Using the Strumpy Shader Editor in Unity:

Section 1: Basics

Installation → With Unity, either drag the UnityPackage into Unity, or, alternatively, select "Import Package" from the Assets Menu inside of Unity. Place the UnityPackage into your Standard Packages Folder (Applications/Unity/Standard Packages on OSX) to have Unity automatically offer to include the package on forming a new project.

As of this writing, the package uses Pro-Only features for the drawing of the preview, and is strictly limited to Unity3, as it creates surface shaders.

What does it do, exactly? → The Strumpy Shader Editor (SSE) provides a visual interface for the creation of Shader Graphs (.sgraphs), aimed at simplifying the process of shader design and to make it more ‚"Artist accessible"- It is based on Unity3's new "Surface Shader" concept, which allows the creation of shaders that properly interact with Unity's lighting schemes (Forward, Deferred, and Vertex/Fixed)

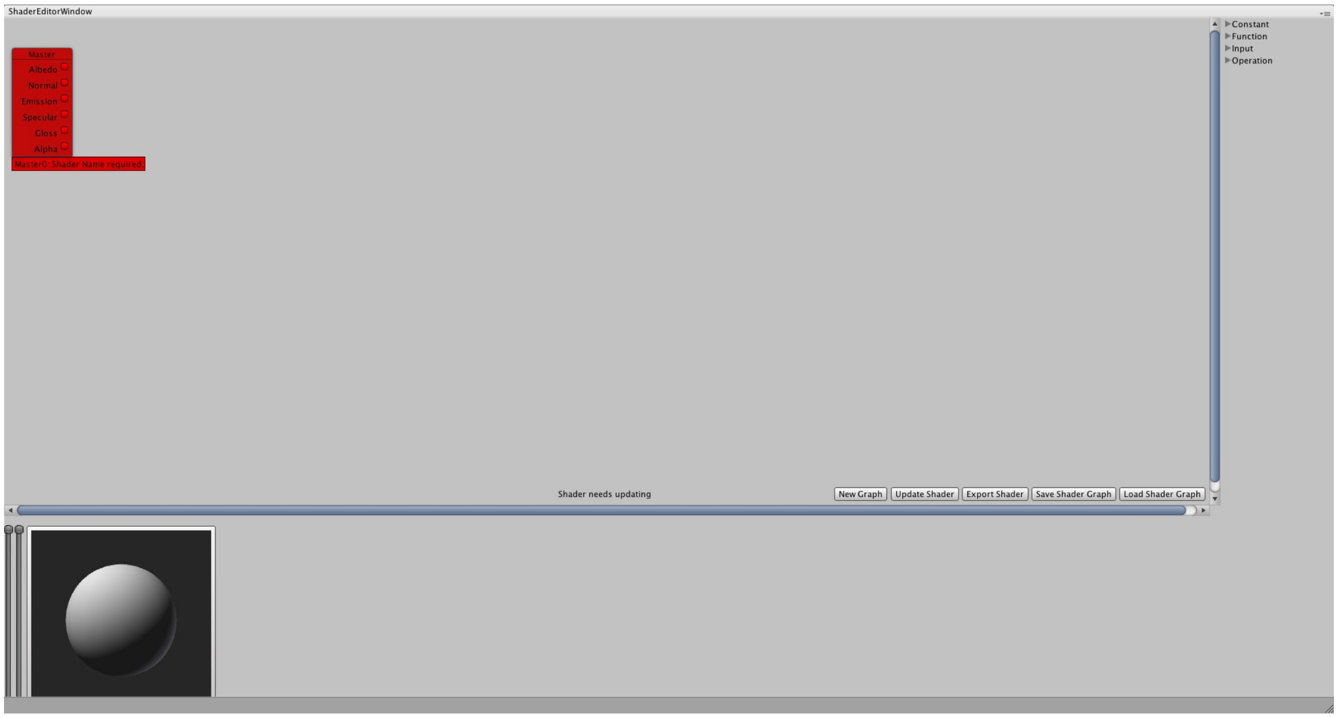
However, it should not be expected to create every conceivable effect, and is unsuited for tasks requiring advanced transformations or particularly advanced effects (Primarily those which require loop unfolding, such as relief mapping, ray-marching, crepuscular, etc)- It is more suited for defining how various texture maps should interact, a field in which it largely excels. It should also not be expected to produce elegant output code, nor should it's code output be considered proper- Rather, the code is stripped by the CG-Compiler producing largely optimal code.

Getting Started → After the package is imported into Unity, Open the Shader Editor Window using Window/Shader Editor. You can either doc the window into an existing set of tabs, or into the main Unity Window- Whichever you choose, you can hit the *Space* key to maximize it, which makes observing the graph layout much easier.

