.

List of all the available blend modes

You can set any of the following 25 blend modes:

- Darken
- Multiply
- Color Burn
- Linear Burn
- Darker Color
- Lighten
- Screen
- Color Dodge
- Linear Dodge (aka **Additive**, **Add**)
- Lighter Color
- Overlay
- Soft Light
- Hard Light
- Vivid Light
- Linear Light
- Pin Light
- Hard Mix
- Difference
- Exclusion
- Subtract
- Divide
- Hue
- Saturation
- Color
- Luminosity

Getting started

Initial setup

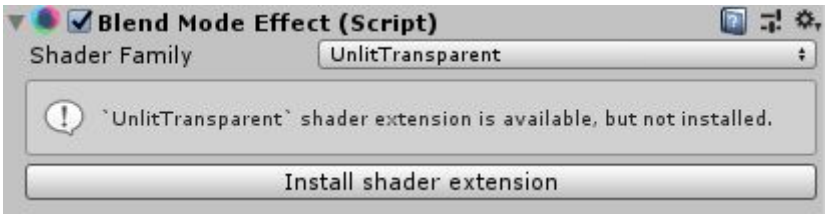
The plugin does not require any special setup: just import the package and you are ready to go! You are free to move the “BlendModes” folder anywhere inside your project assets directory.

Updating from legacy versions of the package

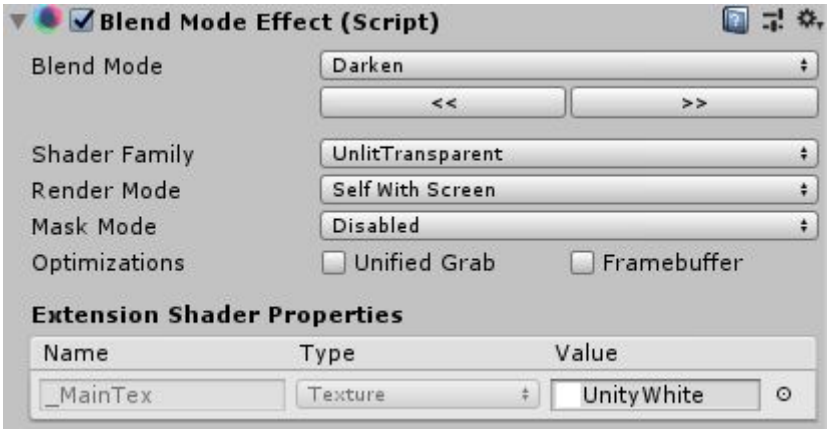
In case you already have BlendModes versions 1.0-2.15 installed, please delete the “BlendModes” folder from the project before importing the updated package. Be aware, that you may need to re-add and re-configure “Blend Mode Effect” components after the update.

Applying the effect for the first time

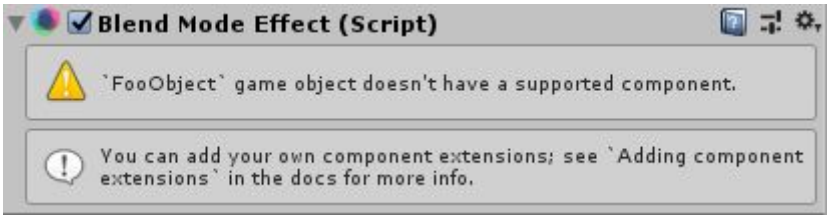
To apply the effect to a game object, just add a “Blend Mode Effect” component. If the object has a supported renderer, you’ll see a shader family selection box and message, that corresponding shader extension is available, but not installed:



Click the “Install shader extension” button to install the required shader extension and wait for the generated shader assets to compile. Now you’ll be able select the desired blend mode and other parameters:



In case no supported components are found on the game object, you’ll see the following message:



As stated in the message, you can add your own extension components to support any third-party renderer that allow setting custom materials; see “Adding component extensions” tutorial for more info.

Please note, that stacking multiple `Blend Mode Effect` components on a single object is only supported for cameras and could lead to an unexpected behaviour for the other object types.

Shader family

Shader family is a set of shaders based on a single master shader. For example, an “UnlitTransparent” shader family is based on the Unity’s default “Unlit-Transparent” shader used for materials that ignore lights and support transparency. To support all the different render and mask modes of the Blend Mode Effect, multiple shader variants are generated, thus forming a shader family.

