Better Shaders
Documentation

Introduction

Better Shaders is a system which targets the following goals:

- Write Lit Shaders in a simple format, akin to Unity's Surface Shader system
- Shaders automatically compile to Standard, URP, and HDRP pipelines
- Shaders can be distributed and packaged for the asset store
- Shader creation is more modular for developers and users of those shaders

If you are unfamiliar with writing text based shaders, this document will not teach you how to write shaders from scratch. However, most of what you learn from Unity's Surface Shader documentation or the many tutorials across the web will apply here, with minor adaptations.

Surface Shaders - History

In Unity 3, Unity introduced a system for writing shaders called Surface Shaders. Most of a shader's code is about lighting - and the surface shader system abstracted you from having to deal with lighting in your shaders. Instead, you provide data about how you want the pixel to be lit in the form of an albedo, normal, and other attributes, and the surface shader system would generate the full shader with all the multiple passes and lighting code included with it. This was a fantastic abstraction, which had the extra benefit of allowing shaders to automatically upgrade through many changes in Unity- The addition of VR, multiple different lighting systems, forward and deferred rendering, Physically Based Rendering, different lightmapping and global illumination engines, etc. A shader written years ago in Unity will load in a modern version of Unity, and compile with all of the new features working, without changes from the author.

However, when Unity introduced the Scriptable Render Pipelines, they decided to abandon the Surface Shader system, and let each team go in completely different directions. Writing a simple shader went from a few dozen lines of code, to many thousands. Further, every time Unity changes something, your shaders will break, requiring you to rewrite large chunks of them. Additionally, shaders written for the High Definition Render Pipeline (HDRP) are not compatible with ones written in the Universal Render Pipeline (URP), or vice versa - and all shaders written for the old pipelines are not compatible with either of them. What was once a wonderful place to write shaders quickly became the worst place to write shaders, and despite the community's insistence for years, Unity has refused to do anything to make the situation better. Further, for asset store authors, it's now impossible to have users install your assets and have them just work, and have to resort to shipping multiple copies of all their assets in zip files that user must extract for their correct Unity version and SRP. As the unity versions increase, so does the number of copies that must be maintained, since shaders written in HDRP for Unity 2019 are not compatible with shaders written for HDRP in 2020.

Enter Better Shaders

So I created Better Shaders to help rectify this issue. With it, you can write a shader in a simple surface shader style, and it will compile into the currently installed SRP automatically. These shaders can be distributed to other users of Better shaders as is, or exported to work without Better Shaders, and packaged into a single file which will extract the correct shader for the correct version of SRP/Unity for your users.

Finally, Better Shaders adds features to make writing shaders easier and more modular, allowing you to effectively subclass other shaders, or have users combine them from stackable shader fragments, adding snow to an existing shader. You no longer have to struggle with building a giant mega-shader, you can now build things as independent modules which can be easily combined via GUI or code.

Getting Started

Better Shaders comes with a nice set of examples to get you started, from Lit and Tessellated shaders, to stackable effects. Under BetterShaders/Samples/Basic you will find

various shaders that show off the basics. Under the BetterShaders/Samples/Stackable folder, you will find a collection of stackable shaders which can be used to apply effects to existing shaders.

Better Shaders has a series of templates that you can use as starting points when creating a new shader. You can access them from the right click create menu in the project, under Create/Shader/BetterShaders. The **Documented shader** will create a template with all the major blocks and code stubbed out, with explanations about what each thing does. The major blocks of a shader are all defined by START_ and END_ block notation, with all blocks being optional. In fact, a blank file will compile just fine.

The main **block** types in Better Shaders are:

Options

The option block lets you set compile time features of the Better Shaders system- for instance, you can turn on tessellation, alpha, etc. These are described in the Options Block Options section below, or by creating a documented shader from the menu.

SubShaders

The Subshaders block allows you to inline blocks of other shaders into this one. Unlike a traditional .cginc include, this will pull all the block data into the shader, and chain any chainable functions together. With this, it's possible to inherit other shaders, adding functionality to them. For instance, the LitTessellation example SubShaders the Lit shader, and adds tessellation.

Properties

The properties block is for specifying material properties. Consult the Unity docs for how to specify material properties.

CBuffer

In HDRP and URP, the batching systems only work if all of your uniforms are put into a special CBuffer. So while you can declare uniforms directly in your code block, placing

them here will keep batching working in SRPs. Note that you do not need to put texture declarations in these blocks, only uniforms (float, float2, etc).

Defines

The defines block is inserted at the top of the shader. You can put various defines in here to turn features on or off in your code, add #pragma directives, or insert code that has to be declared before anything else in the shader is run.