The default display (Fig 1-1) shows the basic layout of the editor. When launched, a new graph is created, and the output node is placed in the upper corner. This entire upper left area is the work area, where the graph is built. On the bottom of the work area you will find the buttons for file operations, such as new, export, save, and load, aswell as the "Update Shader" button. The Update Shader button will update the shader used in the preview to reflect changes in the graph- Please be patient, it can take several seconds for it to generate all the permutations for various lighting conditions. When a node is selected, there is an additional row of buttons along the top left, allowing you to delete the node, or break it's connections.

Located on the bottom of the window you will find the preview- The two vertical bars on the far left will adjust the rotation of the preview model, where clicking on the preview display will show material settings for the display, in the same format as when you would create a material normally.

Finally, on the right of the display, you will see the collapsable list of node blocks you can add- I heartily recommend keeping these elements collapsed, so you can quickly find the node your looking for, without having to thumb through a long list.



Master0: Shader Name required.

Shader needs updating

Figure 1-1: Overview

Section 2: Getting Started

Designing your first Shader → The function list can be mighty intimidating, so rather then dive into that, let's build up a simple shader; For that, first thing we need to do is assign a shader name- To do this, click on the Master node in the node view- Below, next to the preview you will receive a prompt for the variables of the node, in this case the name of the shader. This is what it will be listed under when you create a material in your scene, for example, if you filled in *Custom/MyShader*, you will be able to find it in the material drop-down under that name. As soon as you start to type a valid name, the Master Node will recolor itself grey, showing that it has no error.

Once this is done, expand the Input section in the nodes list, and add a new input of the *color* type. New nodes will appear in the upper left corner of your node window, so you may end up having to move the Master node (Click and drag it) to see your newly created node. Unwired nodes will appear blue, indicating they are not yet connected to the graph, and thusly will be excluded from compilation. To remedy this, let's wire the Color value to the material- Click the box on the left of the color node (it's output) to start drawing a connection, and place it on the Albedo input of the Master node. With this, you can now hit "Update Shader" and have the preview display reflect your new shader- If you did things right, you will end up with a solid black preview. This is because that color input you created defaults to a black tone, to adjust the value it's using for the display, click the preview and adjust the color value (Fig 2-1). Rather then actually requiring the user to define a color before being able to see the result, you can adjust the default color by clicking on the Color node you had created, and setting the value to, say, a bright white. You should also rename the color name to something that better expresses what your doing, such as "*Color*", which is what is used by the rest of Unity's built-in shaders to describe color (aswell as being what color-changing scripts look for!), By using names consistent with other shaders, the artist is free to change the shader on the material without having to reassign all of his or her references. If your going to be releasing the shaders you create with this tool, be sure to properly name your inputs!

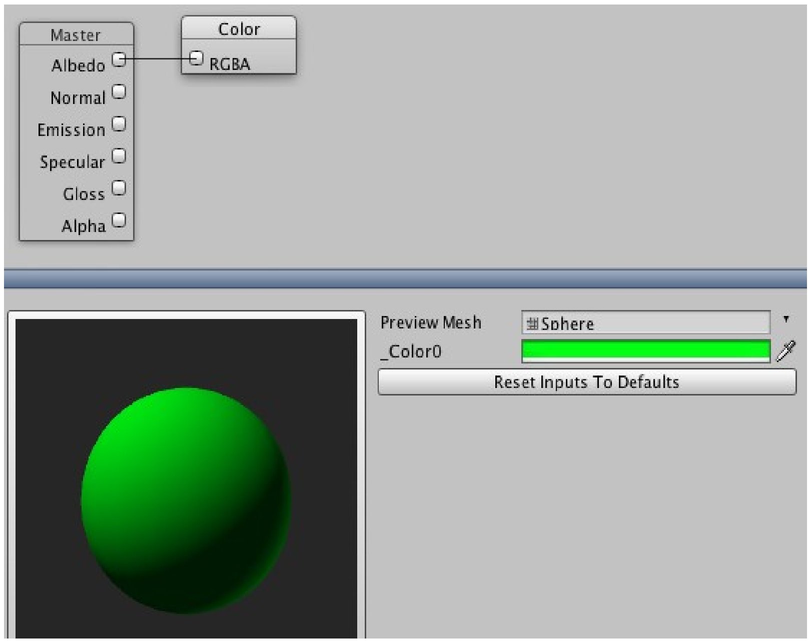


Figure 2-1: A adjustable color shader we just created

When naming your inputs, Unity has some specific naming conventions used in the builtin shaders, which are as follows:

Colors:

Color – The main color, accessable by script with Material.color
SpecColor – The specular color
Emission – The emissive color
Shininess – The glossiness amount
ReflectColor – Reflection color

Textures:

MainTex – The primary texture, accessible by script with Material.mainTexture
BumpMap – The normal map texture
Cube – The reflective cubemap, as used in cubemap shaders
Illum – Illumination Map, not to be confused with Emission
ParallaxMap – Parallax map, used by parallax shaders (Expect depth from Alpha)
DecalTex – Decal texture, or secondary diffuse map

By keeping names consistent with the builtins whenever possible, you can make it easy to quickly flip through shaders in the editor.