Available shader families for any given renderer are specified in the corresponding component extension. When only one shader family is specified, the selection box in the editor will be hidden.

List of the currently available shader families

Camera	Used to apply blend mode effect to cameras
DiffuseTransparent	Based on Unity’s `Diffuse-Transparent` shader; supports lighting and transparency
ParticlesAdditive	Based on Unity’s `Particles-Additive` shader; used for particles and other VFX
SpritesDefault	Based on Unity’s `Sprites-Default` shader; used for sprites
TMProDistanceField	Based on TextMesh Pro `DistanceField` shader; used for TMPro UIs
TMProMobileDistanceField	Optimized version of the `TMProDistanceField` shader family for mobile platforms
UIDefault	Based on Unity’s `UI-Default` shader; used for UI graphics
UIDefaultFont	Based on Unity’s `UI-DefaultFont` shader; used for UI text
UnlitTransparent	Based on Unity’s `Unlit-Transparent` shader; ignores lighting, supports transparency

You can add your own shader families; see “Adding shader extension” tutorial for more info.

Render mode

The Blend Mode Effect can be applied in two modes: “Self With Screen” and “Texture With Self”.

Self With Screen

While using this render mode the object's texture will blend with anything that was drawn before on the screen. The effect is similar to layer blending in Photoshop.

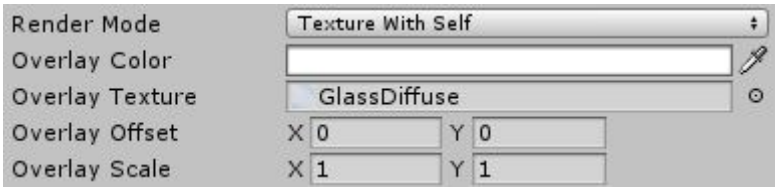
Be aware, that in order to access the screen texture (aka backbuffer) a grab pass is used, which could hurt the performance on some older mobile devices. Consider enabling available optimization options when using multiple instances of the effect working in this render mode.

Texture With Self

Similar to “Pattern Overlay” layer style in Photoshop, this render mode will blend specified overlay color and texture with the object's texture.

This mode will not execute any expensive operations and could be used freely when targeting low-end mobile devices.

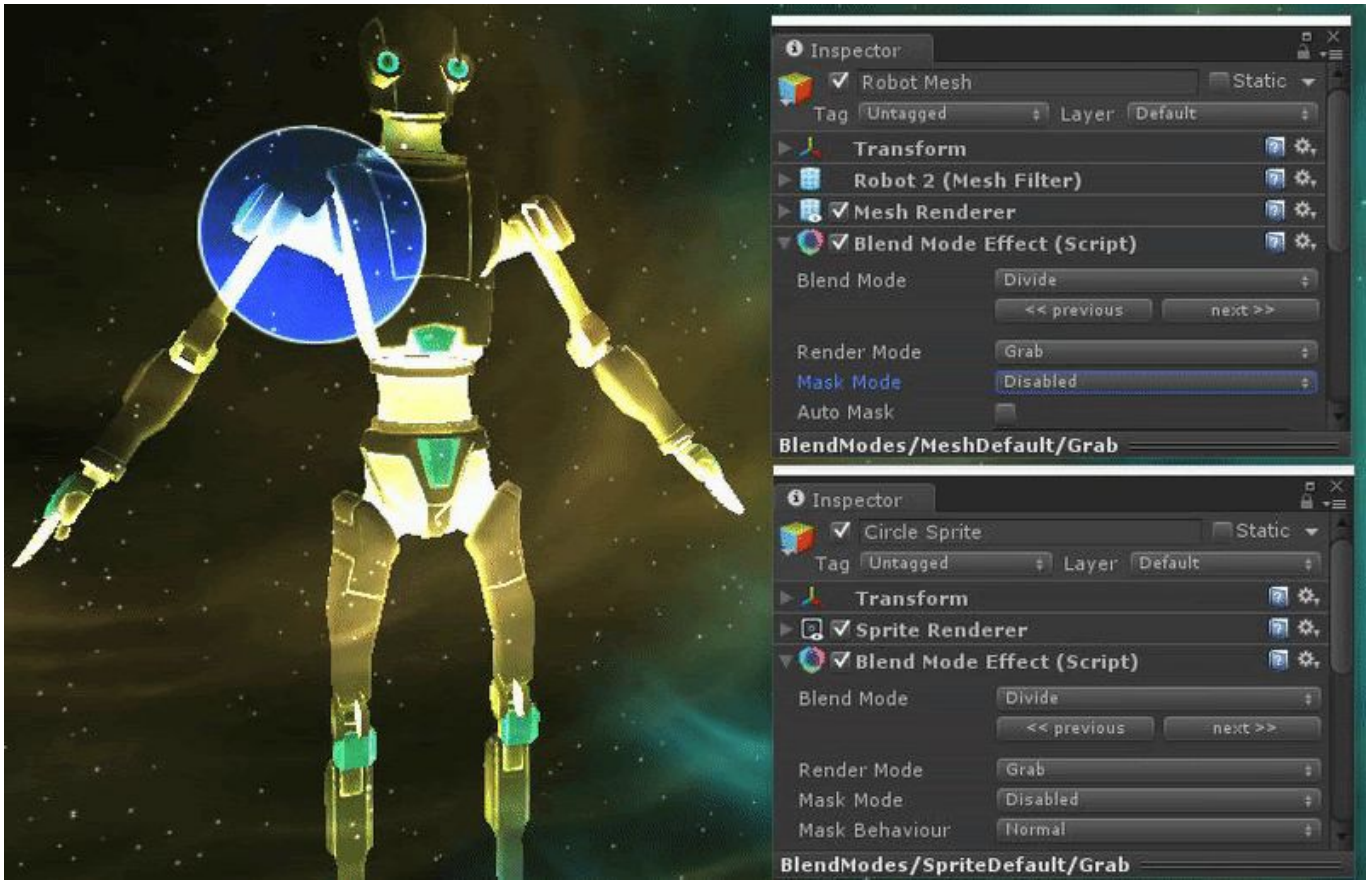
When “Texture With Self” mode is activated you’ll see additional parameters in the editor:



Overlay Color	Choose a color to blend with the object’s texture; when a `Overlay Texture` is set, it will also work as a tint color.
Overlay Texture	Choose a texture to blend with the object’s texture
Overlay Offset	UV offset of the `Overlay Texture`; you can also animate this property to achieve various special effects
Overlay Scale	UV scale of the `Overlay Texture`

Masking (selective blending)

It's possible to selectively blend an object with blend mode effect via the masking feature. For example, you can apply the effect only when the blended object is over specific (masked) objects and fill the rest with normal blend mode or cut the pixels out (make them transparent).



To configure masking, use the **Mask Mode** property:

- **Disabled**: ignore masks, blend with everything (default);
- **Nothing But Mask**: blend only with objects affected by mask;
- **Everything But Mask**: blend with everything, except objects affected by mask.

You can also control what will be drawn in the masked areas via the **Mask Behaviour** property:

- **Cutout**: pixels from the masked areas will be discarded (become transparent);
- **Normal**: pixels from the masked areas will be rendered in normal blend mode.

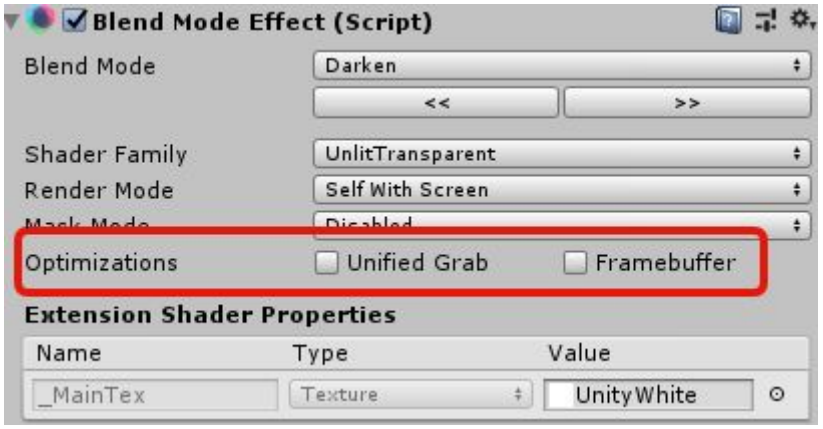
For the masks you can use Unity's "Mask" and "Sprite Mask" components.

Stencil buffer is required for the masking feature. Enabling "**Disable Depth and Stencil**" optimization option in the build settings will prevent the effect from working correctly when masking is enabled.

