Shader Graph Markdown

Unity 2019.4 — 2021.2

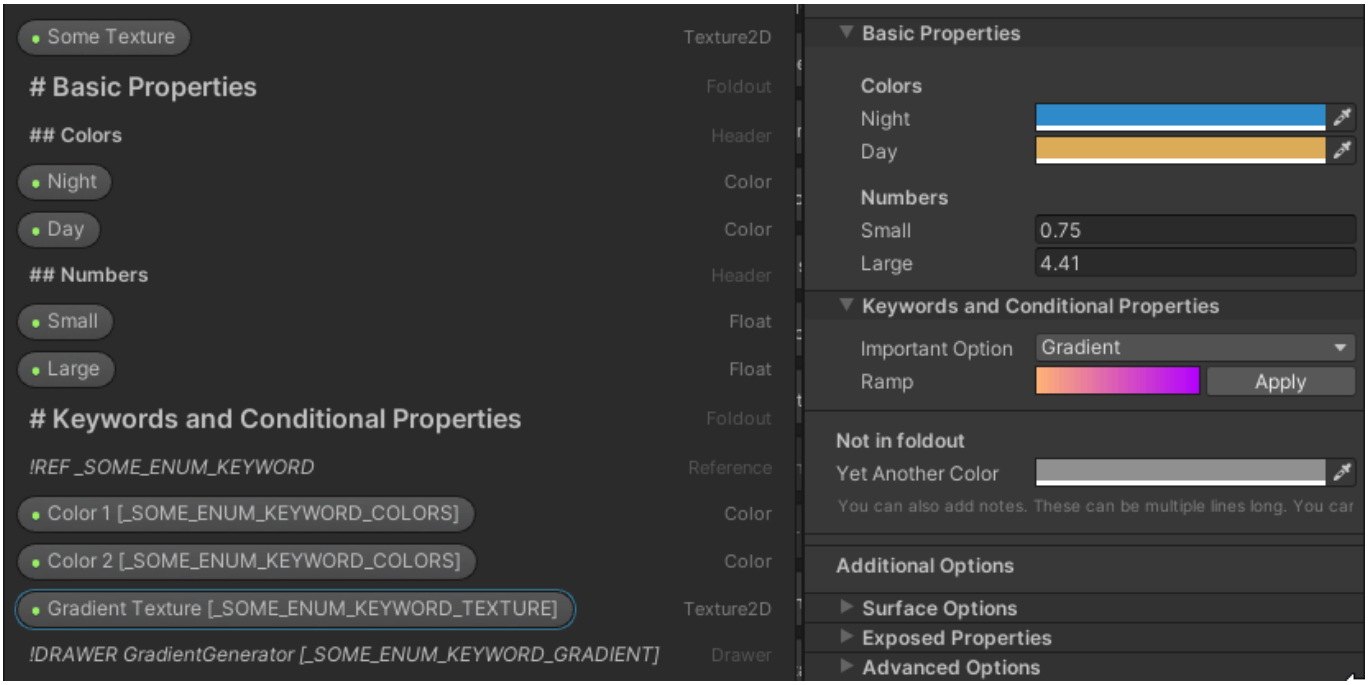
License

Shader Graph Markdown is available on the Asset Store for commercial use.
Other versions are only allowed to be used non-commercially and only if you're entitled to use Unity Personal (the same restrictions apply).

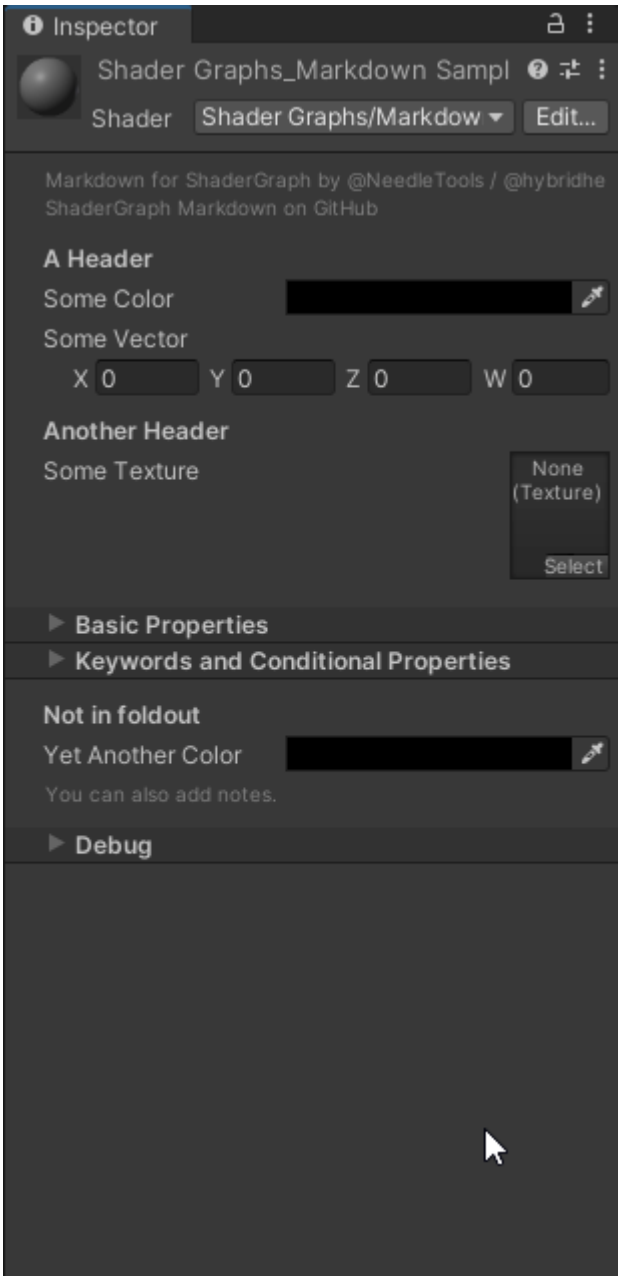
What's this?

Shader Graph Markdown supercharges your creativity by allowing you to create great-looking, easy-to-use custom shader editors right from Unity's Shader Graph.
It uses "dummy properties" to draw a nice inspector for your materials, and decorates the Blackboard (where you edit properties) so that the markdown is readable and looks good.