Texturing →

As fun as having a single colored object is, most people would agree that to liven up a game your going to need to use texture maps. The node for a plain 2D shader is Samper2D, where for a cube-map it is Sampler Cube, both to be found under Inputs in the node list. With SSE, most of the operators will pass a four component vector to each-other, however, samplers require a special function to read from them- Tex2D, or in the case of Cubemaps, TexCube.

Let's create a simple diffuse shader: Start by hitting the “New Graph” button, and again, give the master node a name, such as *MyShaders/Diffuse*. With that out of the way, add your Sampler2D node, and drag it a fair distance to the left of your master node, so we have room to build between them. In accordance with the naming conventions, give your sampler the proper name, in this case, *MainTex*. Notice that the Sampler2D has two outputs, the actual sampler, aswell as a UV parameter. The UV is the modified texture coordinates, after the user has defined the Scale X/Y aswell as the Offset X/Y, and as not necessarily the same for different samplers. I'll get back to this shortly, but for now, add a new function node, Tex2D. Tex2D samples a pixel from the provided sampler, filtered according to the texture settings from the assigned texture, as follows:

For filter mode Point, Tex2D will return the pixel it is over

For filter mode Bilinear, you only get the pixels value if you sample directly on it, otherwise the value is interpolated between adjacent pixels

For filter mode Trilinear (which only works with power of two textures), the result will factor adjacent pixels aswell as adjacent mipmap levels, removing hard transitions when dealing with high anisotropy levels (best used for floor textures, or particles with broken aspect ratios)

To use Tex2D, you connect the desired sampler and the position to sample- also be sure to wire in the UV-set you want to use for that sample. Once you have connected them, wire the Tex2D node to the Albedo input of the master (Fig 2-2). A very large proportion of the cost of a shader is determined by how many Tex2D nodes you have, since each one directly correlates to the graphics card reading a value off the texture. However, there is a major hit when you have *dependent-texture-reads*, that is, when one texture read is required to evaluate another texture. Graphics cards are optimized to perform all the texture reads together, which isn't as feasible when you layer your texture reads, such as in a distortion shader.