Performance optimizations

`Self With Screen` optimizations

While using “Self With Screen” render mode, you’ll have two performance optimization options:



Unified Grab

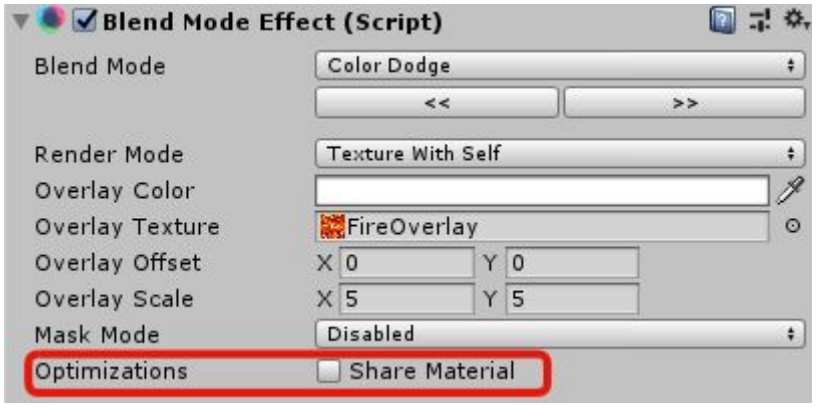
Share a grab texture with the other instances of the effect, that have this optimization enabled. Significantly improves performance when multiple blend mode effects are used simultaneously, but the instances won't properly blend with each other.

Framebuffer

Use [`framebuffer_fetch`] instruction (when available) instead of a grab pass to access the backbuffer. Significantly improves performance, but is only supported on selected devices (most notably PowerVR-based ones on iOS). You can safely assume this will work on all the modern iOS devices (iPhone 4S / iPad 3 or newer).

`Texture With Self` optimizations

While using “Self With Screen” render mode, you’ll have following performance optimizations available:



Share Material

When enabled, material instances with the same shader family and blend mode type will be shared. Can reduce draw calls count in some cases, but all the material properties will also be shared.

Component extensions

To work with any given renderer component, Blend Mode Effect needs a corresponding component extension. A component extension usually consists of a single C# script, that specifies how to get and set materials for the target renderer and which shader families it supports. When you first import the BlendModes package, all the component extensions are not installed by default to prevent dependency issues and save compile time for extensions you may not need at all in your project.

Accessing component extensions

In case you wish to access a component extension used by a particular gameobject, use `GetComponentExtension<TComponent>` method of the `BlendModeEffect` component attached to the object, `TComponent` being the type of the component extension. In case you don't know the exact type, you can use a base extension type `ComponentExtension`. Using `GetRenderMaterial` method of the component extensions you can access the material currently used by the object to change its properties, like the main texture, scaling, color, etc.

Adding component extensions

For example, let's add support for the `Line Renderer` component:

1. Create a C# script somewhere inside project assets, name it anything you want (`LineRendererExtension`, for example);
2. Include`BlendModes` namespace;
3. Inherit our newly created class from the `ComponentExtension` base class and decorate it with the `[ExtendedComponent(typeof(LineRenderer))]` attribute. The attribute will be used by the extension system to find that this class is actually extending the `LineRenderer` type;
4. Override `GetSupportedShaderFamilies` method to specify which shader families should be available for this component extension. `ParticlesAdditive` will work in this case;
5. Override `GetDefaultShaderProperties` method to specify, which shader properties should be exposed to the editor. `ParticlesAdditive` shaders use `_MainTex`, `_TintColor` and `_InvFade` properties, so add them here;
6. Override `GetRenderMaterial` method, so that it will return the render material currently used by the extended component. You can always get a reference to the extended component using the `GetExtendedComponent` method from the `ComponentExtension` class, which we inherited. So, to return the render material currently used by our line render we can do the following: `GetExtendedComponent<LineRenderer>().sharedMaterial`;
7. Override `SetRenderMaterial` method, so that it will set the provided material as the currently used render material. The following will do this for our line renderer: `GetExtendedComponent<LineRenderer>().sharedMaterial = material`;
8. And as the last step (though it's not required), we should specify what should happen when the effect is either disabled or removed from the game object. If we leave it as it is, the blend mode materials will remain to be assigned to the line renderer, but we would rather want to set the default material when the effect is `turned off`. To do this, override `OnEffectDisabled` method and set the default materials to the line renderer: `GetExtendedComponent<TComponent>().sharedMaterials = new [] { defaultMaterial }`.