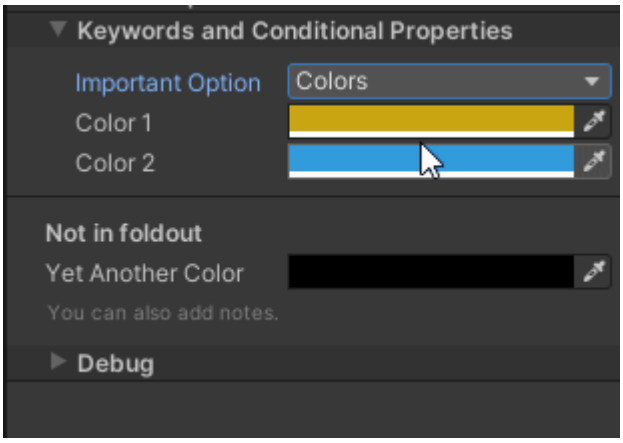
The property naming syntax is inspired by the simplicity of markdown. Please see the Attribute Reference!



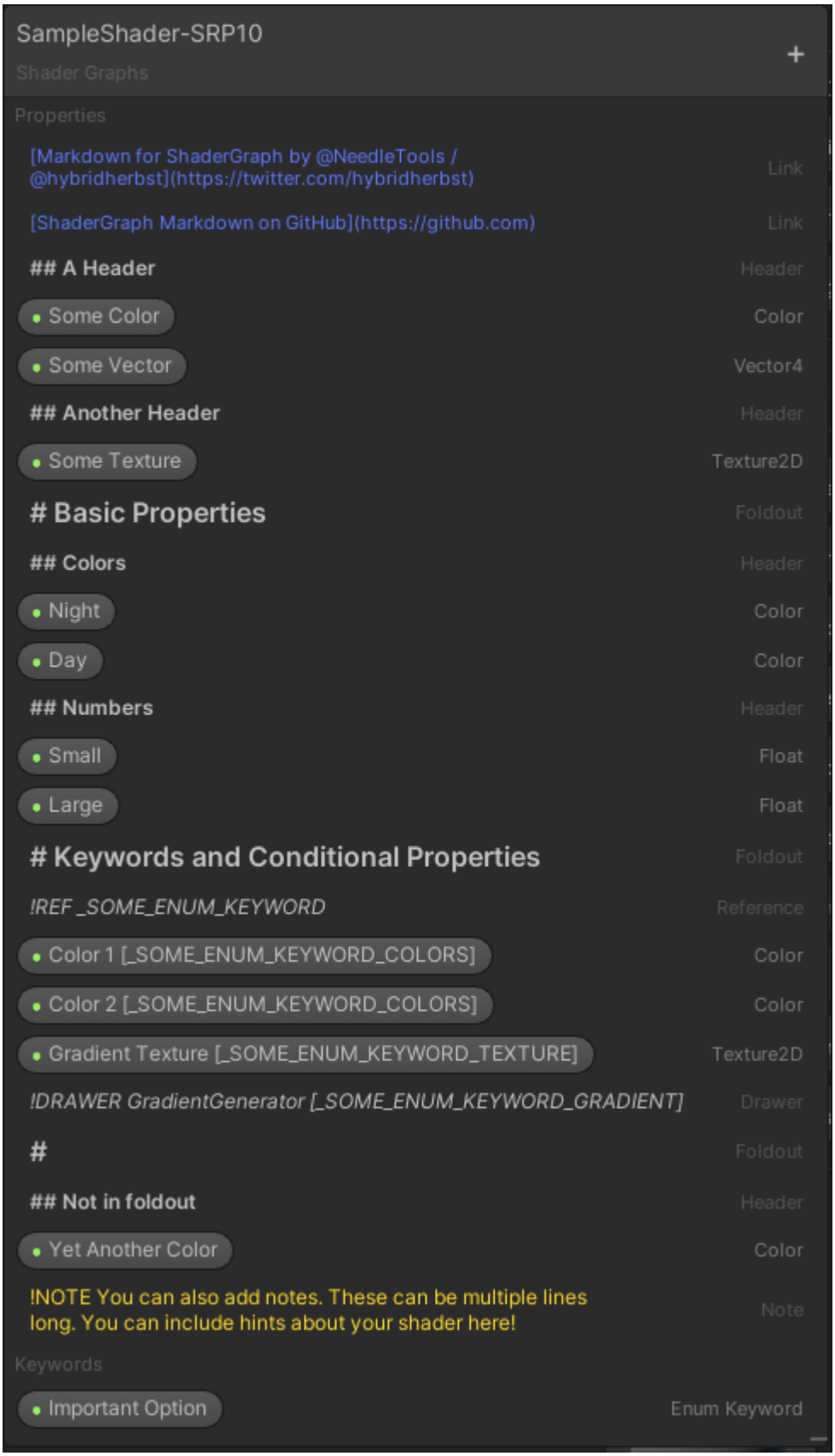
This doesn't affect your shader's functionality - it just makes it much nicer to work with it!
(if someone doesn't have the package, then you just have some extra "dummy" properties)



You can make properties display conditionally, that is, only if a specific keyword option is set:



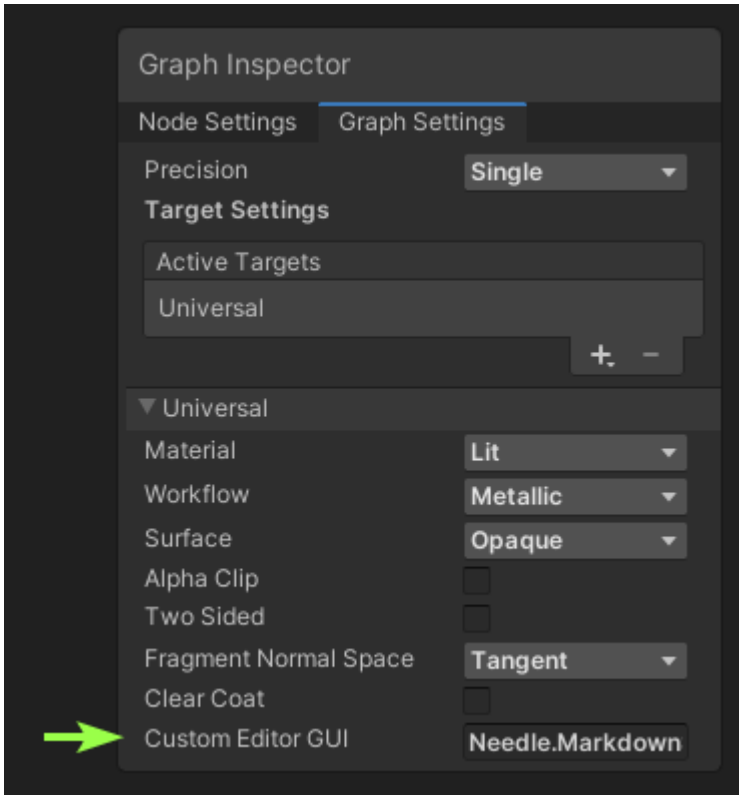
The Shader Graph UI and blackboard are modified to render all "markdown dummy properties" differently, to increase legibility.



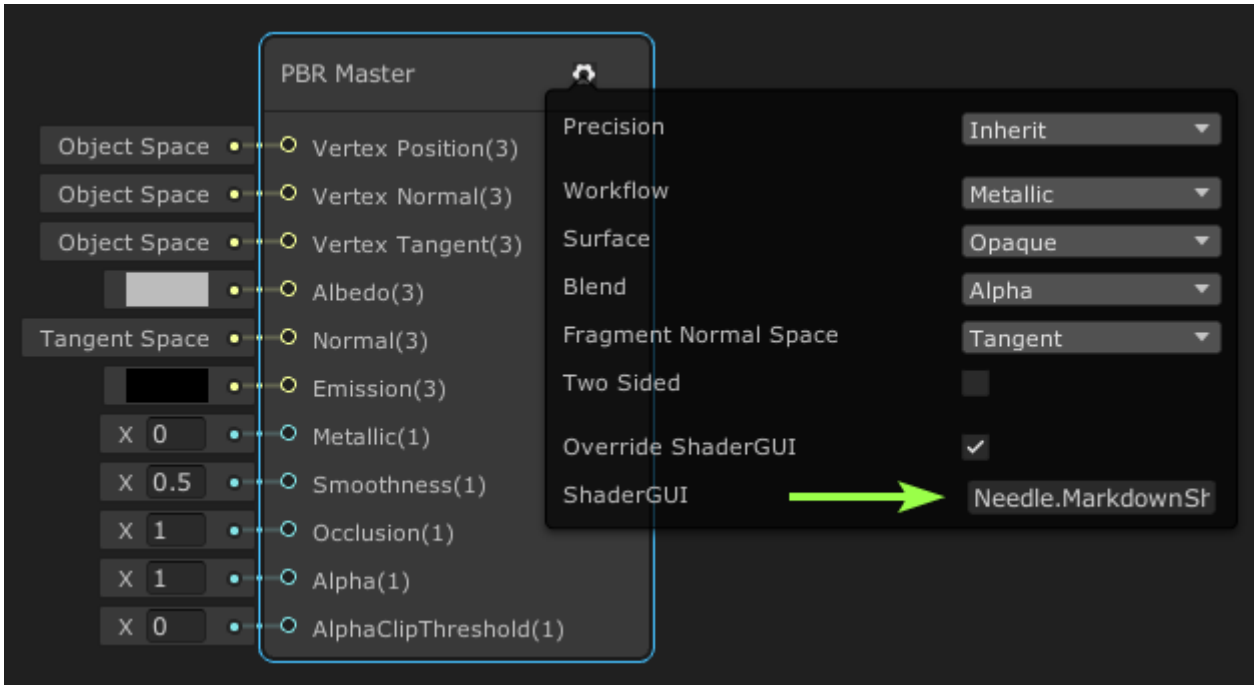
Quick Start

Install via OpenUPM: https://openupm.com/packages/com.needle.shadergraph-markdown/

1. In Shader Graph, tell your shader to use the custom ShaderGUI `Needle.MarkdownShaderGUI`.



Shader Graph 10 / 11



Shader Graph 7 / 8

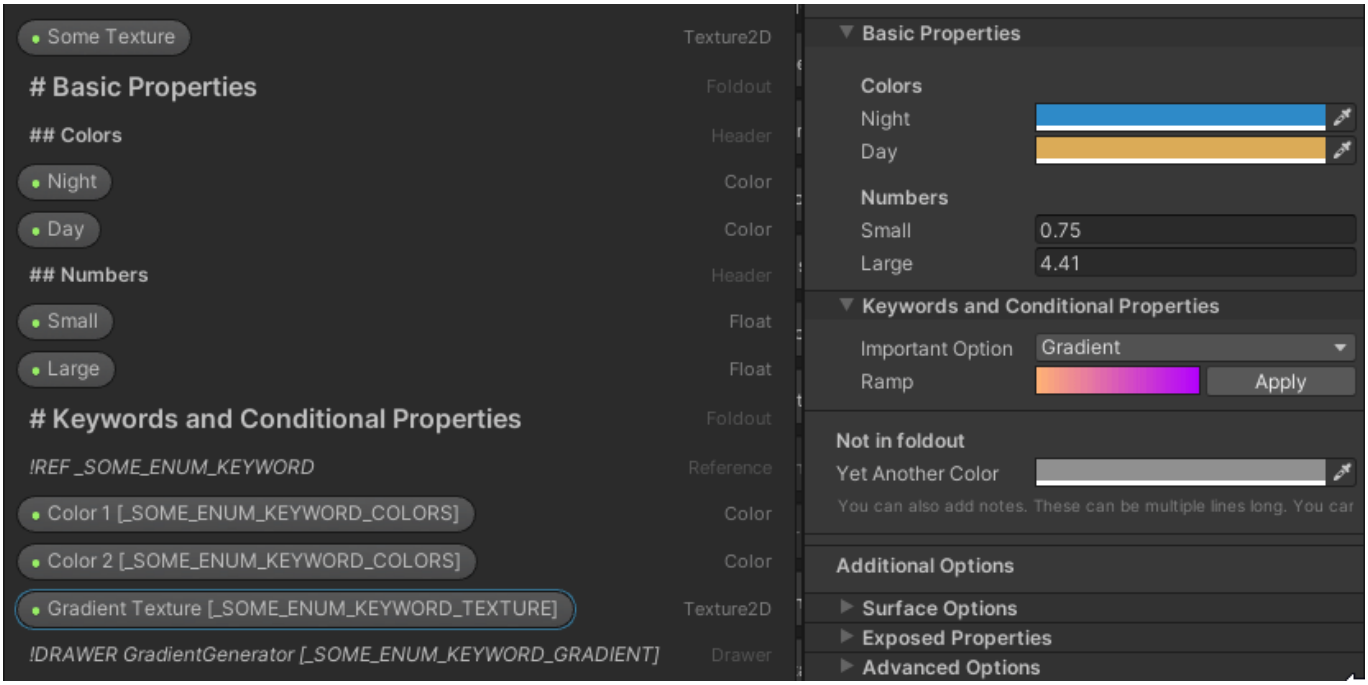
2. Create a new bool property. This is our "dummy" — its only purpose is to specify how the UI is drawn.
3. Name it `# Hello Foldout`. You don't need to change the reference name.
4. You should see the property display change in the Blackboard, to differentiate it from actual properties in your shader.
5. Save your shader, and select a material that uses this shader — done!