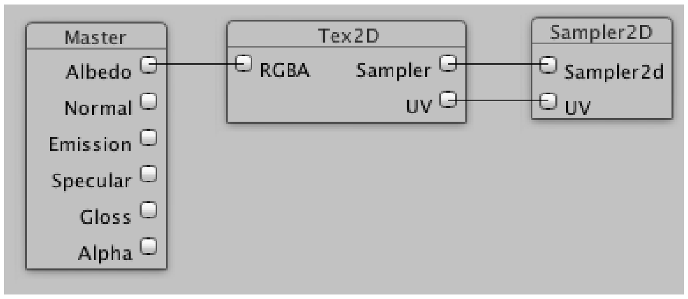


Figure 2-2, a simple diffuse shader

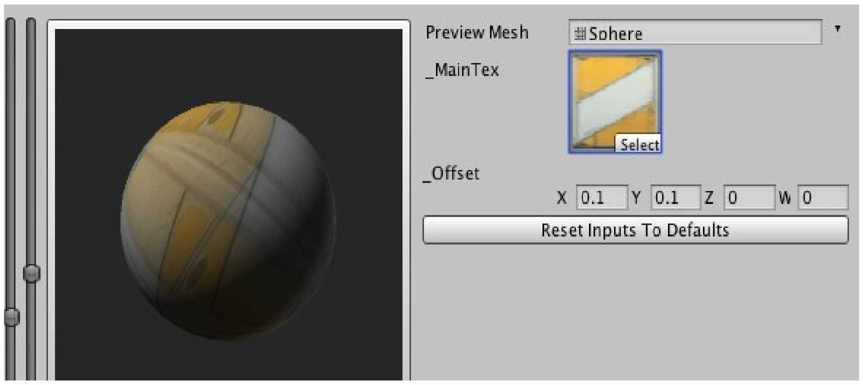


Figure 2-3 The double vision shader

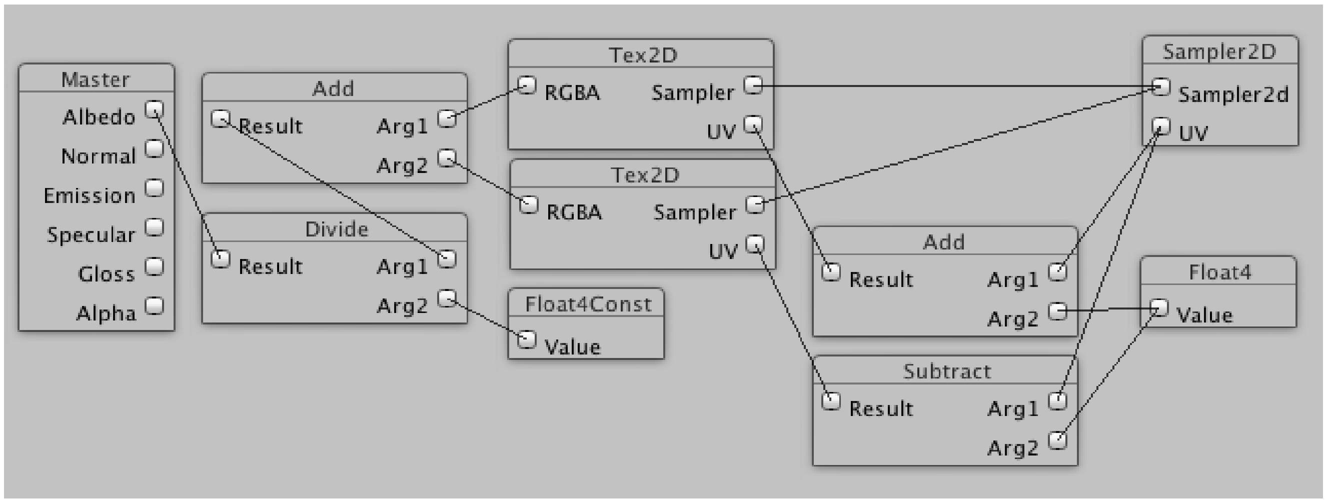


Figure 2-4, the double vision shader graph

Using Operators →

Let's create another effect, known as *Double Vision (Fig2-3),* which is accomplished by reading the same texture twice, with different offsets, then recombining it. This effect is mostly used on organic surfaces, to add variance surface appearance without using additional memory on a detail map. To start, again, create a new shader, as before, name it properly and add your sampler map (Being the main texture, name it *MainTex*), and arrange it so you have a good bit of space to work in. Add two Tex2D, and wire the sampler into both of them, but leave the UV input empty for the moment. Unlike before, where we directly link the texture's UV-set, this time we are going to modify it; under operation in the node list, add both an Add and Subtract operator, aswell as a float4 type input. (Fig2-4)

Float4 (known in ShaderLab as Vector, and in UnityEngine as Vector4) is a four component input type, which we will use to input the desired offset between our textures. Give it an appropriate name, such as *Offset,* and assign a default value- UV coordinates are in the [0,1] range for each texture repeat, so try to keep the default values in the [0,1] range. Connect the Sampler's UV outputs to the first argument of both the add and subtract nodes, and the float4 input's value to the arg2 for both. With this, we now have created two new UV-sets offset by our input values.

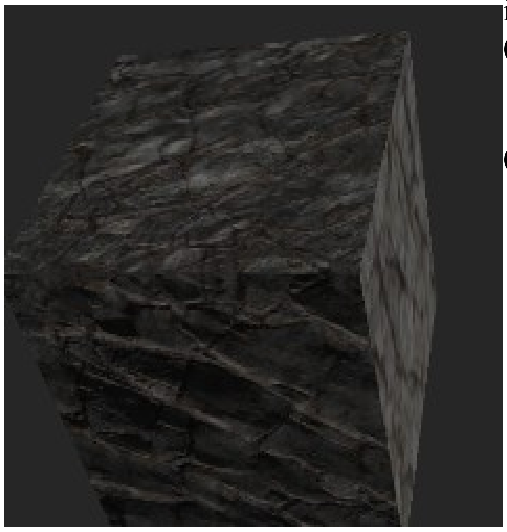
The next step is to actually sample the textures, so wire the results for the Add and Subtract blocks to the Tex2D functions you added earlier. At this point, we now have two texture reads, but the results remain unused. If, at this point, you were to wire the texture output to the Albedo output, you would be able to offset the texture using your new offset parameter in the material settings, however, we need to actually combine the results. There are operators available such as min or max, however since we want the average, the logical approach is to add the results together and divide by two. Thus, to accomplish this, add the new operations Add and Divide, aswell as a constant float4. Add the two Tex2D outputs, then divide by the constant, the values of which you should set to 2,2,2,2. Finally, wire the results to the master.

At this point, it's worth noting alternatives, to help you build better graphs. SSE always swizzles out all arguments into float4 (using a repeating scheme, that is, float is .xxxx, float2 is .xyxy, where float3 is simply a float4 internally), relying on the compiler to strip it down to the proper sizes (which the compiler does quite well). Because of this, you can use a float input and constant instead of float4, and still wire everything correctly, unlike other node based editors like ShaderFX. Further, division is a more expensive node then multiplication, so you can further optimize by multiplying by 0.5 instead of 2. There is also the Lerp Function, which combines two inputs with a given blend factor.