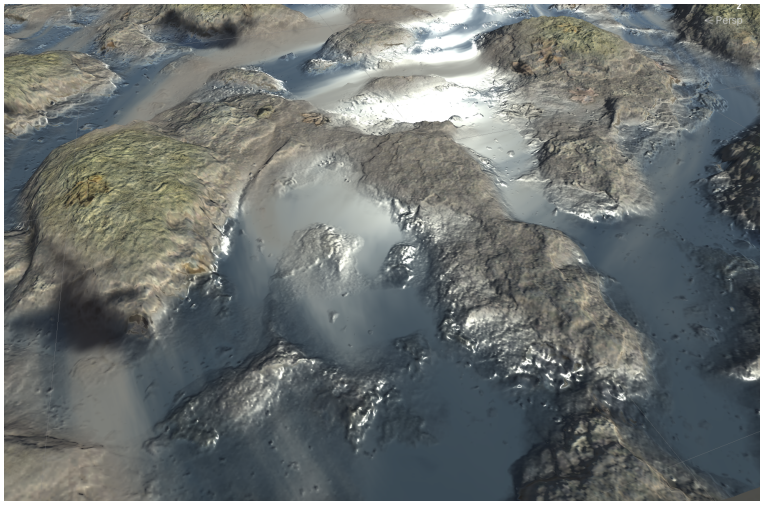
Code

This is the block where you write your code and functions. See the documented shader for all functions, but the main entry functions are:

```
BEGIN_CODE

// modify the surface color
void SurfaceFunction(inout Surface o, ShaderData d)
{
   o.Albedo = d.worldSpaceNormal;
}

// modify a vertex
void ModifyVertex(inout VertexData v, inout ExtraV2F d)
{
    v.vertex = v.vertex + float3(0,1,0);
}

// modify a tessellated vertex
void ModifyTessellatedVertex(inout VertexData v, inout ExtraData d)
{
   v.vertex = v.vertex + v.normal;
}

void FinalColorForward(Surface o, ShaderData d, inout half4 color)
{
   color *= half4(1,0,0,1);
}
```

These functions are ‘chainable’. That means if you inherit the lit shader via the SubShader block, you can write your own SurfaceFunction which will be called after the Lit shader is run. So you might white a shader which adds moss to objects, inherit the lit shader, then blend the moss onto the resulting shading in your function. This makes it very easy to quickly combine existing shader fragments- Inherit from Lit, add moss, add paintable wetness, add snow.

Stacked Shaders

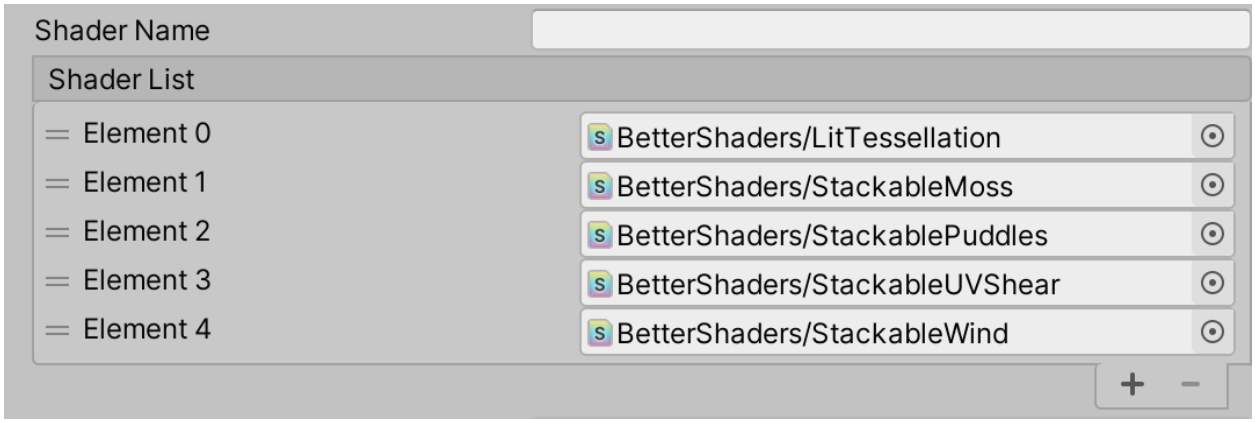


An example of a stacked shader:

- LitTessellation
- Moss
- Puddles
- UVSheer
- Wind

Written as 5 separate shaders, and combined by the user.

Shaders can also be stacked. There’s nothing special about stackable shaders, they just use the subshader system to combine existing shaders together. You can create one from the right click menu in the project view, under Create/Shader/Better Shader/Stacked Shader



Once created, simply add any shaders you want to stack in the order you want them stacked. Note that only shaders created with Better Shaders can stack. When you hit apply, it will import them into a single shader. You can even use shaders multiple times in the stack, applying the same effect. Note there are some caveats when designing shaders for multi stacking, discussed below.

In the example above, the LitTessellation shader forms the base of the stack. Moss is added based on slope and height filtering, puddles are added based on a puddle height, and UVSheer effect modifies the UV coordinates, and then the wind is applied with the modified UVs. Each effect was independently created as its own shader, modifying input data for the next.

Textures and Samplers

The old, standard pipeline had an incomplete API for declaring textures, samplers, and sampling them, and defaulted to the DX9 method of declaring them together. ie:

```
sampler2D foo;

half4 albedo = tex2D(foo, uv);
```

This will still work in all 3 pipelines, but for more advanced shaders you will want to share samplers and use more advanced ways to sample textures. This is where things get a bit tricky. In URP/HDRP there is a clean and complete API for such operations, but in

Standard there is a rather spotty API which is missing many functions and uses very long syntax for everything, like:

```
UNITY_SAMPLE_TEX2D_SAMPLER(tex, sampler, uv);
```

So, for the most part you can use either API- if you use the above line to sample a texture in Better Shaders, it will compile that to the appropriate function in the SRPs. However, the new API for these functions has been ported from HDRP/URP into the standard library as well, allowing you to use a much better written and consistent API than the standard library provided. An example of some of these functions is listed below. For a complete listing, consult the SRP documentation. No, just kidding, they don't document the shaders, you'd have to crawl through the include files in core and just find it. Or, if you want to do it the easy way, you can look at the BetterShaders_Template_Standard_CommonHLSL.txt file which is used in Better Shaders. Anyway, a few of the most common things:

Use this to initialize a structure:

```
ZERO_INITIALIZE(type, name);
```

To declare a texture, use one of these:

```
TEXTURE2D(textureName) ;
TEXTURE2D_ARRAY(textureName) ;
```

To declare a sampler:

```
SAMPLER(samplerName);
```

To sample a texture:

```
SAMPLE_TEXTURE2D(texture, sampler, coord2);
SAMPLE_TEXTURE2D_LOD(texture, sampler, coord2, lod);
SAMPLE_TEXTURE2D_BIAS(texture, sampler, coord2, bias);
SAMPLE_TEXTURE2D_GRAD(texture, sampler, coord2, dx, dy);
SAMPLE_TEXTURE2D_ARRAY(texture, sampler, coord2, index);
SAMPLE_TEXTURE2D_ARRAY_LOD(texture, sampler, coord2, index, lod);
SAMPLE_TEXTURE2D_ARRAY_BIAS(texture, sampler, coord2, index, bias);
SAMPLE_TEXTURE2D_ARRAY_GRAD(texture, sampler, coord2, index, dx, dy);
```

Overall, this is a much better written set of wrappers for texture sampling, so all examples in Better Shaders use this form instead of the older style. Also note the way unity associates the samplers with the data on a texture. If you create a texture and sampler named:

```
TEXTURE2D(_AlbedoMap);
SAMPLER(sampler_AlbedoMap);
```

Then Unity will read the wrap mode, filter mode, and other sampling settings from the _AlbedoMap texture.

Getting data from the vertex to pixel shader

Better Shaders hide's all the structures from you, unlike surface shaders. This greatly simplifies the parser, and makes it easy for Better Shaders to manage tessellation better than surface shaders did. It also keeps code consistent and well named, so you don't have to guess what "pos" or "normal" really means anymore. However, sometimes you need to generate data in the vertex shader and pass it to the pixel shader. To manage this, you are passed an ExtraV2F structure with the vertex functions - this contains 8 float4 values that you can use however you want. They will get sent through the interpolator stages and be available on the SurfaceData structure in the pixel shader.

```
void ModifyVertex(inout VertexData v, inout ExtraV2F d)
{
   d.extraV2F0 = ComputeData(v);
}
void SurfaceFunction(inout Surface o, ShaderData d)
{
   o.Albedo = d.extraV2F0.xyz;
}
```

Some notes:

- You should only use the extraV2F data for data meant to be taken across the shader stages. Using it as a scratch pad in a single stage will cause the data to be copied to

the pixel shader even if it's unused there. The stripper and shader compilers can not detect this.

- There are 8 extraV2F float4's available, however, how many interpolators you have available depends on the shader model you are targeting, which you can override in the Options block.
- You can learn more about interpolator counts on different hardware and APIs from Unity's docs: https://docs.unity3d.com/Manual/SL-ShaderSemantics.html

A note on motion vectors in HDRP. If you are moving vertices with the _Time global, you must use the version of time on the extraV2F structure. This is because motion vector generation requires transforming the vertices based on the last frames matrix's and time. As such, _Time would be for this frame, so use extraV2F's .time value instead, which will be correct for the evaluation frame.

Understanding Stripping

When a shader is compiled, any unused parameters of the various structures are stripped from the code. Don't use the vertexColor parameter? It will be stripped from the shader completely, keeping things lean. However, if you use the vertexColor anywhere in your shader, it will be copied through all the stages; and sometimes this is not desirable. For instance, perhaps you encode some data into the vertex color, then decode it in the vertex shader? By default, this data would still be copied to the pixel shader where it isn't used. However, Better Shaders provides a simple way to optimize this. You can add any data you don't want copied to the pixel shader with the StripV2F option in the Options block.

Now in complex shaders, you might only require certain interpolator data when certain shader features are enabled. Perhaps you use vertexColor, but only in one specific mode. Having the vertex color data used in all modes is slightly wasteful, so it's possible to have the use of vertex color as well as any extraV2F variables wrapped in define checks by using an option.

See the Stripping example in the Samples/Basic folder for an example of both these techniques.

Common Function Library

You can find common functions to abstract across pipelines in BetterShaders_shared.txt. Here's some examples

```
float3 TransformWorldToObject(float3 p)
float3 TransformObjectToWorld(float3 p)
Float3 ObjectToWorldSpacePosition(float3)
float4 TransformWorldToObject(float4 p)
float4 TransformObjectToWorld(float4 p)
float4x4 GetWorldToObjectMatrix()
float4x4 GetObjectToWorldMatrix()
half3 GetSceneColor(float2 uv)
float GetSceneDepth(float2 uv)
float GetLinear01Depth(float2 uv)
float GetLinearEyeDepth(float2 uv)
float3 GetWorldPositionFromDepthBuffer(float2 uv, float3 worldSpaceViewDir)
float3 GetSceneNormal(float2 uv, float3 worldSpaceViewDir)
half3 UnpackScaleNormal(half4 packednormal, half scale)
void GetSun(out float3 lightDir, out float3 color)
```

Note that GetSun returns the main light in Standard and URP, and the first directional light in HDRP. Note that TransformWorldToObject and TransformObjectToWorld return the real world position, not the camera relative position in HDRP. If you are computing the world position in the vertex or tessellation stage, use ObjectToWorldSpacePosition instead for the camera relative world position.

