Soft Mask

Actual for version 1.2

# Tutorial: Adding the Soft Mask support to your own shader

## Introduction

The major part of Soft Mask's job is performed by a shader that renders the masked elements. It means that Soft Mask needs to inject some logic into that shader. Soft Mask easily does it for the standard UI shader because it knows exactly what this shader does. In fact, Soft Mask just replaces the standard shader by its own one. But if you use custom shaders for UI, you need to adapt them by yourself because Soft Mask doesn't know anything about them and can't just quietly replace them.

To support Soft Mask, a shader should do just one additional thing: multiply the resulting pixel opacity by the mask value. Unfortunately, due to the way shaders are structured, you can't just add some code in a single place. Instead, you have to perform several small modifications through the entire shader. This tutorial will be your guide in this process.

## Step-by-step guide

Let's go through the entire procedure step by step. As an example we will use a real-world shader: the SDF shader from TextMesh Pro. It's pretty complicated, which lets us to see some subtleties that could occur in the adaptation process.

To have the shader at hand you can download a free version of TextMesh Pro from Asset Store. The file is named `TMP_SDF.shader` and by default it can be found in the folder `Assets/TextMesh Pro/Resources/Shaders`.

The Soft Mask package also contains an example of the support in a very basic shader. It located at the path `Assets/SoftMask/Samples/Materials/WaveWithSoftMask.shader`. You can check out how the steps described here were applied to it.

> This tutorial uses a TextMesh Pro shader just as an example. If you actually need to mask TextMesh Pro texts, you don't have to implement it by yourself, because it's already done. Just download and import the integration package from Asset Store. For more information see the section Using Soft Mask with TextMesh Pro in the Soft Mask documentation.

## Step 1: Declare that the shader supports Soft Mask

First of all, we should let Soft Mask know that our shader now supports masking. Soft Mask determines this by checking presence of the `_SoftMask` property. It should be a 2D texture with the solid white default value.

Let's add the following line to the shader parameters declaration:

```
    Properties {

        …

        _StencilWriteMask  ("Stencil Write Mask", Float) = 255
        _StencilReadMask   ("Stencil Read Mask", Float) = 255

        _ColorMask         ("Color Mask", Float) = 15

        _SoftMask          ("Mask", 2D) = "white" {}
    }
```

*Hereinafter added code will be marked bold.*

## Step 2: Add #pragma and #include

We need to include `SoftMask.cginc` file in order to use Soft Mask API. We also need to add a `multi_compile` pragma to support all possible variants of Soft Mask workings: Simple, Sliced and Tiled.

```
    SubShader {

        …

        Pass {

            …

            CGPROGRAM
            #pragma target 3.0
            #pragma vertex VertShader
            #pragma fragment PixShader
            #pragma shader_feature __ BEVEL_ON
            #pragma shader_feature __ UNDERLAY_ON UNDERLAY_INNER
            #pragma shader_feature __ GLOW_ON
            #pragma shader_feature __ MASK_OFF
            #pragma multi_compile __ SOFTMASK_SIMPLE SOFTMASK_SLICED SOFTMASK_TILED

            #include "UnityCG.cginc"
            #include "UnityUI.cginc"
            #include "TMPro_Properties.cginc"
            #include "TMPro.cginc"
            #include "Assets/SoftMask/Shaders/SoftMask.cginc"
```

You may note that yet another variant slot is used here (`__`). It's intended for a case when masking isn't used at all: an element isn't nested into a Soft Mask or Soft Mask is disabled. In this case, all Soft

Mask code expand to no-operation instructions (such as multiplying by 1) which should be eliminated by the optimizer.

If you know that in this particular shader you don't need all the variants, you can omit unnecessary ones. If you will try to use such shader on an element that is nested into a mask with an unsupported mode, errors will be shown in the console.

## Step 3: Add Soft Mask-related parameters to a structure that is passed from the vertex shader to the fragment shader

Soft Mask abstracts away which operations are performed in both vertex and fragment shaders as well as data that should be passed between them. This is achieved by using macro definitions.

To add required data to a VS-to-FS structure we use macro `SOFTMASK_COORDS(texCoord)`, where `texCoord` — index of the TEXCOORD interpolator which Soft Mask is allowed to use.