To see a more complex example, you can also import the sample shown in this Readme via `Package Manager > Shader Graph Markdown > Samples`.

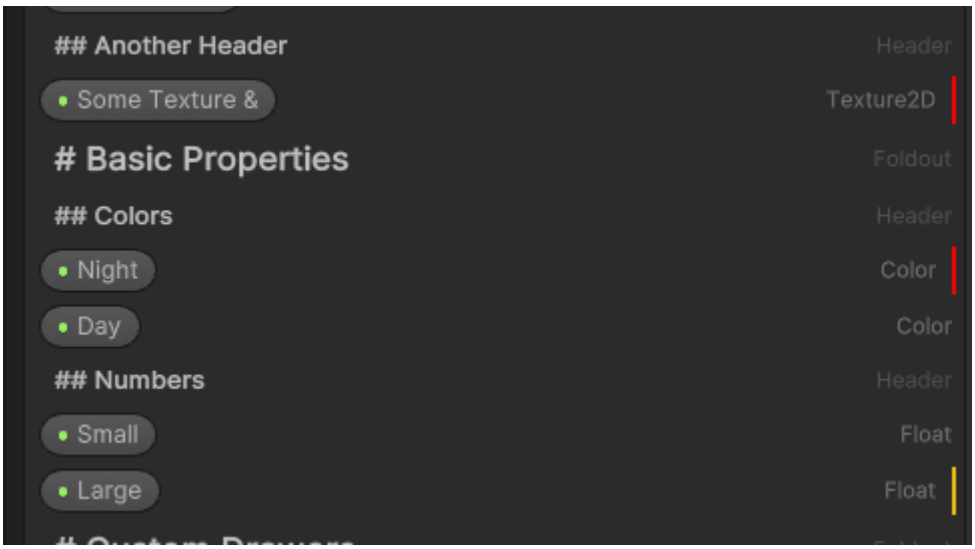
Features

Markdown in your Shader Graph

Simply create "dummy properties", e.g. floats or bools, name them with markdown (e.g. `# My Foldout`) and they'll display beatifully in your Shader Graph blackboard.



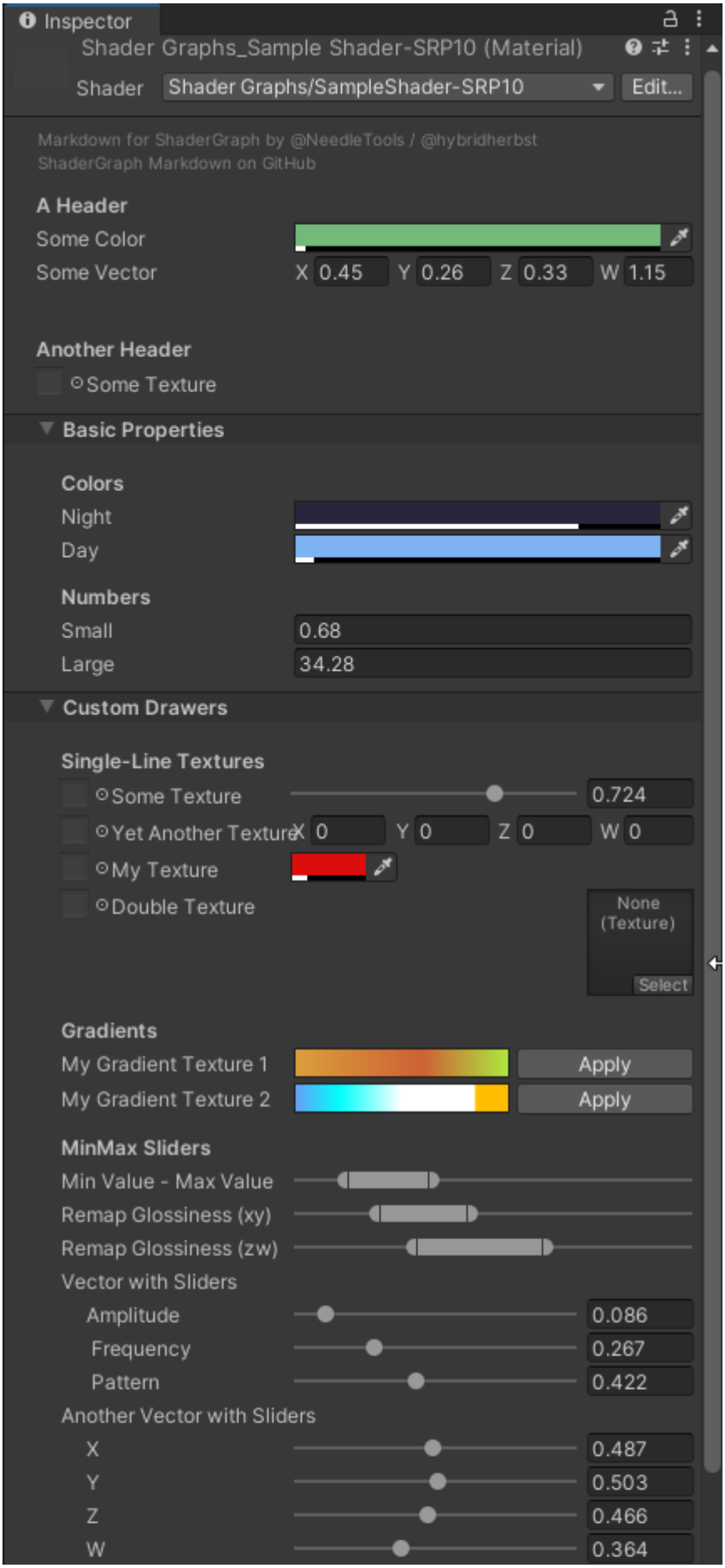
Additionally, Shader Graph Markdown encourages you to use "good" property names and not the default ones, by displaying Blackboard Hints - a little red warning for "you're using the default reference name, don't do this!" and a little yellow warning for "your reference name doesn't start with _, that's recommended!".