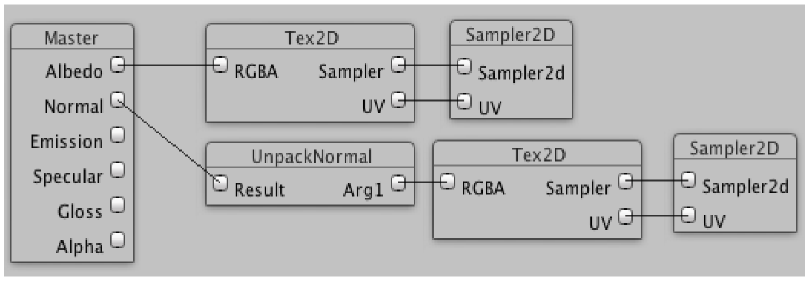
Section 3: Using Secondary Maps

In an attempt to further vary surfaces, we have heralded the arrival of Secondary Maps. These include Normal, Additive, Detail, Illumination, Specular, Gloss, countless others, all aimed at improving the visual appearance of the surface. Normal maps are of particularly common use, being a map that encodes a value by which offset the surface normal, and is commonly used in modern lighting schemes.

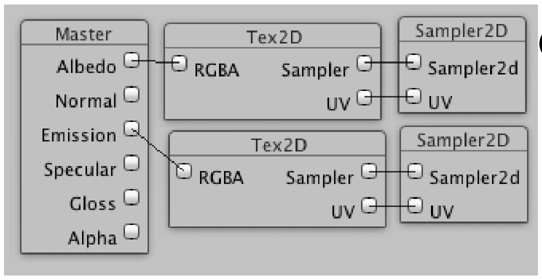
Using normal maps requires a bit more effort then one would expect, as you have to integrate the UnpackNormal function. To set up your graph, create two samplers (*MainTex* and *BumpMap*, as per the naming conventions). For the *BumpMap* sampler, set the default texture to *Normal*, which shall prevent the shader from looking 'wonky' when the texture is not set in the material parameters. Similarly, you should set diffuse maps to *white*, additive maps to *black*, aswell as illumination maps to *black*- the rule of thumb is you try to make an unassigned texture have the least impact on the final result. The UnpackNormal function will format the raw Tex2D result so it can be fed into the normal input for the Master node. (Fig3-1/2)


(Figure 3-1) Output of the normal mapped shader

(Figure 3-2) Layout of the normal mapped shader



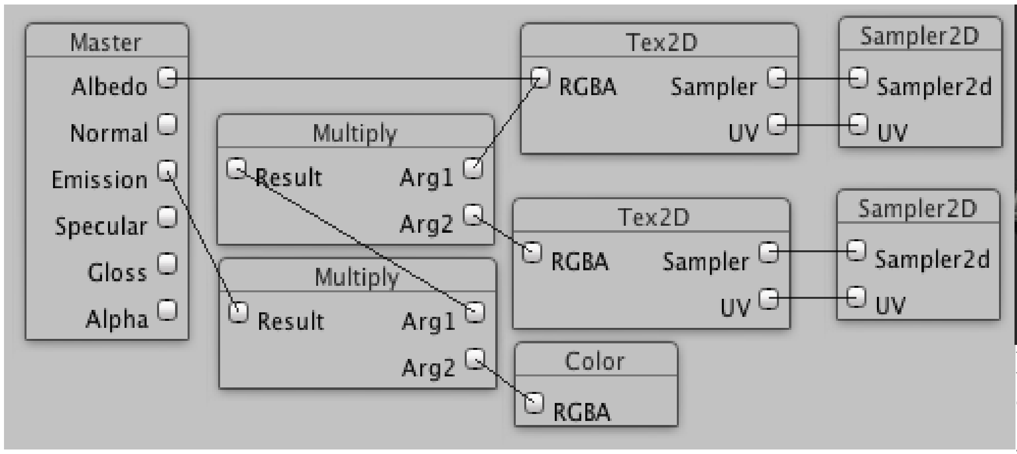
Next on the list of outputs is "Emission"- Emission is a function of how self-illuminating the surface is, and is combined with the texture such to ignore current lighting. Nothing special has to be done for illumination maps, and the simplest programs can provide a texture output directly. (Fig 3-3)



(Figure 3-3) Illumination map Graph

It is not always the case, however, that this will suffice, and I heartily suggest either multiplying the illumination map by the diffuse map, or atleast multiplying it by an input color (which, by convention, would be named *Emission*) (Fig 3-4/5)

Specular maps are an indicator of how reflective the surface is at any given point, and should be expected to vary across the surface of the material, so you usually will see Specular Maps included in most engines. Glossiness, on the other hand, is how "Sharp" the reflections should appear- a high glossiness indicates that the surface has extremely sharp reflections, where a glossiness appears to be slightly rough and plastic. While this can vary across a single surface, it usually is constant (use a Range type input for these), but when combining wet/dry surfaces in a single material you may find yourself wanting to use a full surface map.