Advanced Stackable Tricks

When working with stackables, you might want to know if another shader is in the stack, or you might want to have some data available to other shaders in the stack. This is all quite easy to do, and if you look at the Lit Tessellation and Stackable Puddles shader you can see how this is done.

First, in the LitTessellation shader we add a define block:

```
BEGIN_DEFINES
#define _HAS_LIT_TESSELLATION 1
END_DEFINES
```

Now, other shaders can know if that shader is in the stack with them by checking this define. Note that you will only know if the other shader is in the stack, not if it's before or after us. For that, you'd need to do a little trick like this in the shader that wishes to know if the above shader comes before us in the stack:

```
BEGIN_DEFINES
   #if _HAS_LIT_TESSELLATION
        #define _TESSELLATION_IS_BEFORE
   #endif
END_DEFINES
```

Next, we setup some data we'd like to be available to other shaders in the stack, and set it in our modify function. We do this using the Blackboard Block.

```
BEGIN_BLACKBOARD
   float3 originalVertexPosition;
   float vertexHeightOffset;
END_BLACKBOARD

   void ModifyTessellatedVertex(inout VertexData v, inout ExtraV2F d)
   {
      d.blackboard.originalVertexPosition = v.vertex;

      float2 uv = v.texcoord0.xy * _AlbedoMap_ST.xy + _AlbedoMap_ST.zw;
      float height = SAMPLE_TEXTURE2D_LOD(_AlbedoMap, sampler_AlbedoMap,
uv, _DisplacementMipLod).a * _DisplacementAmount;
      v.vertex.xyz = v.vertex.xyz + v.normal * height;

      d.blackboard.vertexHeightOffset = height;

   }
```

In this case, we'd like to know the original vertex position, and the final height we used to perturb the vertex along the normal axis. Uniforms declared inside the blackboard block are added to the ExtraV2F and ShaderData structures, as a struct named 'blackboard'.

Note that this data does not get sent to different stages, if you set it inside the ModifyVertex function, then that data is only valid in other vertex functions.

Also note that the structure is declared inout - without this, any data written would only be in the scope of this function.

Next, in the puddle shader, we add the ModifyTessellatedVertex function, calculate our own offset, and if ours is above the current height, we use it instead- this clamps the vertex to the water height.

```
void ModifyTessellatedVertex(inout VertexData v, inout ExtraV2F d)
   {
      #if _HAS_LIT_TESSELLATION
         half mask = 1;
         half pudHeight = 1;
         GetPuddleParams(mask, pudHeight,
GetPuddleMaskLOD(v.texcoord0.xy));
         float minOffset = pudHeight * _DisplacementAmount;
         if (d.blackboard.vertexHeightOffset < minOffset)
         {
            v.vertex.xyz = d.blackboard.originalVertexPosition + v.normal *
minOffset;
            litTessDataHeightOffset = minOffset;
        }
      #endif
   }
```

Using this method, you can expand the possibility of what can be stacked together quite a bit. If the user puts this shader on a non-tessellated shader, or one which doesn't provide this data, then this effect simply won't happen.

To review: The blackboard is a struct inside the ExtraV2F (vertex and tessellation stages) and ShaderData (fragment/pixel shader) structures created with whatever you declare inside the Blackboard block. You must mark the function signature with an inout to modify the data, and the data is only available in the stage it was written in. Finally, for safely, use defines and conditional compilation to handle the case that your stackable is used outside of the system you are currently building.

Include stack only on one pipeline

If you wish to have one of your stackables only appear in, say, HDRP, you can mark it with the Pipeline “HDRP” in the options block, and it will only be included on the HDRP version of the shader.

Designing for Multiple Stacks

Multiple copies of the same shader can be layered together via includes or the stackable interface. For instance, you could use the moss stackable to stack another texture, with its own settings, on top of the first. When building shaders designed for multi-stacking, some care is needed.

First, any code other than the official functions (SurfaceFunction, ModifyVertex, etc) needs to be wrapped in include guards.

```
#ifndef __STACKABLE_MYSTACKABLE_INCLUDES__
#define __STACKABLE_MYSTACKABLE_INCLUDES__

void AddF2F(float f) { f += f; }

#endif
```

To allow for multiple copies of the same effect to run, the shader generation system will find all of the variables declared in the property and cbuffer blocks, and rename them throughout the code by appending a postfix onto the end of them. This may cause issues if you do any of the following if you use the _Ext_ string in your properties or _DEF_ string in your keywords. Note that only shader_feature_local keywords are handled in the mutation system- programmatically adding keywords to the global namespace would be evil.

Finally, if for some reason you want to mutate something which is not covered here, the string %STACKIDX% will get replaced with the current stack index number if found in your properties, cbuffer, code, or defines block.