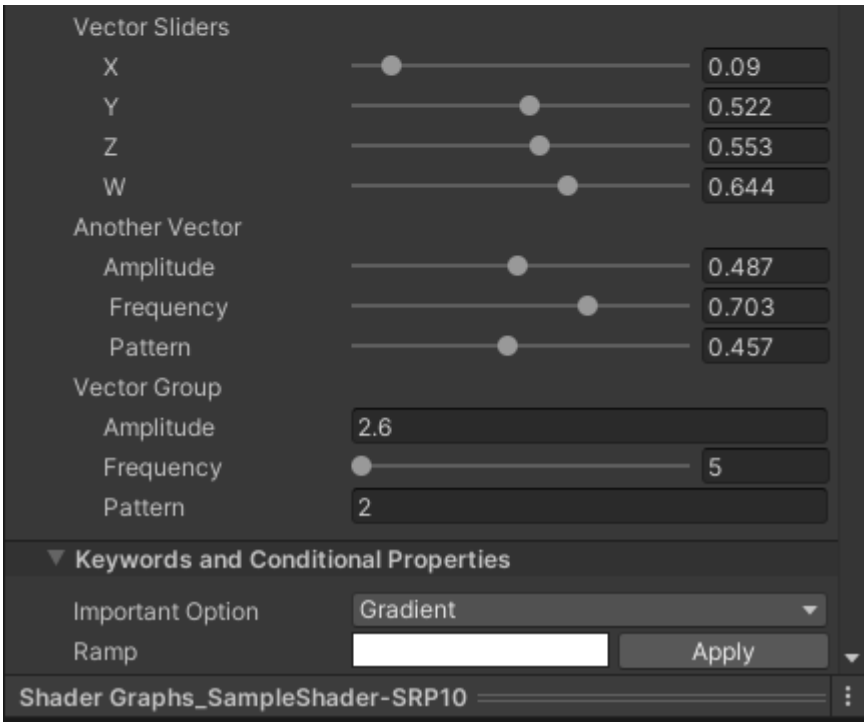


Why? Because using default reference names *will* come back and bite you once you want to switch to a different shader on the same material, start animating material values, want to access anything from a script, etc. - Unity uses those reference names for a lot of things, and it's best practice to get them right from the start.

Custom Shader GUI

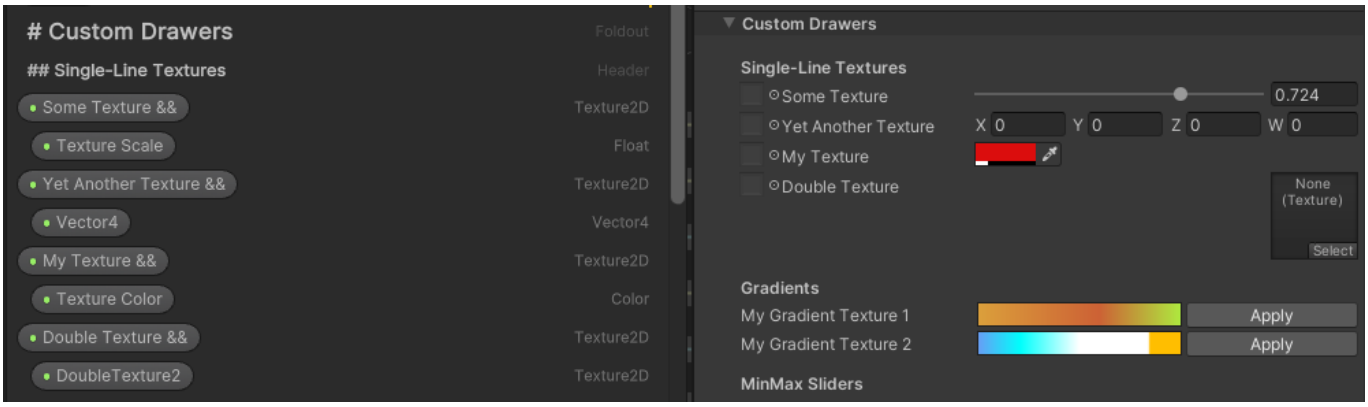
Set ShaderGraph Markdown as Custom Shader GUI to get nice, customizable editors for your shaders - without a line of code, and scaling to a high complexity that would otherwise require writing custom editors.