You can find the complete example script on GitHub.

You may also use the built-in component extensions as a reference, when implementing your own. Find currently installed component extensions at `.BlendModes/Extensions/Components`.

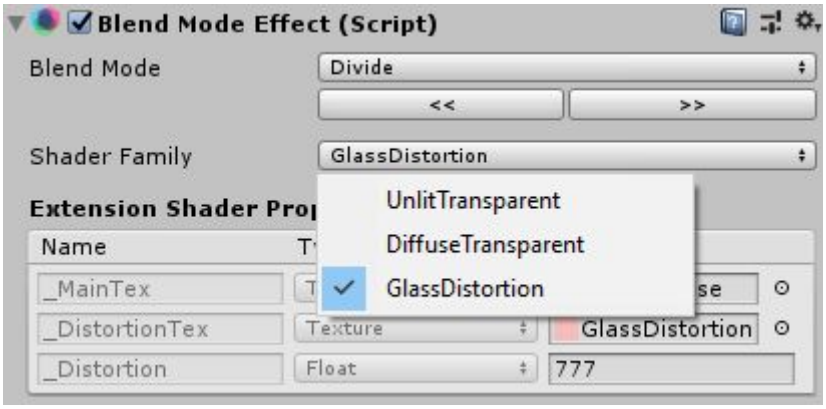
Shader extensions

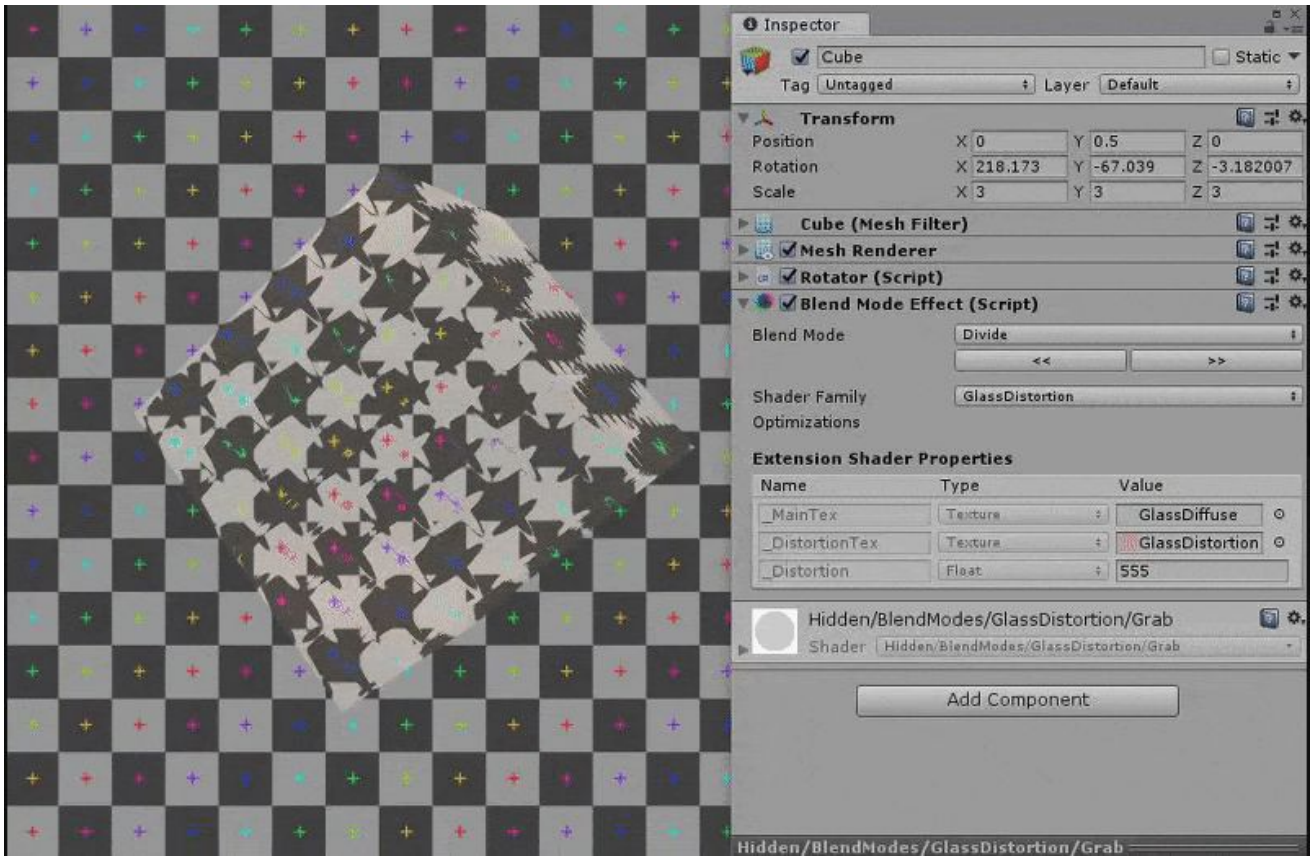
To work with any given shader family, Blend Mode Effect needs a corresponding shader extension. A shader extension consists of multiple shader assets. When you first import the BlendModes package, all the shader extensions are not installed by default to save compile time for the shader families you may not need at all in your project.