In the case of TextMesh Pro our additions look like this:

```
struct pixel_t {
    float4  position        : SV_POSITION;
    fixed4  color           : COLOR;
    float2  atlas          : TEXCOORD0;  // Atlas
    float4  param          : TEXCOORD1;  // alphaClip, scale, bias, weight
    float4  mask           : TEXCOORD2;  // Position in object space(xy), ...
    float3  viewDir        : TEXCOORD3;

#if (UNDERLAY_ON || UNDERLAY_INNER)
    float4  texcoord2      : TEXCOORD4;  // u,v, scale, bias
    fixed4  underlayColor  : COLOR1;
#endif
    float4 textures        : TEXCOORD5;
    SOFTMASK_COORDS(6)
};
```

As you can see, the maximum used interpolator index is 5 (TEXCOORD5), so we write `SOFTMASK_COORDS(6)`.

## Step 4: Perform necessary calculations in the vertex shader

After adding a slot to the VS-to-FS structure, we want to fill up this slot in a vertex shader.

To do it we will use `SOFTMASK_CALCULATE_COORDS(output, pos)` macro. It receives two arguments:

- An instance of the structure that we modified in the previous step. The macro calculates and fills up the added fields in the passed instance. The vertex shader should return this instance or its copy.
- Vertex position that was passed to the vertex shader.

In most cases use of this macro looks like this:

```
…

SOFTMASK_CALCULATE_COORDS(OUT, IN.vertex);
```

```
    return OUT;
}
```

Where OUT is a shader result and `IN.vertex` — the field in the input structure of vertex shader with POSITION semantic.

In the case of TextMesh Pro, things get a bit trickier because the output structure is initialized not by assignments but via the structural initializer. So, to make the code compile, we need something like this:

```
    pixel_t output = {
        vPosition,

        …

        float4(faceUV, outlineUV),
    #ifdef __SOFTMASK_ENABLE
        float4(0, 0, 0, 0),
    #endif
    };

    SOFTMASK_CALCULATE_COORDS(output, input.position)
    return output;
}
```

*Please note, that using of `__SOFTMASK_ENABLE` and exposing the actual data behind `SOFTMASK_COORDS(n)` is an abstraction leak and it is not recommended when other options available.*

## Step 5: Apply a mask in the fragment shader

Finally, to apply a mask, we need to multiply the alpha of the resulting pixel by the value calculated with `SOFTMASK_GET_MASK(input)`. The sole argument to this macro is an instance of the VP-to-FP structure.

In a common case this looks like this:

```
    half4 color = …

    color.a *= SOFTMASK_GET_MASK(IN);

    …

    return color;
}
```

But TextMesh Pro uses premultiplied alpha (i.e. blending mode is `Blend One OneMinusSrcAlpha`) which means that we should multiply not only alpha but color channels too:

```
     …

    return faceColor * input.color.a * SOFTMASK_GET_MASK(input);
}
```

That's it! We've added the support of Soft Mask. Now it's time to check that our shader compiles and works.

## Troubleshooting

It's not hard to make a mistake during this process. In this case, a masked element may be displayed incorrectly or not displayed at all. Here are some tips that can help you to figure out what is going wrong:

1. Ensure that the shader doesn't have compilation errors. The surest way to do so is to choose the shader in the Project Tree and look at the Inspector.

2. If you use `#include` not only for SoftMask.cginc and don't see any changes as you modify the code, try to re-import both included files and your shader. Some Unity versions had an error that sometimes caused modifications of .cginc files didn't update dependent shaders.

3. To determine whether a mask is sampled correctly, try to replace the fragment shader return by something like `return half4(SOFTMASK_GET_MASK(IN), 0, 0, 1);` This should draw visible parts of an element in red and masked ones leave invisible.

4. To evaluate that mask works well, don't forget to move a masked element to the area where masking is really applied. For example, if the element is inside a scrolling list, move it to the container border.

## Your feedback

...is highly appreciated! If you have any questions or suggestions on how to improve this tutorial, please post them in the Soft Mask forum thread or email me at knyazev751@gmail.com.