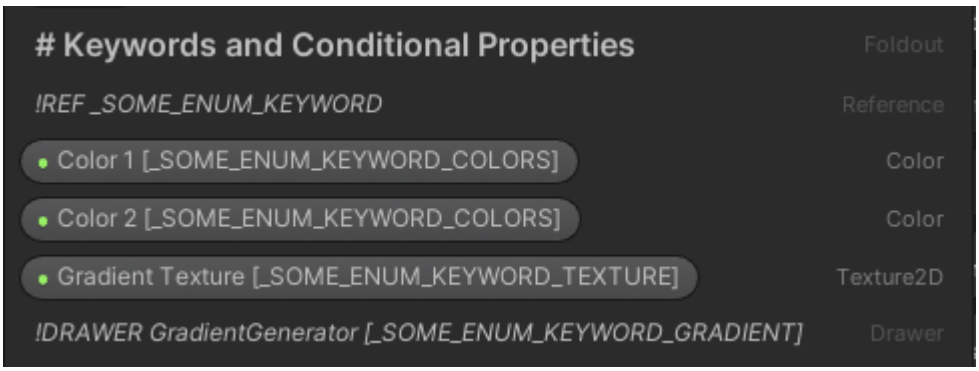
Inline Properties

Textures can have & appended to render them as smaller inline textures. If you use &&, the next property will be rendered right next to it, a common pattern throughout URP and HDRP for drawing sliders, colors etc. right next to a texture property. They'll be auto-indented in the Blackboard.

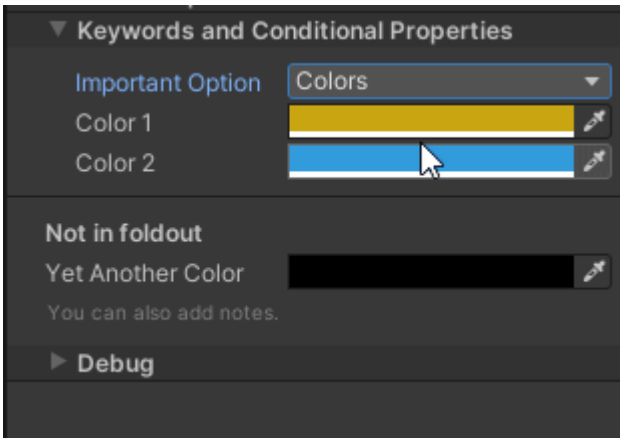


Conditional Properties

All properties — both regular ones and "markdown" properties — can have a condition appended to their display name. This will make them only display if that specific keyword is set.

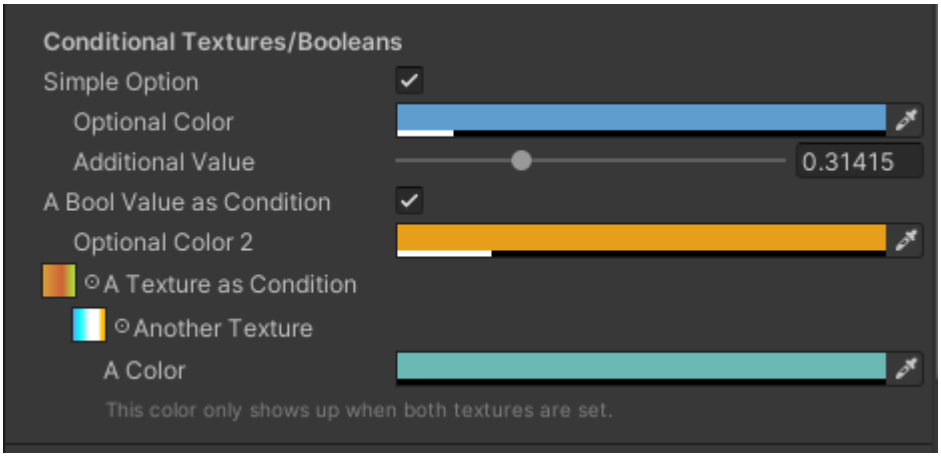


The `!REF` property draws an enum or boolean keyword.
Other properties can specify a "condition" in which to draw them; this is very useful to de-clutter inspectors when you have specific properties in specific conditions.



Boolean/Enum keyword conditions are specified in the form `KEYWORD_OPTION`.

You can also use boolean properties and textures as conditionals:

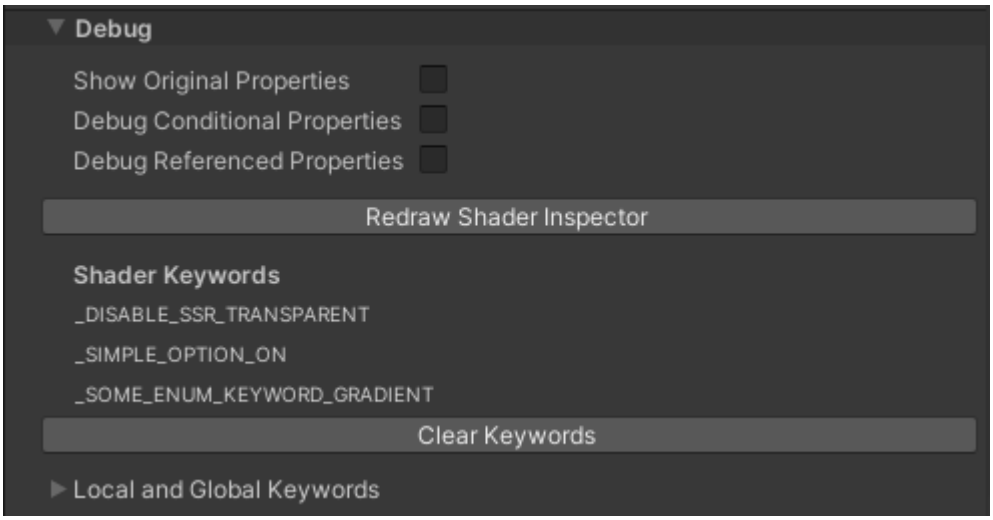


You can construct more complex conditions by using `and`, `&&`, `or`, `||`, `!`, `>`, `<=` ... and other operators - please let us know if something doesn't work as expected!
You can also combine boolean keywords, enum keywords, textures, vectors, colors, floats in the same condition. Vectors compare by length, colors compare by max(r,g,b).

Debug Section

At the bottom of your shader, there's a Debug section that contains info and helpers to configure your custom UI, and to work with keywords in general.

Especially the option to `Debug Conditional Properties` is helpful when you're setting those up, as it will show all properties and allow you to quickly check for wrongly set up conditions.