Adding shader extensions

For example, let's create a `GlassDistorion` shader family, add a shader variant, which will allow it work with the `Self With Screen` render mode and add support for this new shader family to the mesh renderers:

1. First, we need to add `GlassDistorion` to the list of the available shader families for the mesh renderer component extension. While we could directly edit the script in the package sources, that's not practical, as it would cause conflicts when updating the package. Instead, we will create a stand-alone script in our project, that will extend the `MeshRendererExtension` and add the shader family there;
2. Create a `GlassMeshExtension` script and inherit it from the `MeshRendererExtension`. Add `[ExtendedComponent(typeof(MeshRenderer))]` attribute so that the extension system will be able to find it as the extension for the `MeshRenderer` type.
3. Override the `GetSupportedShaderFamilies` method and add `GlassDistortion` to the list of the available shader families;
4. Now, we need to create the shader itself. Create a `Assets/Shaders/GlassDistortionGrab.shader` asset. Exact name of the asset and location doesn't matter, though, but we need to remember the path, as it will be required later;
5. Let's use the `Glass-Stained-BumpDistort` default shader as the base. To add blend mode effect support to this shader, we have to insert several pre-defined expressions. Use [this example source] to find out the exact modifications (each modification has an accompanying comment);
6. Notice the name, that we used for our shader: `Hidden/BlendModes/GlassDistortion/Grab` — it's mandatory to use specific names for the extension shader, in order for the extension system to find them and associate with render modes. See `[Extension shader names]` for details;
7. Next, we need to make sure, that the extension system knows where to find our shader. Select the `./BlendModes/Resources/BlendModes/ShaderResources` asset (it's automatically created when you install a shader extension for the first time) and add `Assets/Shaders` to the `Additional paths` list, then click `Update shader resources`;
8. `GlassDistortion` should now appear in the list of the available shader families when Blend Mode Effect is used with a mesh renderer.





Notice, that neither render modes, nor optimizations and masking nodes fields are visible when our newly created shader family is selected. That's because we've only created shader variant for the `Self With Screen` render mode. To add all the missing variants, we have to create 7 more shaders. You can use the built-in shader extensions as a reference for implementing each of the shader variants. Find currently installed shader extensions at `.BlendModes/Extensions/Shaders`.

You can find the complete example script and shader on GitHub:
github.com/Elringus/BlendModesTutorials/blob/master/Assets/Scripts/GlassMeshExtension.cs
github.com/Elringus/BlendModesTutorials/blob/master/Assets/Shaders/GlassDistortionGrab.shader

Extension shader names

Extension shader name is used to associate it with specific shader family and render mode: Hidden/BlendModes/**FamilyName**/**Mode**. While `**FamilyName**` could by anything (but unique amongst other existing shader families), `**Mode**` should have one of the following predefined values:

Grab	`Self With Screen` render mode
UnifiedGrab	`Self With Screen` render mode + `Unified Grab` optimization
Framebuffer	`Self With Screen` render mode + `Framebuffer` optimization
Overlay	`Texture With Self` render mode
GrabMasked, UnifiedGrabMasked, FramebufferMasked, OverlayMasked	Same as above + masking support

Scriptable render pipelines (SRP)

The SRPs are still hardly production-ready (despite the official claims from Unity) and lack many features compared to the default rendering system. It's not recommended to use the render pipelines, unless you're an advanced user and ready to solve any potential technical issues and limitations.

URP (universal render pipeline, aka LWRP) is supported with some limitations (more info below) due to lack of Grab Pass and multipass shaders in general.

