FlexibleBlur Manual

Contents

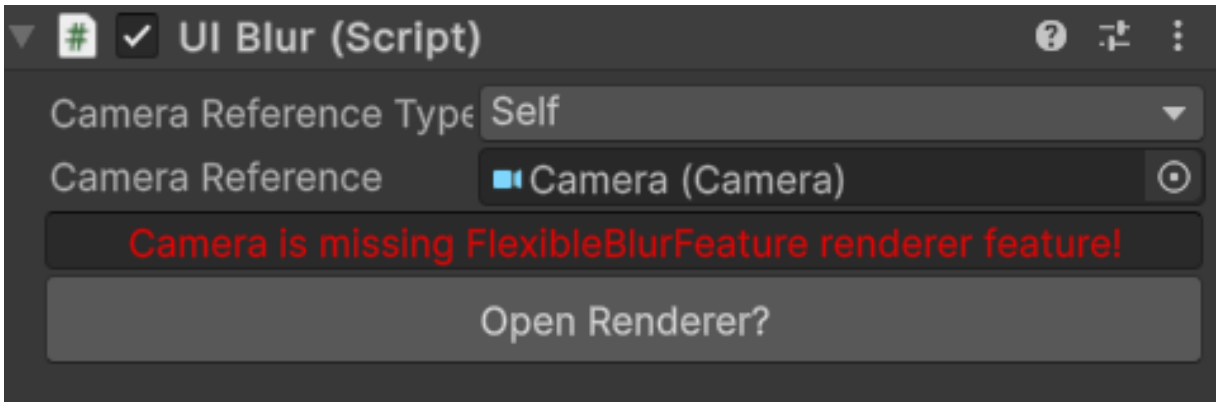
Setup

FelxibleBlur is only supported in the URP renderer, and is not supported in BiRP or HDRP.

The simplest way to add blur to a UI is to place UIBlur and/or BlurredImage components on a canvas. Those components reference a Camera to receive blurred output from, and to accomplish that the Camera's renderer should include the FlexibleBlurFeature.

The difference between UIBlur and BlurredImage components is that UIBlur draws directly onto camera output. BlurredImage reads/writes to a separate render texture which is then used in a UI material.

When blur components reference Cameras that are not properly set up, they will clue you in to the Renderer that needs to be adjusted, as so:



For Screen-Space Overlay Canvases: For a "Screen-Space Overlay" canvas (the default Canvas type), in the general use case, all blur components should reference the topmost camera

For other Canvas Types: For the "Screen-Space Camera" or "World Space" canvas types, the setup may be more involved. When a blur component is added via the Unity editor, the camera reference is automatically populated based on common usage. When the camera reference is empty, the component will attempt to use Camera.main.

BlurredImage components should be set up in such a way that:

a) At the point in time the blur is processed, the Camera can see everything that is supposed to become blurred, and

b) At the time the blur is processed, the Camera should *not* be able to see the blur that was processed on the previous frame. If the previous blur is visible, the results accumulate, and a feedback loop results which blows out the output.

UIBlurs on the other hand are lightweight components that render directly to the camera. Whatever point they render at, the blur from the previous frame has already been cleared. Thus, they do not create positive feedback.

The benefit of the BlurredImage components is that it is more feature rich and behaves like normal UGUI Image components. This includes all the benefits of an Image component, like masking and sprite slicing. For technical reasons, BlurredImage components support batching while UIBlurs do not (more on batching later).

There are a number of ways to set up the scene depending on what you're trying to achieve, but general rules of thumb to follow are:

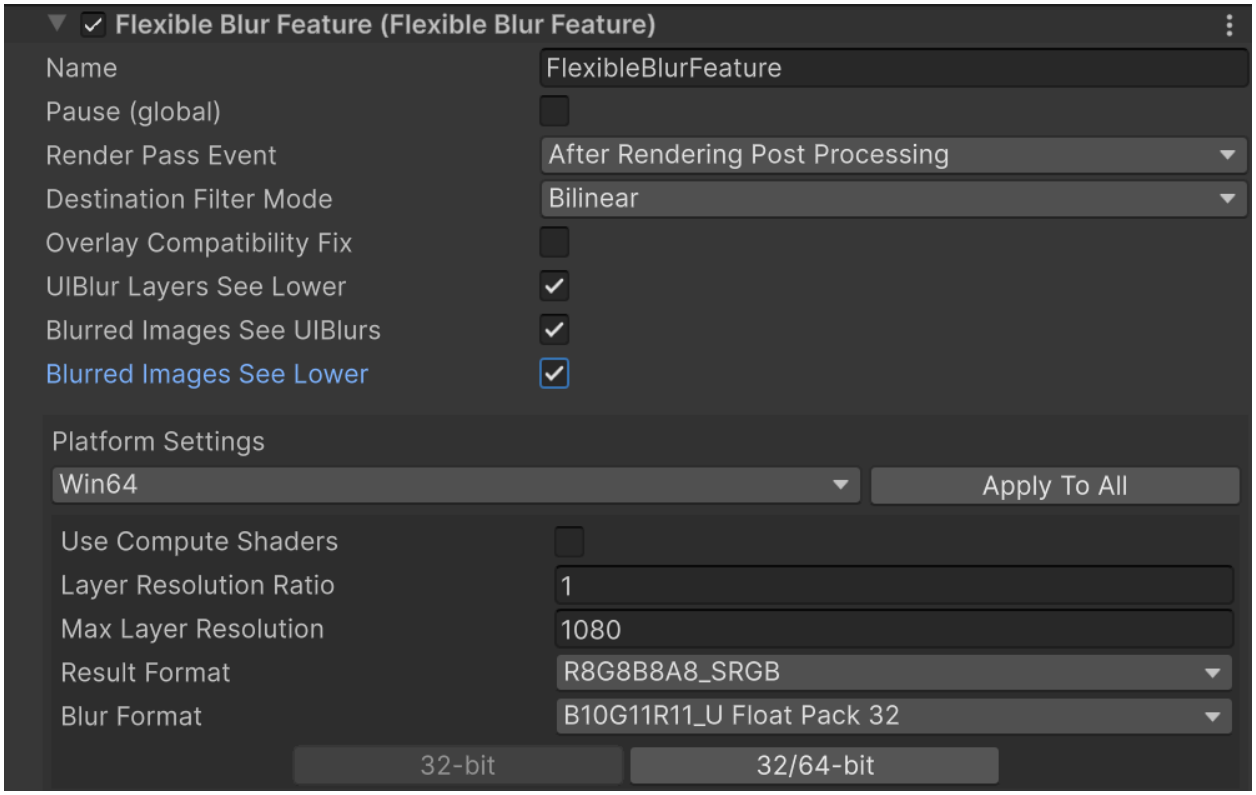
For *BlurredImage* components:

- We'll refer to the Camera that can see the canvas the BlurredImage component is placed on as the "Canvas Camera". This is the "Render Canvas" in a "Screen Space - Camera" Canvas. For a World Space Canvas, it is whichever Camera the Canvas is visible in (generally the one that contains the UI layer in its culling mask).
- **Never** set the BlurredImage "Camera Reference" to a Camera that is rendered **above** the Canvas Camera in such a way that it can see the canvas. This will create feedback.
- If the "Camera Reference" is rendered **before** the Canvas Camera, then the BlurredImage component will never create feedback.
- **If you set "Camera Reference" to the Canvas Camera**, then in the appropriate FlexibleBlurFeature you **must** set the RenderPassEvent to "Before Rendering Transparents" or lower to avoid feedback *or* use a **Render Objects** feature to force the canvas to render *after* the FlexibleBlurFeature (described later).

For UIBlur components:

- Anything that works for BlurredImage should also work for UIBlur components.
- The main difference is that, for any situation where BlurredImage would be susceptible to positive feedback, UIBlur will not experience any feedback, but instead will render **above** every other element in the Canvas, which may be undesired, but is also a way to achieve stacked, UI-preserving blur.

The Flexible Blur Feature



Make sure that all cameras that are referenced by blur components are set to a Renderer that contains a FlexibleBlurFeature. The feature has negligible cost when included but unused. When it is needed but not included, affected UIBlur and BlurredImage components will guide you through adding it in their inspectors.

Render Pass Event: What stage of the rendering pipeline blurs are drawn in. If set to “After Rendering Transparents" or above, BlurredImages should reference a Camera *below* their Canvas Camera to avoid accumulation each frame and getting “blown out”. BlurredImages that are children of “Screen Space - Overlay” Canvases, as well as UIBlurs generally, should never blow out.

Destination Filter Mode: Whether the final blur result will have bilinear or point filtering. At or near native resolution, there is little visual difference. With a low “Max Layer Resolution,” point filtering can be used for a pixellated look.

Overlay Compatibility Fix: When enabled, uses a less optimized but more compatible blit method for overlay canvases. When disabled, UIBlurs may be off by a pixel or two in some edge cases. Leave disabled unless you notice an issue with UIBlurs.

UIBlur Layers See Lower: When enabled, UIBlur layers stack so that higher layers blur the results of lower layers. Costs one blit per layer, after the first layer.

Blurred Images See UIBlurs: When enabled, BlurredImages will blur the results of UIBlurs. No significant performance difference, but occasionally adds another render texture.

Blurred Images See Lower: When enabled, BlurredImage layers stack so that higher layers blur the results of lower layers. Costs one blit and requires an additional render texture per layer, after the first layer.

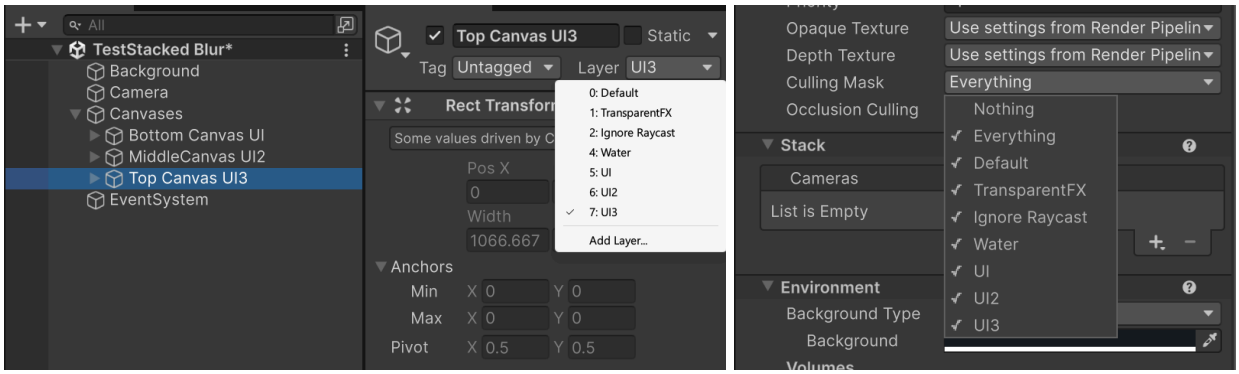
Platform Settings: *Settings that are adjusted for individual platform build targets.*