Expand/Collapse All Properties for URP/HDRP 7

In URP and HDRP 7, properties are configured right in the Blackboard, taking up lots of space. They can be expanded/collapsed, but that takes forever... Shader Graph Markdown fixes this by allowing you to Alt + Click on the foldouts to expand/collapse all properties.

Keywords

Shader Variants in Unity are controlled through Shader Keywords. URP and HDRP use these extensively to control shader behaviour (basically all the produced shaders are "Uber Shaders"). If something goes wrong that can result in wrong rendering, so the options here help you to

- see which keywords a shader uses (local and global)
- see which keywords are currently active
- clear and reset shader keywords to the ones HDRP/URP want to define for the current state.

Quickly enable / disable MarkdownShaderGUI

A context menu entry on every Shader Graph-based material allows to quickly switch the material inspector between the default one and MarkdownShaderGUI. This is useful for debugging (e.g. finding some missing Conditionals).

Custom Drawers

You can totally make your own and go crazy, the system is pretty extendable. The shipped ones (`MinMaxDrawer`, `VectorSlidersDrawer`, `GradientDrawer` etc.) should be a pretty good starting point. When referencing a custom drawer, you can both use `!DRAWER DoSomethingDrawer` and `!DRAWER DoSomething`. Drawers are ScriptableObjects and can thus have persistent settings.

Feel free to jump into our support discord and let us know if you need help or something isn't working as expected!

Attribute Reference

1. `# Foldout`
   A foldout header

2. `## Header` A header, similar to the `[Header]` attribute in scripts
3. `### Label` A regular label, not bold and with no extra space.
   Useful before indented properties.
4. Append `&&` to Texture properties
   - this will render the *next* property inline (most useful for Color or Float properties)
   - if the next property is named `_MyTex_ST` (with `_MyTex` matching the texture property name), a tiling/offset field will be drawn
5. Append `&` to Vector properties to have them display as 0..1 sliders
   - You can optionally specify the slider names: `Vector with Sliders (Amplitude, Frequency, Pattern) &`
   - If you leave them out you'll simply get a bunch of X,Y,Z,W sliders
   - If the vector property starts with `_Tiling` or `_Tile` or ends with `_ST`, it will be drawn as Tiling/Offset property field
6. Append `&` to Texture properties to render them as small texture slot
   (not the monstrous default Shader Graph one, the nice one that URP/HDRP use for everything)
7. Prepend a number of dashes (e.g. `-` or `---`) to indent properties.
   Nice for organization and showing where conditionals belong.
8. `!NOTE Any note text`[1]
9. `[Link Text](URL)`[1]
   A web link.
10. `!REF KEYWORD_NAME`
    A reference to a bool/enum keyword to be drawn here - by default they end up at the end of your shader, with this you can control exactly where they go.
11. Conditional properties: Append `[SOME_KEYWORD]` to your foldouts, drawers or properties to only make them show up when the condition is met. Conditions can be
    - boolean keywords (make sure to include the `_ON` part)
    - enum keywords
    - texture properties (when the texture is not null)
    - boolean properties (when the bool is true)
    - float properties (compare using `<, >, ==` etc.)
    - color properties (max(r,g,b) is used as comparison value)
    - vector properties (vector length is used as comparison value)
12. `!DRAWER MyDrawer`
    This will draw custom code, similar to a `PropertyDrawer` for the Inspector. Drawers are specified as subclasses of `MarkdownMaterialPropertyDrawer`. Examples:
    - Define `!DRAWER Gradient _MyTextureProperty` to render a nice gradient drawer that will generate gradient textures in the background for you.

    - Define some colors, and *before* them add `!DRAWER MultiProperty _Col1 _Col2 _Col3`. This will render three colors all in one line.

13. `#` (hash with nothing else)
    End the current foldout. This is useful if you want to show properties outside a foldout, in the main area, again.

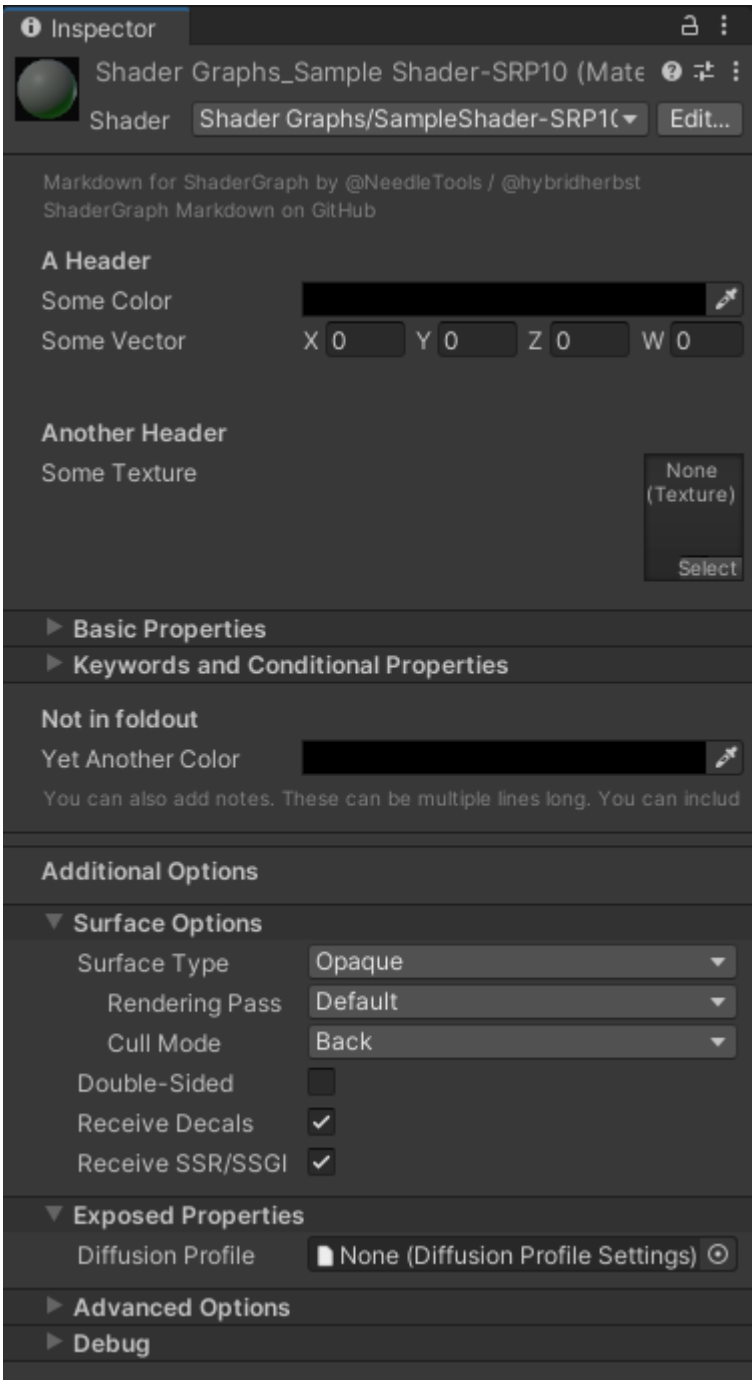
[1]: Will not be shown if the previous property was conditionally excluded.