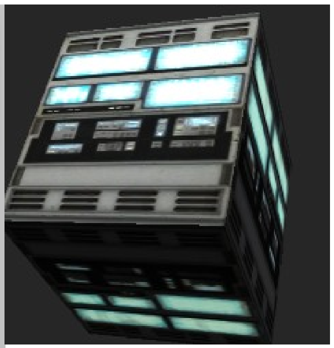
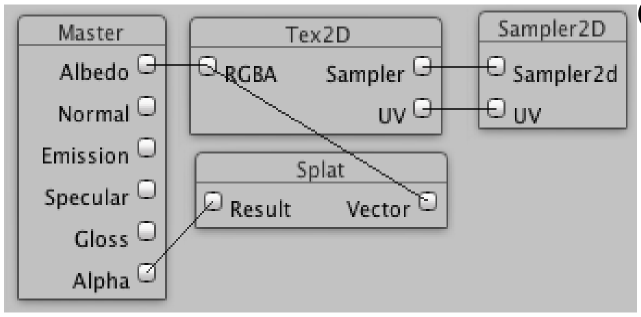


Figure 3-4 (left): The extended illumination graph. 3-5 (up) Result

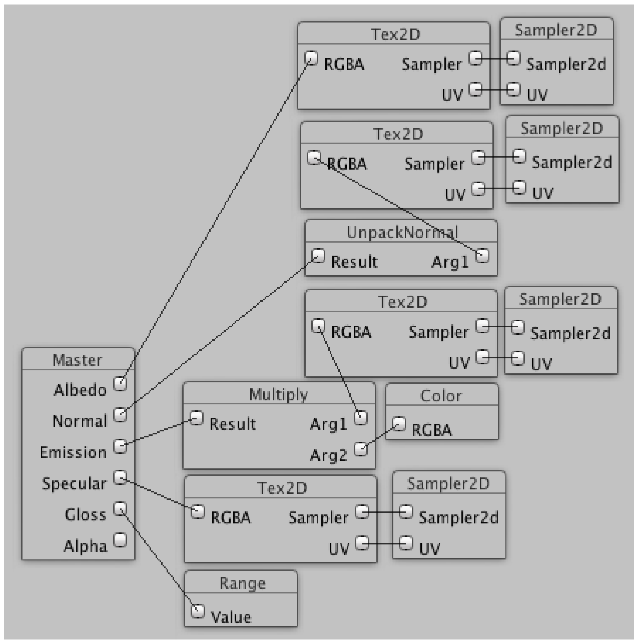
Alpha mapping, unlike the other modes, is generally contained in the fourth channel of another texture. If we directly map the Tex2D RGBA result to the alpha component of master, we would end up mapping the Red Channel (RGBA has components mapped to XYZW, so when used as a single float, it takes the X value, or the R-channel), instead of the alpha channel like we want. To get around this, we can use the Splat function, which takes a specific channel from the original and copies it out over the other channels (Which the compiler will then strip any unnecessary computation from), allowing you to use an individual specific channel. To use it, simply create the node, and select the proper splat channel, in this case, W, then push the result to the alpha component of Master. (Fig 3-6)



(Figure 3-6) Alpha Mapping

SSE assumes that by using the alpha output, your shader is met to be alpha blended (As opposed to alpha-tested), an assumption you can correct by removing the transparent queue tag from the output shader, and adding an alpha-test declaration as documented in the ShaderLab Docs, though for the most part this should be unnecessary.

With all this covered, it becomes fairly simple to put it all together (Fig 3-7), where you have multiple texture maps being read and submitting to different elements of the master node. Be sure to take care with how you lay out your nodes, as it can become extremely easy to create a tangled mess.



(Figure 3-7) Diffuse, Bump, Illumination and Specular with glossiness slider

Appendix: Node listing

Name	Function	Purpose
Constants		
Float	Homogenous Float4	For when you need a vector (or single float) which has all of the same component. (Effectively a scalar)
Float4	Four Component Constant	Same as float, but you can assign a different value to each component.
One	Same as float4(1,1,1,1)	All values set to one, use with Subtraction for a one – value (Invert)
Zero	Same as float4(0,0,0,0)	Serves no common purpose, provides shorthand for sampling the corner texel of a sampler2D
Functions		
Abs	Absolute Value	For each component (Separately!), remove the sign element. See also Saturate.
ACos	Arc Cosine (1/Cos)	Interpolate from [-π,π] to [-1,1] with sharp ends and an eased center. For an sharp center and eased ends, see Smoothstep.
All	If x, y, z AND w are not equal to zero, returns One	Good for splatmaps, to assign holes where there is no texture.
Any	If x, y, z AND/OR w are equal to zero, returns One	Alternative for All
ASin	Arc Sine (1/Sin)	As ACos, except revered (positive to negative, negative to positive)
ATan	Arc Tangent (1/Tan)	Angle of the input slope, however, you should use Atan2 unless you know what your doing
ATan2	Arc Tangent with separate inputs	As the common Atan2 function, returns the angle formed by an input slope. Both arguments take the X of the input vector (so use Splat to disambiguate), Y over X notation.
Ceil	Ceiling	For each component, round up to the nearest whole number. See also Floor.
Clamp	Confine the value into a range	Forces Arg1 to be not less then Arg2, and not more then Arg3- If it is out of that range, it is set to Arg2 or Arg3, respectively. See also Saturate.
Cos	Cosine	Converts [-π,π] to [-1,1] with smooth ends and sharp inner transition, periodically flipped. If you are *not* using it for trigonomitry, you probably mean to use Smoothstep.