> **Use Compute Shaders:** Compute shaders are generally faster on modern hardware and don't create traditional draw calls, but there is still overhead to calling compute shader functions. Anything that increases draw calls in the fragment path will also have some (generally lesser) cost in the compute shader path. The fragment path may be more compatible, and may be faster on architectures with poor compute performance. Note that this option only affects blur functions, while other functions (like blitting) still use traditional shaders.
>
> **Layer Resolution Ratio:** The size of each blur layer's result texture, relative to the camera resolution. For example, 2160p camera resolution with a 0.5 ratio yields a 1080p blur layer result texture.
>
> **Max Layer Resolution:** The absolute maximum resolution for blur layer result textures. Generally, there isn't much need for this to be higher than 1080p (and at 4K+, these textures can become expensive). The one downside to having the blur layer result textures a different resolution than the camera is that a faint outline may appear at the edges of UIBlurImage components (while BlurredImages should not be affected).
>
> **Result/Blur Format:** The color format that is used for RenderTextures created by the feature. Result Format pertains to the Render Textures that contain the ultimate result that is read from/written to by BlurredImages, while Blur Format pertains to the temporary Render Textures that are created when downscaling and performing blur operations. Buttons along the bottom set the formats to common defaults.

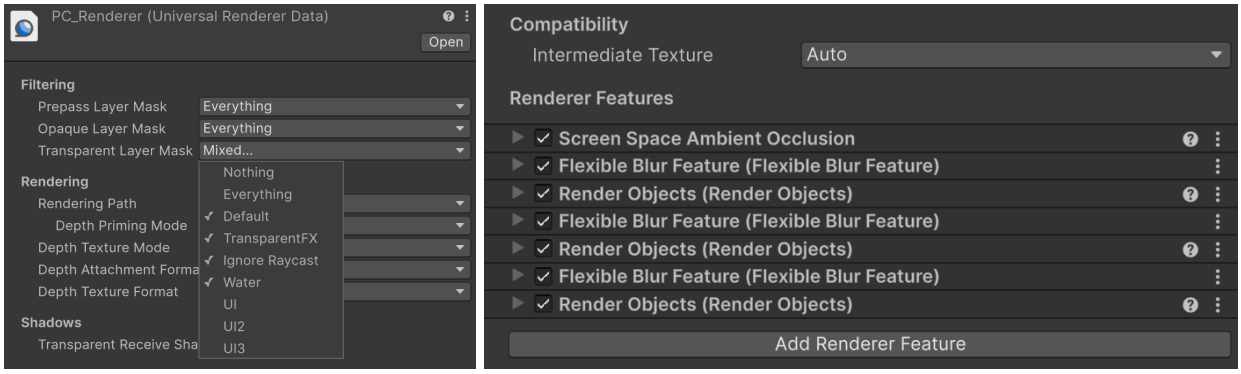
Using Multiple Flexible Blur Features (advanced)

It is possible to add multiple Flexible Blur Features onto the same Renderer (UniversalRendererData). Typically, this is done so that the output of one feature can be seen and processed as input by the next feature (with a feature such as **Render Objects** placed after Flexible Blur Features to control the drawing order of canvases), but can also be used in the case that blurred elements require different feature settings.

As an example of how to create a single camera, triple stacked UI-preserving blur setup, create three Screen Space - Camera canvases, and assign each its own Layer. Make sure the camera includes all UI layers in its culling mask, and all canvases are set to the same camera as their Render Camera:

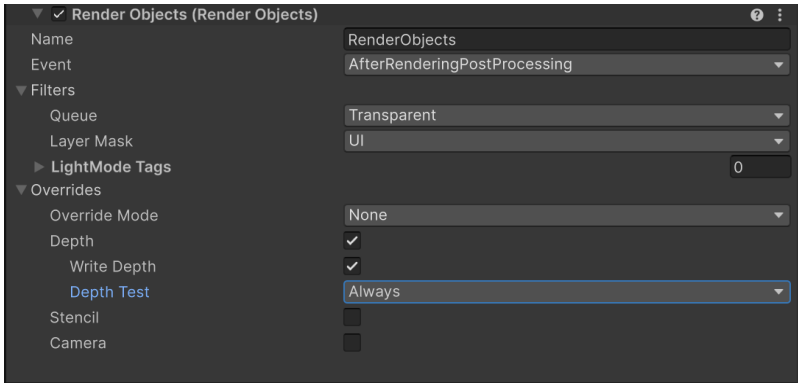


In the applicable Universal Renderer Data object, remove all UI layers from the Transparent Layer Mask. For each UI layer, add a Flexible Blur Feature followed by a Render Objects feature:

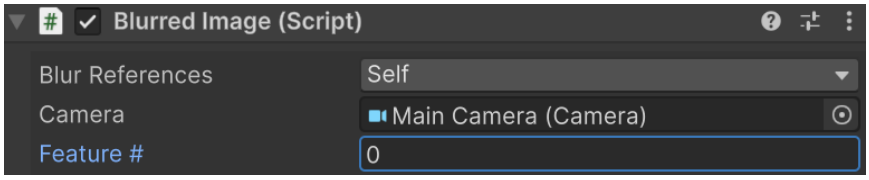


All features *must* be set to a Render Pass Event >= that of the previous feature. Aside from that, Flexible Blur Features require no additional settings. Render Objects have a very specific setup, however.