Preventing Shaders from Stacking

Sometimes you don’t want a shader to stack. An example might be where you have a shader like the Lit shader in the Samples directory- it’s not designed to stack, and many

other shaders may want to include it. By specifying the stackable option in the options block, you can prevent a shader from being stacked, so that no matter how many times it's subshader'd in, only one copy will be included.

```
BEGIN_OPTIONS
   Stackable "False"
END_OPTIONS
```

Shared API for Grab Pass/Scene Color, Depth and normal buffers, etc

Grab pass style effects are fully supported on all three platforms. Though note that each render pipeline treats these differently. In Standard, it will do a full pass of the scene buffer at the moment it gets to the grab pass shader. In URP/HDRP, it uses the opaque camera buffer (which you must enable in URP cameras). That said, the syntax is trivial:

```
BEGIN_OPTIONS
   GrabPass { "_Grab" }
   Alpha "Blend"
   Workflow "Unlit"
END_OPTIONS

BEGIN_CODE
    void SurfaceFunction(inout Surface o, ShaderData d)
    {
        o.Albedo = 1 - GetSceneColor(d.screenUV);
        o.Alpha = 1;
    }
END_CODE
```

The depth texture API is:

```
float GetSceneDepth(float2 screenUV);
float GetLinear01Depth(float2 screenUV);
float GetLinearEyeDepth(float2 screenUV);
```

Note that when rendering in URP or Standard with the Forward renderer, you have to explicitly set a flag on the camera or render pipeline to generate the depth texture.

The API to sample the Normal Buffer is:

```
GetSceneNormal(float2 screenUV, float3 worldSpaceViewDir);
```

Like the depth texture, the availability of the normal buffer is dependent on settings in URP and Standard pipelines. Also note that the data is very different between pipelines, because of the way it is generated.

In Standard/Forward rendering, unity renders the scene with a replacement shader. This means that vertex displacement, tessellation, and normal mapping do not appear in the buffer. However, in deferred rendering they will.

In URP 10 and below, a normal buffer is not available. However, Better Shaders will recreate a normal from the depth buffer. Unlike in the standard pipeline, this will properly represent tessellation and vertex modification, but texture based normals will not be present. In URP10, a proper normal pass is available making textured normals present.

If you need the world position from the depth buffer, you can use this API to get it:

```
float3 GetWorldPositionFromDepthBuffer(float2 uv, float3 worldSpaceViewDir)
```

Custom CBuffers and Instanced Properties

While you can declare Custom CBuffers and Instanced properties in your code as normal, however, if you want to support stacking the same shader multiple times, or have multiple shaders in the stack use the same buffers, you can declare them like so:

```
BEGIN_CBUFFER(MyBufferName)
   half4 _Color;
END_CBUFFER
```

```
BEGIN_INSTANCING_BUFFER(_MyProps)
   UNITY_DEFINE_INSTANCED_PROP(half4, _Color)
END_INSTANCING_BUFFER
```

Everything defined in these blocks will be merged together into a single CBuffer/InstanceBuffer in the final shader, and if the same shader is stacked multiple times, it's names will be mutated with the _Ext_<stack index> extension.

Overriding Pass Data

Sometimes you want to override the way that specific passes work. For instance, changing the culling mode, applying an offset, or modifying the stencil data. To do this, you can include pass blocks, which target individual passes, or all of the passes.

```
BEGIN_PASS("All")
    Cull Front
END_PASS
```

This will change the culling mode to Front in all passes. Note that you can have multiple blocks targeting different passes, however, if the all pass is used it will override all named passes. The pass targets are All, Forward, ForwardAdd, GBuffer, Depth, Shadow, Meta, and Select. Also note that in HDRP, stencil is used heavily for the default drawing, so if you override these settings you will need to understanding how it's used in HDRP if you plan to use your shaders in HDRP.

Additional Passes

You can add additional passes for your shader with the BEGIN_CUSTOM_PASS block. Note that there is no abstraction layer available in custom passes, so any utility functions or SRP specific libraries will need to be manually included. For an example, see the outline shader in the demo scene.

Pass Specific code

Better shaders has a definition in each pass, so that if you need to write pass specific code you can. Unlike unity's definitions, these are consistent across all 3 pipelines:

_PASSFORWARD
_PASSGBUFFER
_PASSFORWARDADD (both forward and forward add are defined in the forward add pass)
_PASSSHADOW
_PASSSCENESELECT
_PASSDEPTH

URP Specific Passes:

_PASSDEPTHNORMALS (_PASSDEPTH is also defined)

HDRP Specific passes:
_PASSTRANSPARENTDEPTHPREPASS
_PASSMOTIONVECTOR
_PASSMETA
_PASSFULLSCREENDEBUG
_PASSUNLIT

World Space Normals

If you want to provide normals in World Space instead of Tangent Space, you can add

```
#define _WORLDSPACENORMAL 1
```

To the defines block. Note that if this is defined anywhere in your stack, it will be true for everything in the stack.

Custom Templates

By default, Better Shaders will use it's included templates for the supported pipeline. The templates contain all the pipeline specific code, like the lighting model. It is possible to have your own optional, custom templates, and to choose these via the options block or with a stackable.

For instance, a common use case for this is making significant changes to the lighting model that could not easily be made from within the regular framework. To do this, duplicate the template files for the given pipeline you want to modify (and rename them), and create a copy of the Pipeline*.cs file of the pipeline you want to modify and modify it to load your new template files and process them accordingly.

To specify the use of your templates, you can override the template in the options block. For instance:

```
BEGIN_OPTIONS
   AdapterStandard MyCustomAdapterStandard
   AdapterURP2019  MyCustomAdapterURP2019
END_OPTIONS
```

This would cause this shader to use the MyCustomAdapterStandard code when compiling to the standard pipeline, and the MyCustomAdapterURP2019 code when compiling URP for 2019. When compiling under URP2020, or HDRP, the built in adapters would be used.

Note that you can stack or subshader this option into an existing shader. So for instance, if you wrote some custom tool shading lighting code and created templates for them for each render pipeline, any shader written with Better Shaders could have this added to its stack to change the lighting model used. This is incredibly powerful, allowing you to ship new lighting models that work with existing shaders.