Cross	Cross Product	Returns a vector perpendicular to Arg1 and Arg2. If the inputs were not normalized, you will need to normalize this.
Degrees	Convert radians to degrees	Shorthand for multiplying by a constant, see also Radians. This should probably never be used in an optimized shader.
Distance	Distance between inputs	Returns *actual distance* between the two input vectors, be careful about forth dimensional distance (If you have problems, use mask)
Dot	Dot Product	Primarily used as a normalized dot product, in which it returns 1 for matching vectors, 0 for perpendicular vectors, and -1 for opposite vectors, with a sinusoidal falloff.
Exp	Raises to the power of ten	Reverse of Log- this is *not* the same as Pow
Exp2	Raises by a power of two	Reverse of Log2
Floor	Floor function	For *each* component, round down to the nearest whole number
Frac	Fractional component	For *each* component, returns x – floor(x)- that is, repeats the value in the [0,1] range
Fresnel	View Angle	The view angle for the pixel, with an **optional** normal component, to make it sensitive for your normal maps
Length	Magnitude function	Length of a single component, common misconception that this results in the brightness of a color- For that, dot product by your saturation constant (See Internet)
Lerp	Linear Interpolation	Blends Arg1 and Arg2 by Arg3, **Primary function for texture blending.** It is linearly interpolated, so applying a ramp function or exponent to Arg3 can be used to regulate the falloff. **See also Smoothstep**.
Log	Logarithm	Same as Log10 (Possibly fixed in later update)
Log10	Logarithm of ten	Reduce by powers of ten, reverse of Exp
Log2	Logarithm of two	Reduce by powers of two, like the Exp functions, only useful for nonlinear textures.
Mask	Zero specific parts of a vector	Used to strip specific components out of a vector, important for distance and length. Unlike Splat, which takes a single channel across all channels, mask explicitly removes channels.
Max	Return the highest values	For each component of the inputs, return the highest value for each.

Min	Return the lowest values	For each component of the inputs, return the lowest value for each
Normalize	Set Length to one	**One of the most important components**, however be aware that SSE assumes values are provided in float4, so mask may be necessary.
Pow	Raise to a specified exponent	Because colors data is *generally* in the [0,1] range, you can use pow to sharpen any curve function, for example, often specularity is approximated with pow(cos(fresnel),64)
Radians	Degrees to Radians	Opposite of Degrees
Reflect	Vector reflection	Reflects the first vector across the normal of the second vector, you would use this for most envmapping shaders.
RSqrt	1/Sqrt of each component	Reciprocal square root, useful as an optimization over Sqrt where applicable, implemented on hardware as a fast inverse square-root (as opposed to exponential bit-shifting, as regular square-root)
Saturate	Clamp to [0,1]	Used primarily to eliminate negative returns, should always saturate before ramping.
Sign	Returns representation of sign	For positive values, returns 1, for negative, -1, for zero, it will return 0, as always, this is a *per component* operation
Sin	Trigonometric Sine function	Same as Cos but with a 50% offset, most of the time you really mean to use Cos.
SinCos	Combination of Sin and Cos	Fetch both the Sine and Cosine for a given input. **This function is done automatically by the compiler.** Just ensure that your taking both the sin and cosine from the same inputs. There is essentially little to no cost to take the Sin or Cos when your taking the other.
Smoothstep	Smooth Interpolation	Same as Lerp, with a smooth ease in/out. **One of the more commonly used functions,** it is much cheaper then using Sin, Cos, or Pow for a ramp.
Splat	Copy component across vector	**Isolate a single component** across the vector, maps to the .xxxx, .yyyy, .zzzz, and .wwww swizzles. Used for splatmapping (hence name), channel isolation, and, in combination with Mask, custom swizzling.
Sqrt	Squareroot function	Often used for distance calculation or ramping, Implemented internally as an exponential bitshift, making it cheaper then Pow