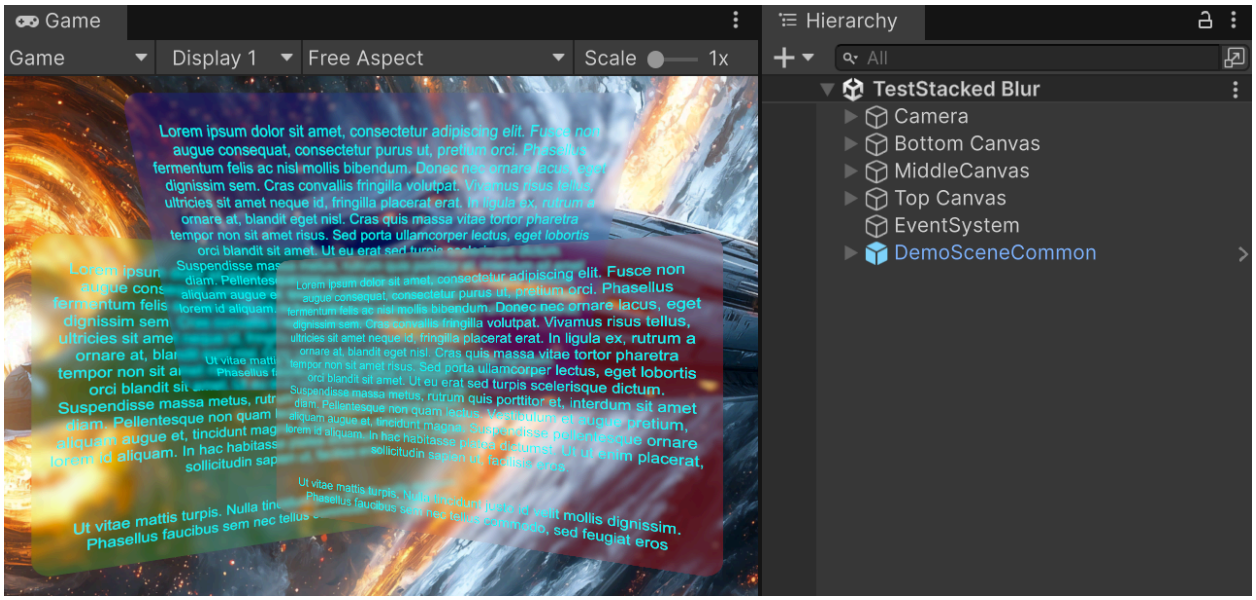
Set Render Objects to the Transparent Queue. Each feature's Layer Mask will correspond to the UI layer being drawn. In our case, the first Render Objects feature will be set to the UI layer, the second to the UI2 layer, and the third to the UI3 layer.
It is good practice to override Depth to "Always," so canvases will not be occluded by other objects in the scene:



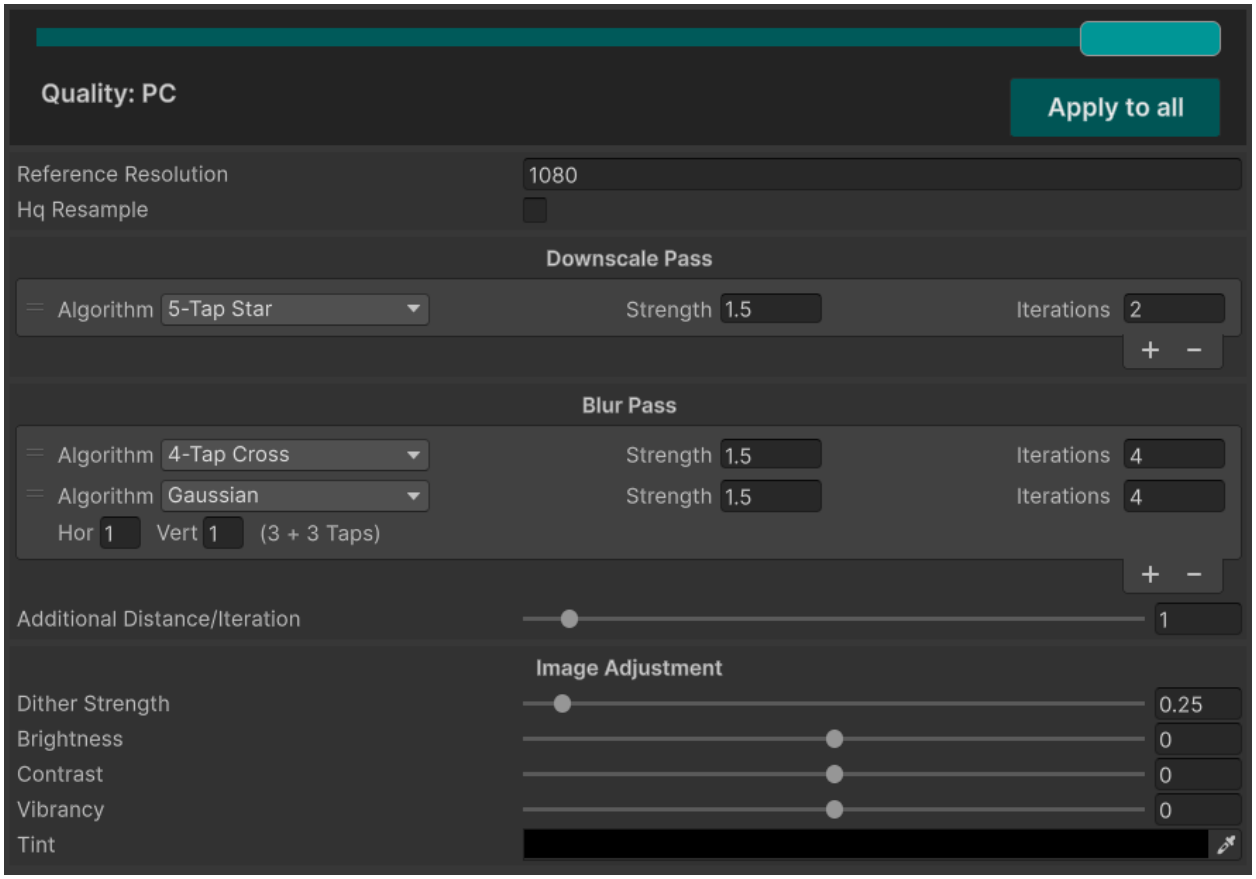
Make sure that blurred elements are referencing the correct feature. When there is more than one feature available, a "Feature Number" field will become visible on UIBlur, BlurredImage, and BlurReferenceProvider components:



Final result:



Blur Presets/Blur Settings



*Blur Presets are used to control the blur settings of many blur components from a common place. They are the only means by which blur settings can react to quality level. BlurredImages that share a preset can be **batched**. Tweaking these settings can be instrumental in achieving a specific style, at high quality level, with low performance cost. This flexibility can also be a footgun—8k reference resolution with no downscaling and 32 blur iterations is overkill.*

Reference Resolution: One of the most important settings. Controls how blur areas are initially resized, which allows blurs to appear consistent regardless of application resolution. If set to 0, then no resizing is performed, which means blurs will appear weaker when application resolution increases. On the other hand, if reference resolution is set to “1080” then for any application resolution below 1080p the blur will be upscaled to the size it would be on a 1080p display, creating extra work. It is not a bad idea to set this to (or near) the lowest common resolution you expect to be used, and it’s generally advised to err on the side of lower values. You **probably never** want to set reference resolution higher than 2160. The trade-offs to setting this value low is that it can make it difficult to accomplish more “subtle” blurs, and lower values can exacerbate temporal noise.

HQ Resample: Applies a 7-tap hexagonal filter when resampling to the reference resolution. This smooths out the result and drastically reduces temporal noise when paired to a low reference resolution. It is advised to leave this disabled when paired to a high reference resolution, as it makes little difference to stability while costing performance.

Downscale Passes: Each downscale pass halves the resolution of the blur, meaning subsequent downscale and blur iterations only touch ¼ as many pixels. Thus, one or two downscale iterations can drastically reduce the number of texture samples.

Blur Passes: Performed after downscale passes. The **Additional Distance/Iteration** setting controls how much the strength (the distance that texture samples are taken) is increased with each subsequent iteration.

Common Properties of Downscale/Blur Pass Elements

Algorithm:
As of this writing, there are 9 different blur algorithms to choose from. “Tap” in the algorithm name refers to the number of texture samples that are taken.

3-Tap Checkerboard, 4-Tap Corners, 4-Tap Cross, and 8-Tap Corners + Cross are **not** recommended for downscaling because they do not sample the central texel, and thus are more prone to temporal noise.

Quadratic and Gaussian are configurable-tap dual-pass separable algorithms. This means that, for example, a 3x7 tap Gaussian kernel samples 3 horizontal texels and 7 vertical texels, with both horizontal and vertical passes needing their own draw calls (it is possible to skip either the horizontal or vertical pass by specifying 0 taps). Besides being very high fidelity, these algorithms are useful when you want to apply more blur along one dimension than the other.

Strength: The distance that texels are sampled at. It is usually a good idea to set this to x.5, rather than x.0, which helps take advantage of GPU texture filtering.

Iterations: The number of times to apply the algorithm. Each iteration adds another draw call, or two for separable algorithms. This asset uses a novel optimization where most algorithms, and separable algorithms especially, are more efficient every other iteration (eg. 2nd, 4th, 6th). Full details are beyond the scope of this document.

Last Section

Dither Strength: How much dithering to apply to the blur. This has the effect of smoothing out banding artifacts, which may manifest with stronger blurs, or when the region being blurred has very little variance in color, or when using lower precision texture formats.