HDRP is not supported out of the box, but you can add a custom pass to make it work; feel free to use `BlendModes/Runtime/RenderBlendModeEffect.cs` script (custom pass for URP/LWRP) as a reference when creating a custom pass for HDRP. Be aware that we're not providing any support for using the effect under HDRP.

Limitations

Following are limitations when using Blend Modes with URP/LWRP:

- `Self With Screen` mode will always work with `Unified Grab` optimization enabled (instances of the effect won't "see" each other).
- Masking is not supported.
- Camera (post-processing) effects are not supported, but you can fake them with a full-screen UI image using the blend mode effect.
- UI objects will correctly render only when either `Screen Space - Camera` (with a selected camera) or `World Space` canvas render modes are used for the parent canvas, otherwise the screen texture will render upside-down; UI sorting won't work.
- Only the special LWRP-compatible shader families are supported (more info below).

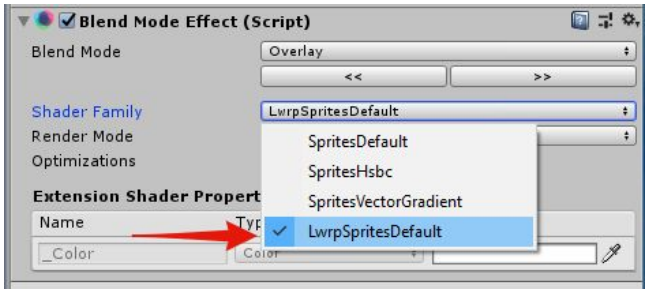
Known Issues

Following are the known issues when using Blend Mode effect under scriptable render pipeline.

- Object with the effect applied may not be visible under scene view. This is most likely related with a Unity bug and should be fixed in the next Unity releases.

Setup

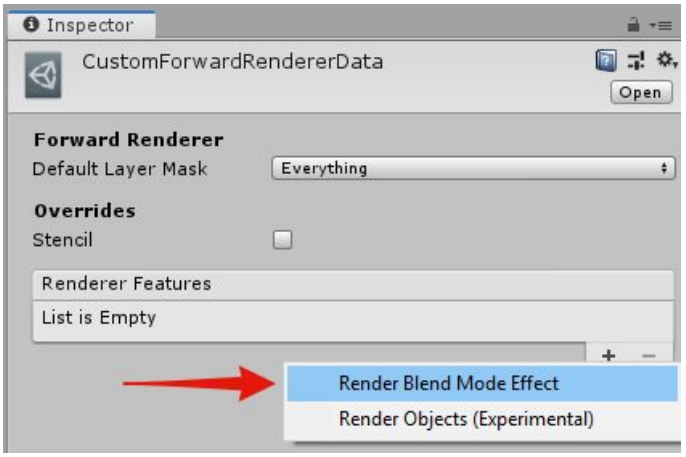
When URP/LWRP is installed and enabled in the project, `Shader Family` property of `Blend Mode Effect` will contain shader families with LWRP support (prefixed by **Lwrp**):



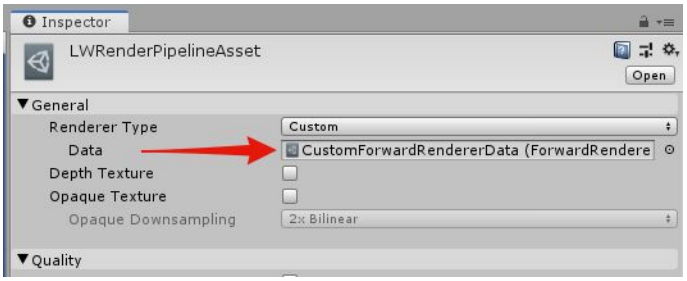
Only the special LWRP shader families will work with SRPs. It's also required to add a special renderer feature in order for the LWRP shaders to work with the pipeline:

1. Create `ForwardRendererData` asset with `Create -> Rendering -> Lightweight Render Pipeline -> Forward Renderer`;

2. Select the created asset and add `Render Blend Mode Effect` renderer feature:



3. Override the renderer type used in LWRP asset with the created `ForwardRendererData` asset:



Compatibility

Unity version

The plugin is compatible with Unity ver. 2019.1 or higher. When imported for previous Unity versions, a legacy version of the plugin will be used (ver. 3.17 for Unity 2018.x and ver. 2.15 for Unity 4.x-2017.x). Documentation for the legacy version of the plugin is available here: https://goo.gl/CXo6RN

Due to a regression issue introduced in Unity 2018.3.0a4, when built with IL2CPP scripting backend under .NET 4.x scripting runtime Blend Mode Effect script will fail to work. The issue was fixed in 2018.3.5f1 patch.