Step	Threshold Components	For each component of Arg1 and Arg2, will return 0 if <, or 1 if >, effectively thresholding by Arg2. Used to extract masks, or in place of a conditional check. For cards without dynamic branching (or, in most cases, even with it) this is used internally for conditionals.
Tan	Tangent function	Largely used for ramping and constraining ramps. You shouldn't need to use this for actual rotational math, see ATan2 and SinCos for that.
Tex2D	Read from a texture	**Function used to read textures**, reads the assigned Sampler at the position given in UV. See the texturing section for more information.
TexCube	Reads a cube-map texture	Read the value of the cubemap for the provided direction. This is moderately more expensive then Tex2D. See Tex2D, Reflect, and World Reflection.
UnpackNormal	Adapt normal to surfacespace	Scaleshifts the normal value from a color value of [0,1] to the proper range of [-1,1]. Because texture maps don't store negative values, this function adjusts the values accordingly. **Required for reading normal maps**.
UVPan	Selective channel addition	Used in the UVPan shader example as a custom node, adds the selected channel of input to each of the selected UV channels. Normally this would require Splat, Add, and Mask.
Input		
Color	User-Defined color	Set your default such that, when unassigned, it has minimal impact on the surface.
Cosine time	The cosine of time	A smoothly interpolating periodic representation of time, used to sway surfaces, such as water.
Float	Single value input	Single value, expanded to fill all components as by splat or the swizzle .xxxx
Float4	Set four inputs values	These values should be masked and splatted for individual use.
Float2, Float3	Set two or three inputs	These are **not supported** by Unity, use float4 instead
Range	Single value input, clamped	Same as Float, but displays as a slider in the material settings, should be used to keep values in a rational range, whereas float should be used for things like exponents or timescales.
Sampler2D	Texture2D Input	See texturing section for full details
SamplerCube	Cubemap Input	Same as Sampler2D, but for TexCube

ScreenPosition	Position on the screen	Used for overlaying textures in screenspace, aswell as effects like fading things towards the edge of the screen (eg, vignetting)
Sin Time	The sine of time	Reverse period of Costime
Time	Various representations of time	Each component represents a different timescale, some naturally periodic. Use splat to filter out your desired setting.
Vertex Color	Colors provided by Mesh	Vertex interpolated color, using this can be a major memory saver.
View Direction	Direction of view to surface	You *probably* are really looking for Fresnel. This is used for effects like triplanar texture mapping, or directionally specific coverages (Say, semi-procedural moss growing on trees)
World Position	Position in world space	Can be used largely with scripting to create effects that are dependent on global position, such as objects appearing wet when underwater, or to fade near a given location.
World Reflection	World Reflection vector	Used as the input for texCube for reflection mapping, can also be obtained with view direction and reflect.
Operation		
Add	Addition	Add two inputs, performed per component
Divide	Division	Divide two inputs, performed per component. Most of the time you can optimize with a reciprocal function and Multiply instead. Because most values are in the [0,1] range, division as a general rule will make values larger, you should often use this with saturate.
Multiply	Multiplication	Again, performed per component. See Dot for the sum of components.
Subtract	Subtraction	Performed per component, use Saturate to trim for negative values if so desired.
Missing (Not included for various reasons), mostly for people porting shaders		
HSin/HCos/HTan	Hyperbolic Trigonometry	For almost all of you, these won't even have any meaning. They have very specific uses, almost none of which have anything to do with shading.
Round	Rounding	Use Step with 0.5 if you really need it, presumably it was misplaced during development (like Lerp was)
Noise	GPU-Dependent noise	Can be very hardware specific, almost always more expensive then using a noise sampler.

Matrix Functions		Matrixes have not yet been implemented, you can manually rotate textures using a SinCos setup if you need, I've been promised these for a later update.
Modf	Split into integer and fractional	Use Frac and Floor instead
Lit	Compute lighting coefficients	Performed by the surface shader during compilation, you never need to worry about lighting coefficients, use Dot if you really need to emulate.
ldexp	Multiply by a power of two	Can be emulated with Exp2 easily
isnan/isinf/isfinite	Check for specific domains	Never in my professional life in shading have I seen a non-academic use of these functions, if you really insist in needing them, emulate with any, all, saturate and step
frexp	Fractional Exponential split	Again, almost purely for academic use, has very limited use in nonlinearly stored textures. Emulate with Exp and Frac
Fmod	Repeat within a given range	Not sure why this one is missing, emulatable with frac and prescaling
Determinant	Matrix function	Matrixes are not implemented!
Tex1D,Tex3D	Alternate Lookup dimensions	Tex3D isn't exposed for use in Unity, Tex1D can be emulated with Tex2D, or, more often, just a ramp function (Given Tex1D is just a function map)
Tex*Proj	Projected Texture lookups	If you don't know what they are, don't ask for them. Otherwise, you should know how to emulate, though they don't really fit the style of a node editor.
Tex*lod	Pyramidal Texture Lookup	Primarily used to vary sharp/blurry envmaps from the same sampler, and is **DirectX Specific.** If your using it for an anti-aliasing context, it can be emulated with Trilinear filtering on the texture, or *Texture.MipmapBias*
TexRect	Rectangular lookup	Officially dropped in Unity3, emulate with Tex2D
ddx/ddy	Partial derivatives	How fast a given value is changing, almost all of the time you will not need or use this, as it's already handled by texture lookups. Used for anti-aliasing.
Debug	Declare a debugging split	Does not appear to be supported by Unity, Can be emulated by outputting the desired channel to Emission with Albedo set to black.