The down side to this, of course, is that you are modifying SRP specific code, and will have to understand both how that code works, as well as updating it to support new versions of a given SRP. And if you want that code to be truly cross SRP, you'd have to implement the same changes in all the templates. However, this is a powerful feature because it lets you write new lighting models for Better Shaders independently of the other shader code.

Centroid Sampling and interpolator modifiers

Centroid sampling and other interpolator modifiers (_pp and _sat) are available for the vertex color, texcoord, and extra v2f interpolators. You can set any modifier via the options block:

```
BEGIN_OPTIONS
      VertexColorModifier "_centroid"
      TexCoord2Modifier  "_centroid"
      ExtraV2F3Modifier "_centroid"
```

```
END_OPTIONS
```

Tricks and Tips

- If you declare the ShaderData structure as inout in your SurfaceFunction, you can modify the TBNMatrix, worldSpaceNormal, and worldSpaceTangent members, and they will be used in the lighting functions on all 3 pipelines. Check out the Flat shader as an example.
- You can think of the define block as a place to put anything that needs to be included before any code is included. Pragma's, traditional cginc includes, etc.
- You can bring in includes via the SubShader block instead of using a traditional include, and they will get embedded into the output shader file instead of being included via path at compile time. This can be useful if you want to ship your shaders and don't want to require them to be installed relative to some path on the end users machine.
- If you write %STACKIDX% in your shader, it will be replaced by the number of the shader code in the stack. You could use this, for instance, to label your headers when stacking many shaders. Note that the first layer will not define %STACKIDX%, so if you want to use it to index into things, consider something like #define MYINDEX 0%STACKIDX%, as this will be 0 on the first layer an 01 on the second, etc..
- The Surface struct contains an outputDepth, if you write this, the shader will render with the SV_DEPTH semantic and allow you to modify the depth buffer value. Note that it's cheaper to render without SV_Depth, and that the ShaderData.clipPos.z value is equivalent to what the hardware would provide.
- There is a bool isFrontFacing declared in the ShaderData structure. If you use it, the shader will be compiled with the SV_IsFrontFace semantic.
- If you need to tell which render pipeline you are in, you can #if with _HDRP, _URP, and _STANDARD

Alternate Lighting Models

In Built in and URP, some additional lighting models are available. The following defines can be added to the defines block to change the lighting model:

```
#define _SIMPLELIT 1
```

The Simple Lit mode is available in URP and Built in, which uses a simpler calculation for specular response. In Built In, this maps to the lambert shading model.

```
#define _BAKEDLIT 1
```

The Baked Lit model is only available in URP, and removes all runtime lighting code in favor of only supporting baked lighting. As such, only the albedo and normal are used from the surface data structure.

Exporting Shaders

On both the regular and stackable shaders you will find a small export tool. You can select the target render pipelines, and press export, and it will export a shader with the name : originalname_renderpipeline. This is the raw shader code generated by the system.

Packaging Shaders for the Asset Store

Ideally, you can ship your source files and if users have Better Shaders installed, they can stack them or modify them as they wish. But it's also likely you just want to ship working shaders to users who don't have Better Shaders installed. For that, I have an MIT licensed Shader Packager system, which you can download here:

https://github.com/slipster216/ShaderPackager

Using this system, you can export all the shaders you have written to standard vertex/fragment shaders for each SRP version you want to support. Shader Packager will combine those into one file, which appears as a shader to the user, but internally contains all of your shader versions inside of it, and selects the right one to use at Import time. This lets you ship assets that will work regardless of which render pipeline the user has installed.

Custom Editors

You can create Custom Editors for shaders in the system by giving them a CustomEditor “MyEditor” in the options block. However, you can also provide a custom editor for each shader in a stack of shaders, and it will draw each shader in the stack with it’s own editor! This can be done by declaring a SubEditor in the options block:

```
BEGIN_OPTIONS
    CustomEditor "JBooth.BetterShaders.BetterShaderMaterialGUI"
    SubEditor "TextureLayerMaterialEditor"
END_OPTIONS
```

Any shader including this shader via the subshader block or as part of a shader stack will have it’s custom editor set to the BetterShaderMaterialGUI, and this part of the shader will be handled by the TextureLayerMaterialEditor we have written. Any part of the shader not handled by a custom subeditor will be drawn as normal.

You can write your own subeditors like so:

```
using JBooth.BetterShaders;
public class MySubMaterialEditor : SubShaderMaterialEditor
{
      public override void OnGUI(MaterialEditor materialEditor,
         ShaderGUI shaderGUI,
         MaterialProperty[] props,
         Material mat)
      {
           // custom editor code here
      }
}
```

When run, the list of MaterialProperty’s passed will only contain Material Properties for your properties.

To correctly handle multiple uses of the same shader, some care will need to be taken around property and define names. However, this is trivially handled by using the following functions:

```
MaterialProperty myProp = FindProp(“_MyProp”, props);
// this will return the correct property - which may have been mutated if
you are using the same shader in the stack more than once. (ie: it might
actually be named _MyProp_Ext_1)
```

```
string keyword = GetKeywordName(“_MyKeyword”);
// as above, if you are using multiple copies of the same shader, the
shader keyword may have been mutated.

int index = stackIndex < 0 ? 0 : stackIndex;
DrawHeader(“My Shader (“ + (index+1) + ”)”;
// note that the stack index is available as well. This is -1 if you are
the first copy of yourself in the stack. Then 1, 2, 3 for each copy.
```

Using these, it’s quite easy to write editors that work well with multiple copies of the same shader in the stack.