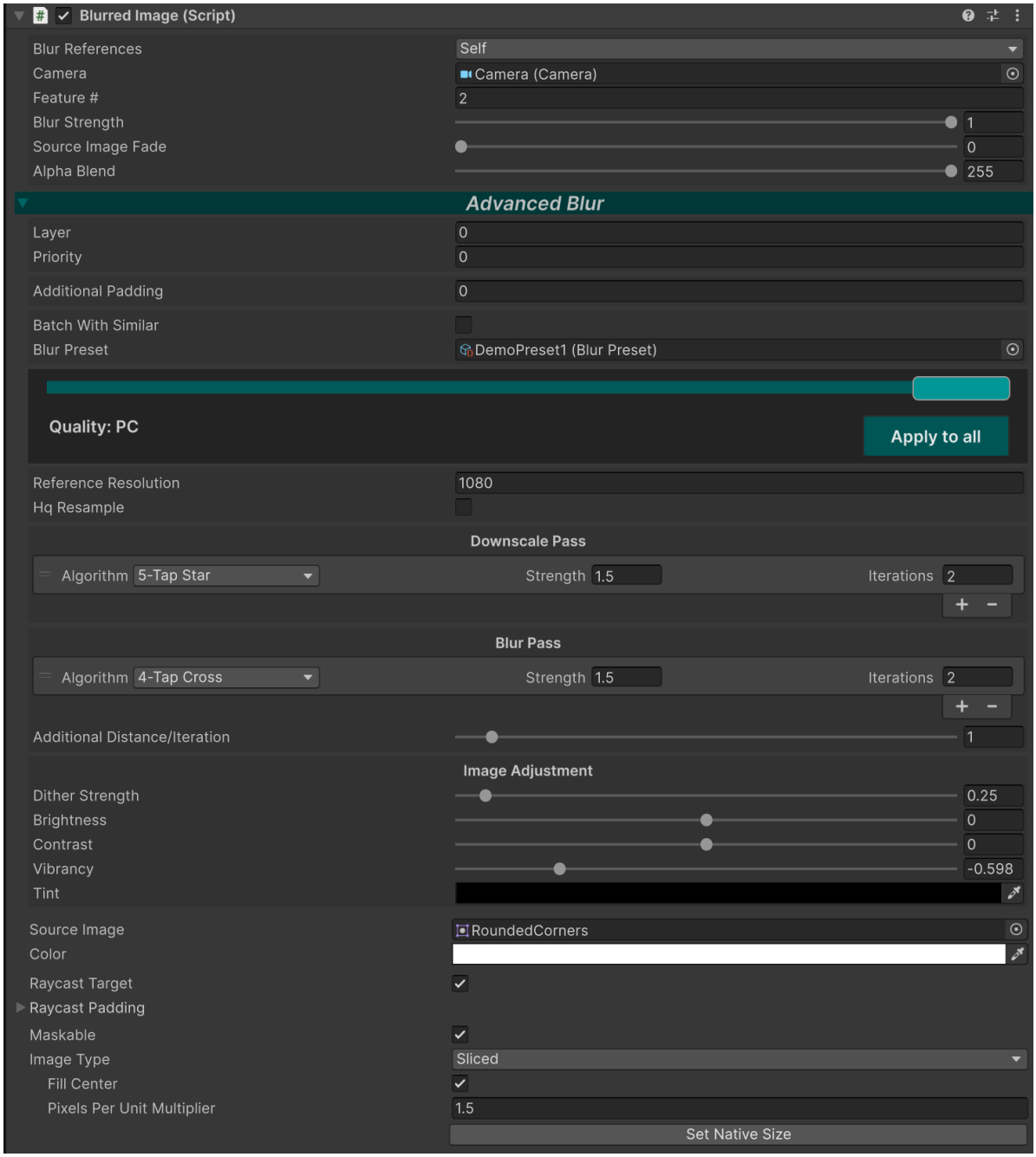
Brightness: Can set between (-1,1) to make the output darker or brighter.

Contrast: Can set between (-1,1) to make the output lower or higher contrast.

Vibrancy: Can set between (-1,1) to make the output more desaturated or vivid.

Tint: Tints the final output, with the alpha channel controlling the strength of the tint.

The BlurredImage Component



BlurredImage is derived from the standard Image component. Instead of writing directly to the screen like UIBlur, the FlexibleBlurFeature looks at all the BlurredImages which reference the same camera and saves the result to a RenderTexture, which BlurredImages use in their output to the UGUI Canvas. It is more heavyweight than UIBlur, but more flexible, supporting Image features and batching.

Blur Strength: Control the overall strength (sample distance) of the blur effect. For static blurs (blurs that don't change strength), it is **vastly** preferable to leave this at 1

and instead modify Blur Settings (reference resolution, downscale, blur passes, etc.) to achieve the desired outcome. FlexibleBlur components with different blur strengths can never be batched.

Source Image Fade: Controls how visible the source image (sprite) is. At 0, the only element the source image contributes is its alpha channel. At 1, the source image is fully visible and completely obscures the blur.

Alpha Blend: This setting adjusts how the blur interacts with the canvas and color alpha channel.

When Alpha Blend is set to 1, the blur's strength on the RenderTexture remains constant. Instead, as the canvas or image alpha nears 0, the entire blurred image fades out, similar to a standard Image component—blending with everything beneath it. This creates a "hazy" fade effect that might not always look ideal, but it allows UI elements behind the blur to become visible as it fades.