A lot of the above can be combined - so you can totally do `--!DRAWER MinMax _MinMaxVector.x _MinMaxVector.y [_OPTIONAL_KEYWORD && _Value > 0.5]` and that will give you a double-indented minmax slider that only shows up when _OPTIONAL_KEYWORD is set and `_Value` has a value of greater than 0.5.

Notes

HDRP Support

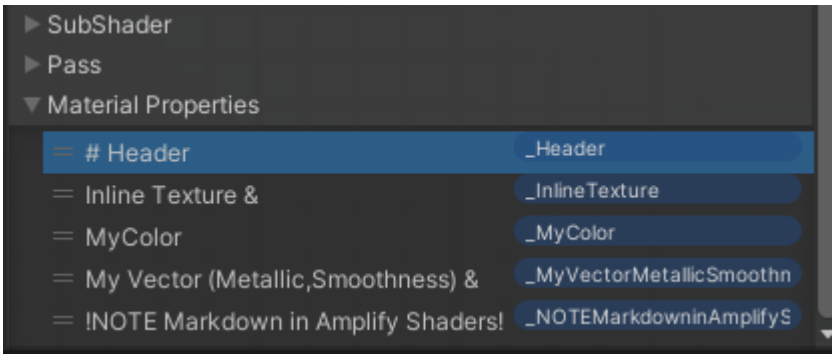
HDRP Shader Graphs are supported. A speciality there is that these already have custom shader inspectors. Shader Graph Markdown finds and displays the "original" inspector in addition to your own properties.

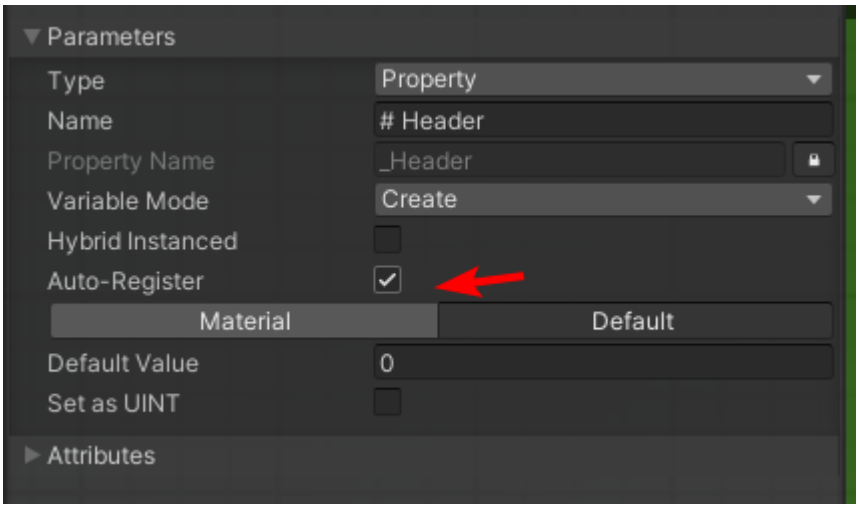


That being said, HDRP does some keyword magic (resetting material keywords at times); if you find something doesn't work as expected, you can use the "Debug" section to reset keywords and/or show the original property list.

Amplify Shader Editor Support

Amplify works as-is, you can specify `Needle.MarkdownShaderGUI` as custom shader inspector and then create properties as usual. You'll have to arrange them somewhere on the board though (they need to be somewhere). Also, turn on "Auto-Register" on the markdown properties, otherwise they'll be stripped away since they are not actually "used" in the shader.





Built-In and other Shaders

Nothing about the custom shader inspector in Shader Graph Markdown is actually Shader Graph-specific! You can use the same properties and drawers for all shaders, be it built-in, surface shaders, BetterShaders, other editors... In many cases, you could also use custom attributes, but especially for foldouts and drawers, Shader Graph Markdown gives a lot of flexibility.

Why isn't there Feature X?

A design goal of Shader Graph Markdown was to keep simplicity to allow for a fast workflow and nice editors, without writing any code. Custom drawers, on the other hand, give a lot of flexibility. The in-between, where you use some kind of markup to describe very complex behaviour, is explicitly not in scope for Shader Graph Markdown.

This is also the reason why there's no options to change header colors, specify pixel spacing values, ... and do other purely stylistic changes.
If you want that, there's other options; one that allows for tons of customization (and thus has a steeper learning curve) is the Thry Editor.

With the above in mind, feel free to reach out via Discord and request features that fit to this philosophy – and please submit bugs if you find them!

Known Issues

- *Changing keyword property entries does not refresh the Enum dropdown in the inspector*
  This is a Unity bug. Case 1176077
  Workaround: reimport any script (Rightclick > Reimport) to trigger a domain reload.

- *Vector fields don't show animation state (blue/red overlay)*
  This is a Unity bug. Case 1333416
- *Conditionally excluded properties can't be inline properties* The condition will be ignored inside an inlined property. Don't inline them if you want them to use the condition.

Contact

needle — tools for unity • Discord Community • @NeedleTools • @marcel_wiessler • @hybridherbst