The SubShaderMaterialEditor class also contains a number of helper functions to write better editors, such as the DrawHeader function above.

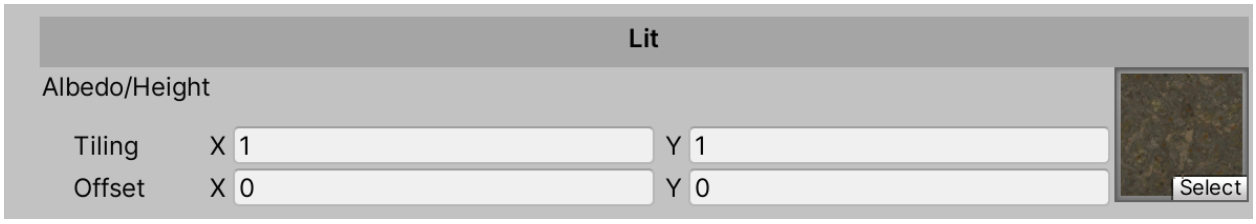
Custom Material Property Drawers

Alternatively, a collection of Material property drawers are included to make your shader GUI’s look better and be more reactive without having to write a custom editor. Unity includes a few of these, but Better Shader greatly increases these capabilities.

Better Header

Unity’s header attribute looks terrible, so a much nicer header is included. You can use the syntax:

```
[(BetterHeader(Lit)]
```



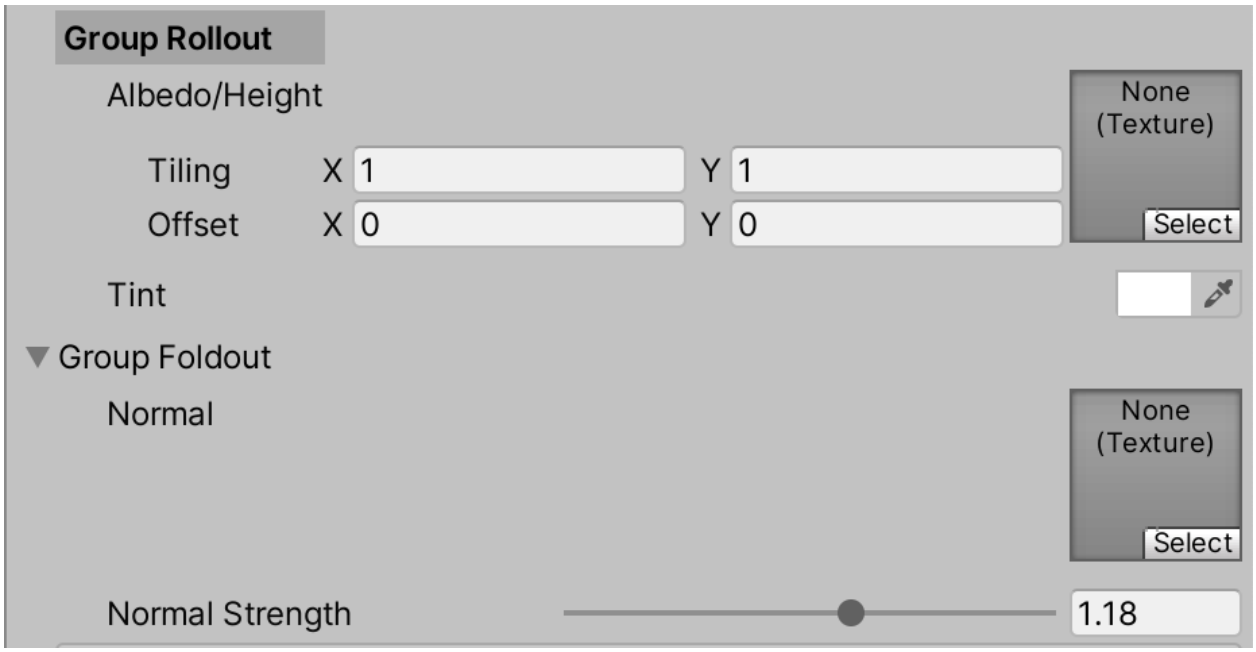
Note that you do not put quotes around your text, and the MaterialPropertyDrawer system doesn’t seem to like some symbols.

Group Rollout

Group Rollout, and Group Foldout allow you to create rollup bars and foldout controls. Any property below it which is supposed to hide when closed needs to be tagged with the Group name.