Conversely, when Alpha Blend is 0, the component doesn't blend with the pixels below at all. Instead, the blur's strength (sample distance) reduces to 0 (via the same mechanism as **Blur Strength**). This is generally more pleasing, but means UI elements behind the blur stay hidden, even as it fades out.

Values between 0 and 1 interpolate between these two behaviours.

Advanced Blur Section

Layer: The order that groups of blurs render in. Depending on the setup of the FlexibleBlur renderer feature, higher layers may blur the results of lower layers, or simply render on top of other layers without taking into account their results.

Priority: The order in which blurs render inside a layer. For blurs that have the same layer and priority, blurs will render in the order that their components were initialized or enabled.

Additional Blur Padding: This setting expands the blur region on the RenderTexture by a specified number of pixels. It doesn't enlarge the blurred image itself—since it still pulls from the same RenderTexture area—but it can enhance temporal stability near the image edges. Beyond performance impacts, a drawback is that expanding the blur

region might unintentionally overwrite parts of lower-priority blurs on the same RenderTexture.

Use Preset: Whether to use a preset to get the settings for the blur, or to supply settings for that instance only. It's a good idea to use presets for all but one-off blurs.

Batch With Similar (*only available when a Blur Preset is supplied*): This groups the BlurredImage with other BlurredImages that share the same Camera Reference, Blur Preset, Priority, Layer, and canvas alpha*. It functions by identifying the topmost, bottommost, leftmost, and rightmost points of all batched images and drawing a single rectangular blur on the RenderTexture based on those boundaries.

*Only when AlphaBlend < 1, which changes the sample distance of the blur on the RenderTexture. If AlphaBlend == 1, then canvas alpha does not matter for the purposes of batching.

This is useful when you have many small BlurredImages that are using the same blur settings in a scene, especially when they are close together (eg. an inventory grid where each element of the grid contains a blurred BlurredImage). In such a scenario, the reduction in overhead and draw calls almost certainly wins out over blurring more pixels—often substantially.

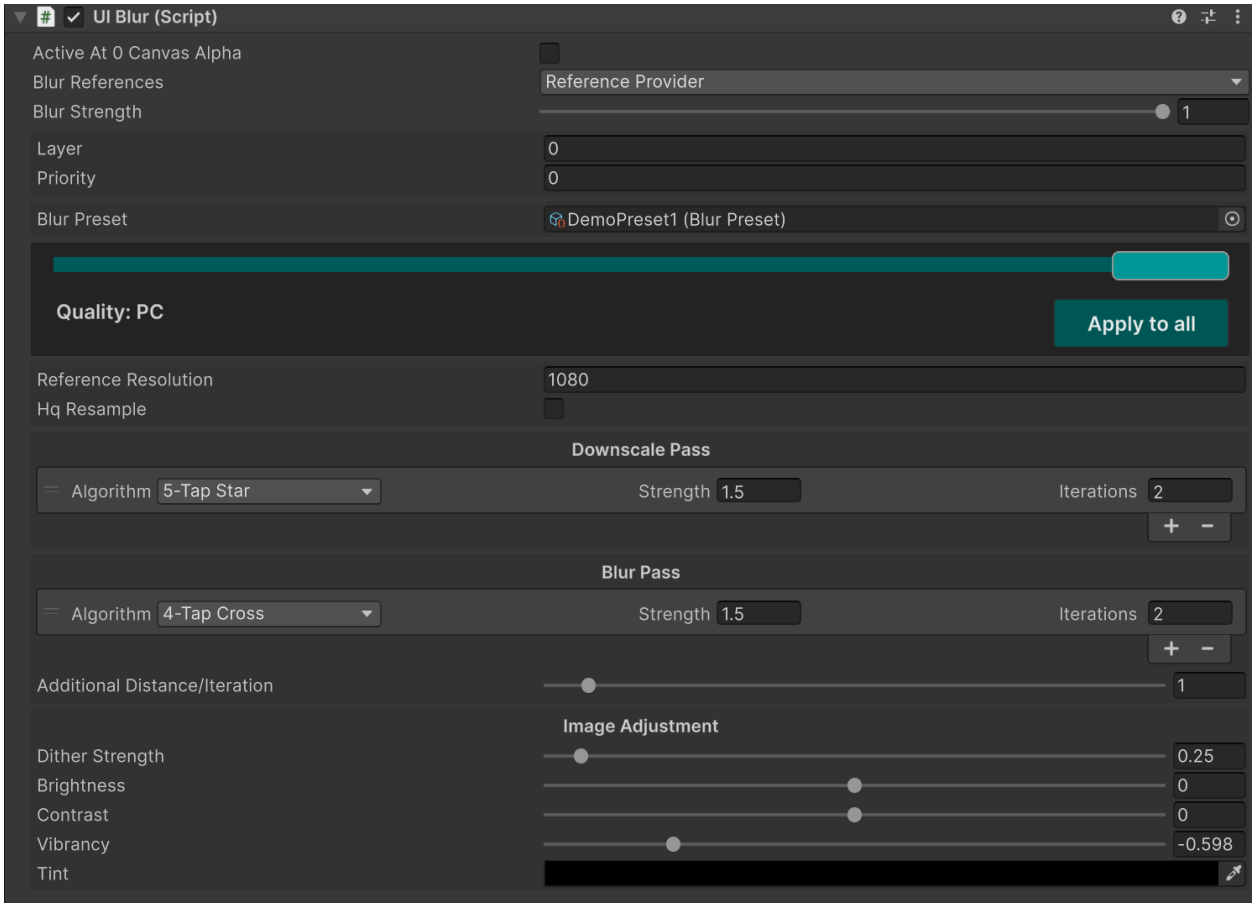
A downside is that more blurred pixels are written to the RenderTexture, which can overwrite any blurs that are lower in priority and placed between batched elements. If you have four batchable small BlurredImages on the corners of the screen, so that the area in between those BlurredImages is much larger than the are of the BlurredImages themselves, then that may not the best scenario for batching.