Scriptable render pipelines (SRP)

URP/LWRP is supported with some limitations, HDRP is not supported; check the docs for more info.

Tested platforms

Standalone (PC/Mac/Linux), VR (both multiple and single pass stereo rendering), WebGL, iOS, Android and UWP (IL2CPP scripting backend only).

Shader keywords

The plugin uses 25 shader keywords (one for each blend mode), while Unity provides 128 in total. Make sure your project has enough “free” keywords before importing the package.

Materials

When a blend mode effect is applied to an object, object’s material is managed by the plugin and shouldn't be changed manually. In case you wish to use a special shader with the blend mode effect, you can either pick one of the existing shader families or create your own shader extension.

Support and feedback

If you need support or wish to provide suggestions, feel free to PM me on the Unity forum. I will usually respond within an hour if you contact me from 9.00 until 21.00 UTC.

Changelog

Version 3.10

- — Exposed `Stencil ID` property when masking is enabled
- — Fixed compatibility with universal render pipeline

Version 3.9

- — Improved shaders compatibility with Unity 2019.3
- — Fixed "not supported exception" thrown by "BlendModeEffect" component in IL2CPP builds
- — Fixed a memory leak when changing blend modes or destroying objects affected by the effect
- — The fixes are backported for Unity 2018 version of the package

Version 3.8 (Unity 2019.1 and later)

- — Added support for lightweight scriptable render pipeline
- — Added assembly definition file to the package
- — All the component extensions are now installed by default; it's no longer required to manually install them for each object
- — Fixed HSBC shaders compilation under PS4
- — Fixed `UnlitTransparent` shader family zwrite issue

Version 3.7

- — It's now possible to stack multiple `Blend Mode Effect` components on cameras
- — Fixed `OnValidate` of `Blend Mode Effect` component being invoked recursively in Unity 2019.1
- — Fixed `ParticlesAdditive` shader family making the output color 2x brighter
- — Exposed `_Color` property of `SpritesVectorGradient` shader family as a temp workaround for controlling vector sprite tint color (`Color` property in the `Sprite Renderer` doesn't work atm)

Version 3.6

- — Added main texture scale (tiling) and offset default extension shader properties to the mesh and skinned mesh component extensions
- — Added color tint property to `UnlitTransparent` shader
- — Fixed `Texture Wish Self` render mode ignoring overlay texture alpha channel

Version 3.5

- — Added tilemap renderer component extension
- — Exposed API to set custom shader properties at runtime
- — Fixed an issue with framebuffer particle shader variants in Unity 2018.3
- — Fixed an issue with Text extension not working without UI extension installed

Version 3.4

- — Component extensions now use a single render material instead of an array
- — Exposed "ScreenUV" property of the camera component extension
- — Fixed an issue with missing standard particle shader in Unity 2018.3
- — Fixed an issue with shader extension properties not being serialized when using prefabs
- — Fixed an issue when UI material fails to update while using UI masks

Version 3.3

- — Added "HSBC" (hue, saturation, brightness, contrast) shader families for sprites, UI, cameras and particles
- — Added "SpritesVectorGradient" shader family to support vector sprites with gradients
- — Extension shader properties GUI will no longer draw when none of the properties are valid for the currently selected shader family
- — Fixed Unity 2018.3 compatibility issues

Version 3.2

- — Reduced `Blend Mode Effect` initialization time in some cases
- — Optimized memory allocations
- — Fixed issues when adding shader extensions from a non-package folder
- — Shader resources will now be automatically preloaded when the app starts

Version 3.1

- — Implemented materials caching and sharing
- — Grouped editor resources in a separate folder
- — Fixed issues with managing shader extensions under MacOS

Version 3.0

- — Shaders and editor code overhauled
- — It's now possible to extend supported components and shaders via public API
- — Most of the native renderers are now supported by default; added support for TextMesh Pro UI components
- — Masking system is now compatible with the native Unity masks
- — Only the shaders actually used in the project are now included to the build, which reduces compile times and build sizes