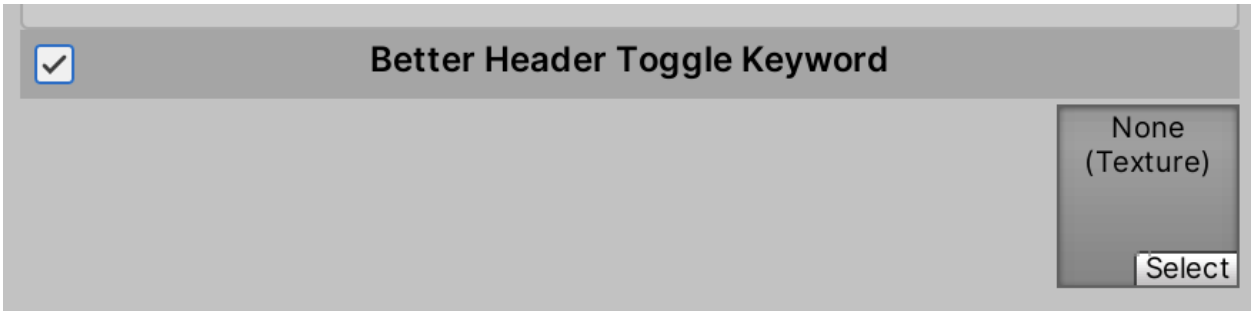
```
[GroupRollout(Group Rollout)]
[Group(Group Rollout)] _AlbedoMap("Albedo/Height", 2D) = "white" {}
[Group(Group Rollout)]  _Tint ("Tint", Color) = (1, 1, 1, 1)
```

```
[GroupFoldout(Group Foldout)]
[Group(Group Foldout)][NoScaleOffset]_NormalMap("Normal", 2D) = "bump" {}
[Group(Group Foldout)] _NormalStrength("Normal Strength", Range(0,2)) = 1
```



Keyword and property Toggle Headers

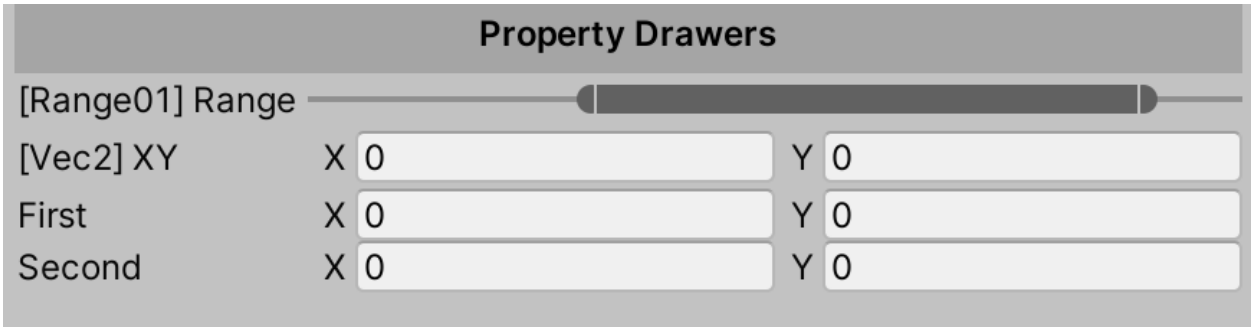
Headers can also have inlined toggle controls, which can toggle either a shader keyword or a float property. You can use the ShowIf attribute to only show properties when these toggles are true.




```
[BetterHeaderToggleKeyword(SomeKeyword)] _SomeKeyword("Better Header Toggle
Keyword", Float) = 1
[ShowIf(_SomeKeyword)] [NoScaleOffset]_MaskMap("", 2D) = "black" {}
```

Property Drawers

There is also a collection of controls that let you work with Vectors better. For instance, you can draw a Vector2 with the [Vec2] attribute. Or if you want to pack 2 vector 2's into one Vector4 property, you can use the [Vec2Split] attribute.



An example of those is here:

```
[Range01]_VectorRange("[Range01] Range control", Vector) = (0.3, 0.9, 0, 0)
[Vec2]_Vector2Draw("[Vec2] XY", Vector) = (0,0,0,0)
[Vec2Split(First, Second)] _Vector2Split("", Vector) = (0,0,0,0)
```

Shader Markdown

Another way to solve the issue of custom GUI is to use Shader Markdown, which you can find here:

https://github.com/needle-tools/shadergraph-markdown

This system makes use of a custom editor to allow you to quickly notate your properties for a better shader UI. Because it uses a custom GUI, it can do a lot more than Unity's Material Property Drawer system.

Syntax Highlighting

For Sublime Text 3 and Rider:

https://github.com/slipster216/BetterShaders_SyntaxHighlightSublime

Option Block Options

Inside the option block you can use the following options

```
BEGIN_OPTIONS
   ShaderName "Path/ShaderName"     // Path to shader
   Tessellation "Distance"          // distance, edge
   Alpha "Blend"                    // Blend, PreMultiply, Add
   Fallback "Diffuse"               // fallback shader
   Dependency "OtherShader"         // dependency shader
   CustomEditor "MyCustomEditor"    // Custom Editor
   Tags { "RenderType" = "Opaque" "Queue" = "Geometry+100" }
   Workflow "Metallic"              // Metallic, Specular, or Unlit
   StripV2F { vertexColor texcoord2 } // don't interpolate to pixel
   Stackable "True"                 // when false, only include once.
   VertexColorModifier "_centroid"  // add modifier to v2f interpolator
   ExtraV2F3Require "_MYDEFINE"     // extrav2F3 only used when MYDEFINE
   Pipeline "HDRP"                  // only stack in Standard, URP, HDRP
   DisableGBufferPass "True"        // disable gbuffer pass
   DisableShadowPass "True"         // disable shadow pass
   ShaderTarget "3.5"               // Force shader target
   EnableTransparentDepthPass "True"// enable transparent depth pass HDRP
```

```
END_OPTIONS
```

Note that options some options come in multiple variants, like ExtraV2F3Require is available for all ExtraV2F# interpolators as well as VertexColor. These can also be combined from multiple stackables, where each defines which defines are required for the interpolator to be used, and it gets compiled into “#if _DEFINE1 || _DEFINE2” etc.