Fill Entire RenderTexture (*only available when Batch With Similar is selected*): Typically, batched BlurredImages write to the smallest rectangular area on the RenderTexture that encompasses all batched elements. When this option is enabled, the blur extends across the entire RenderTexture instead, aligning with the behavior of many other UI blur assets. This can enhance temporal stability, particularly when perimeter elements in the batched group are moving.

The trade-off is that blurring every pixel can significantly slow performance, especially if only a small portion of those pixels is ultimately used.

The remainder of BlurredImage inspector fields are detailed in the **Blur Preset/Blur Settings** chapter, or are common with UGUI Image components.

The UIBlur Component



UIBlur is the more lightweight and draws directly onto its CameraReference. Typically, this means it should render below everything else on the Canvas. UIBlur is a good choice when you only need to blur a rectangle that is below the rest of the UI, but lacks many features of BlurredImage.

Active at 0 Canvas Alpha: Forces the blur to be active even when the canvas alpha is 0 (eg. via CanvasGroup alpha). Mainly useful for when you're setting the UIBlur to copy the background without blurring, eg. inside of another UIBlur to achieve a "subtractive blur" effect.

The remainder of the UIBlur inspector is a subset of the BlurredImage inspector. Besides the fields that come from the base Image component, fields for **Source Image Fade**, **Alpha Blend, Additional Padding,** and everything to do with **Batching** are absent in the BlurredImage inspector.

How To Run Blur On a Potato

FlexibleBlur can scale to *very* low end hardware. But the corollary is that you can shoot your foot off with dumb settings too. It should be possible to run blur with acceptable performance on just about anything device *if* you make the right tradeoffs, which can include:

- Batch as many blurs as possible—use BlurredImage with the same blur preset, and toggle batching on. Alternatively, try to limit the number of blurs on screen at once.
- In your blur settings set the **Reference Resolution** to something low. For large blurs, you can go as low as 240p while still looking decent. Higher reference resolutions are mostly useful for light or subtle blurs. Keep in mind that 240p is one quarter as many pixels as in 540p, which is itself one quarter as many pixels as in 1080p, so this setting scales quickly!
    - In the FlexibleBlur renderer feature, set the **Max Layer Resolution** to a value that is similarly low.
    - Very low reference resolutions can add a lot of temporal noise. This can be largely fixed by enabling **HQ Resample**.
- In blur settings, try to make do with as few iterations as possible. Higher downscale values drastically reduce the cost of iterations, but downscale itself also has a cost. You can also use blur zero iterations and rely entirely on downscale to produce the blur effect.
    - You are performing a juggling act between draw calls and texture samples. You want both to be low, but both are also in tension. A single 65x65 tap gaussian blur does a lot in two draw calls, but is outrageously expensive in samples. Profile your hardware to see where the bottleneck is.
- In the FlexibleBlur renderer feature, experiment with compute shaders and texture formats. The former is likely to perform well when dealing with a weak implementation of an advanced GPU architecture. The latter can have a noticeable impact on performance, but also on quality.
    - It may be possible to counteract some of the quality loss from using lower precision texture formats by increasing the **Dither Strength** of blurs.

Troubleshooting

BlurredImages “bloom out”: When a BlurredImage “blooms out” it is due to the fact that the camera output being blurred contains the final output of the BlurredImage from the previous frame. Thus, blur is accumulated every frame leading to a runaway feedback. There are a number of possible solutions depending on what you are trying to achieve with your use case and performance budget. These may include:

•Change the CameraReference of affected BlurredImage, typically to one that is lower in the camera stack.
•Change the RenderCamera of the Canvas, typically to one that is higher in the camera stack.
•Change the Render Mode of the Canvas to “Screen Space-Overlay”.
•Convert the BlurredImage into a UIBlur. UIBlurs are written directly to the camera target rather than the UI itself, and in most cases are immune to feedback (right click the BlurredImage component title for a context menu that includes conversions).
•Change RenderPassEvent of the Flexible Blur Feature to “Before Rendering Transparents” or lower.
•Use “Render Objects” features to move canvas drawing to after the FlexibleBlurFeature execution.

UIBlur components look wrong on cameras that use the Solid Camera background type: Set the alpha channel of the background color to 100% (255).

Other issues: If experiencing an issue after adding the asset to your project, try restarting the editor to see if the issue resolves. For any issues not listed, or to discuss the asset for any reason, you can reach me at hurleybird@gmail.com.