Custom Decal Types

The system comes with a pretty flexible array of decals out of the box, but there will always be edge cases that need something special. The system is capable of being extended with custom decals that use their own shaders, though you will require a solid understanding of vert/frag shaders and Unity's rendering pipelines if you wish to take this route and implement your own.

The process of implementing a new decal type involves two stages. Creating the "projection", and creating the shaders the projection will use. It's usually best to start with the requisite shaders, then build the projection based on what the shaders require.

Writing Decal Shaders

First of all, we need to determine which shaders you need to write. A projection can require up to 9 different shaders. A set for each shader type (Mobile, Standard, Packed); consisting of 1 forward, 1 deferred opaque and 1 deferred transparent shader each. Luckily, unless you plan on using all 3 shader types for different platforms, and require both deferred and forward rendering, you will likely only need 1 or 2 shaders. To begin, determine which shader method you're using (If you're not sure you're likely using standard) and determining whether you're using forward or deferred rendering. If you're using forward rendering, or want to render your decal in forward rendering, you'll only need to write the one shader, if using deferred you'll likely need both an opaque and transparent variant (To facilitate blending the alpha of the spec-gloss buffer) though this is optional.

As there are so many variations on your starting shader setup, I'm simply going to suggest copying an existing shader within the package. So navigate to the systems shaders (Dynamic Decals > Resources > Shaders > -your shader model- > Metallic) and pick either a forward or deferred opaque to start working with. This will give us a basis for the passes, tags etc. the shader will require.

From here give your copy a new name; I suggest using the existing naming conventions. Also, make sure you keep your new shader within a Resources folder, so it can be loaded by the system at runtime; as well as modify the cginc. paths to find the requisite includes. Now we can start hacking it apart. There's a lot to cover here, so we'll break it into 3 sections, the properties, vert program and fragment program, and use the includes as a base to work from (Dynamic Decals > Resources > Shaders> Cginc).

Properties

Your shader can have whatever properties you need; as long as you keep the following properties to have the shader function correctly:

_Cutoff (Used to clip pixels, though you will need to implement alpha clipping yourself).
_NormalCutoff (Used for angle limit clipping).
_TilingOffset (Used to offset the UVs).
_MaskBase, _MaskLayers (Used for masking).

Vert Program

The existing vert structs and programs can be found in the DeferredProjections.cginc and ForwardProjections.cginc respectively. Everything in the original vert struct/program is required, so you can only add to these. If you need too, I would recommend creating your own cginc. files and creating a copy of the original vertex programs for you're custom shader. You could simply jam it into the pass but as you'll likely be using it in multiple passes/shaders, a separate include will allow you to reuse your vert program in many places. Otherwise, you can just use the included vert programs.

Fragment Program

The existing fragment programs can be found in DeferredPasses.cginc and ForwardPasses.cginc respectively. If you're using your own vert struct/program remember to pass it in here. You're only required to keep the Instance ID setup and the projection declaration; everything else is up to you. This projection struct gives you all the information you need to know about your projection (screen position, world position, UV coordinates and normals). If you're interested in digging through the methods the system uses in the included passes, they can be found within the projections.cginc file.

Creating your Projection

Now that we have our first shader, we can start creating our projection. The projection is the class the system uses to determine the required shader for your decal, set up a material for it, and feed it the shared details of your decal. Each renderer will then use that material with a material property block to feed in per-instance details. To begin, create a new class and inherit it from the Projection class. Be sure to include "using LlockhamIndustries.Decals;" at the top of your class. Now we need to tell the projection which shader to use, which rendering paths it supports, what its properties are and how to apply them, and which properties can be applied per instance.

Referencing your shader

The projection class contains 9 material properties (One for each shader method/rendering path combo), each expecting a material to be returned. By default these all return null. All we need to do here is override the property we are expecting to use, and have it return a material using our shader. The base class includes a method to do just this (MaterialFromShader(Shader)), all it requires is a shader type. So if you're planning on using forward rendering, with the mobile shader method, you would override the MobileForward property to return MaterialFromShader("your shader"). If you're using deferred rendering with the standard shader method (and don't plan to force the decal to run in forward rendering), you would override the StandardDeferredOpaque and StandardDeferredTransparent properties instead, each returning an appropriate shader. Opaque and Transparent variants can use the same shader if you only wish to write the one. Having variants allows us to optimize the opaque shader and use it when applicable.

Supported rendering paths

The projection class contains an abstract SupportedRendering property that uses an enum to determine which rendering paths the projection will support. All you need to do is override this property returning either "Forward", "Deferred" or "Both".

Setting up your properties

Here you will need to create serialized variables for your shaders custom properties. For example, if you have an albedo texture and color, these will need to be stored and serialized. The default shader values (_Cutoff, _NormalCutoff, _TilingOffset, _MaskBase, _MaskLayers) already have variables built into the projection class, so you don't need to create variables for these.

You will also need a custom inspector to modify these variables. This inspector should be inherited from the ProjectionEditor class and call the protected Mark() method whenever one of these variables are changed. There are also a bunch of methods within the ProjectionEditor class that should be called within your custom editors OnInspectorGUI method to allow users to modify the default variables. This is a pain to setup, but a necessary optimization to minimize overhead. This custom inspector will automatically be used by the ProjectionRenderer's inspector.

Applying your properties

Now that we have our properties stored, we need to tell the projection how to apply them to our material. To do this we override the Apply() method. This is the method called to update our projections material. The overridden method should always call base.Apply() to update the default values before anything else. Then we need to tell the material being passed in how to apply our properties. This should look something like this:

```
Material.SetTexture("_MainTex", texture);
Material.SetColor("_Color", color);
```

Setting up your per instance properties

If you have any properties that you would like to be changed on a per-instance basis (For example your albedo color) and they are sampled per instance within your shader, they need to be declared so the system knows to include them in each instances material property block (when applicable), as well as set up a UI for them. To do this we override the UpdateProperties() method to fill our projections properties array with our per-instance properties. The system has its own "ProjectionProperty" struct, which it uses to describe each property. Filling the array should look something like this:

```
public override void UpdateProperties()
{
    //Initialize property array
    if (properties == null || properties.Length != 2) properties = new ProjectionProperty[2];
    //Albedo Color
    properties[0] = new ProjectionProperty("Albedo", Shader.PropertyToID(_Color), albedo.Color);
    //Emission Color
    properties[1] = new ProjectionProperty("Emission", Shader.PropertyToID(_EmissionColor), emissive.Color, emissive.Intensity);
}
```

The first property (index 0) in the array should always be the property responsible for the transparency of your decal. This is the property being adjusted by the in-built Fade component to fade out your decals.

The Finish Line

That's it. The system will automatically detect any classes inherited from the projection class and list them to be available when creating a new projection within the ProjectionRenderer Inspector. You can now create an instance of your brand new projection and start experimenting at your leisure.