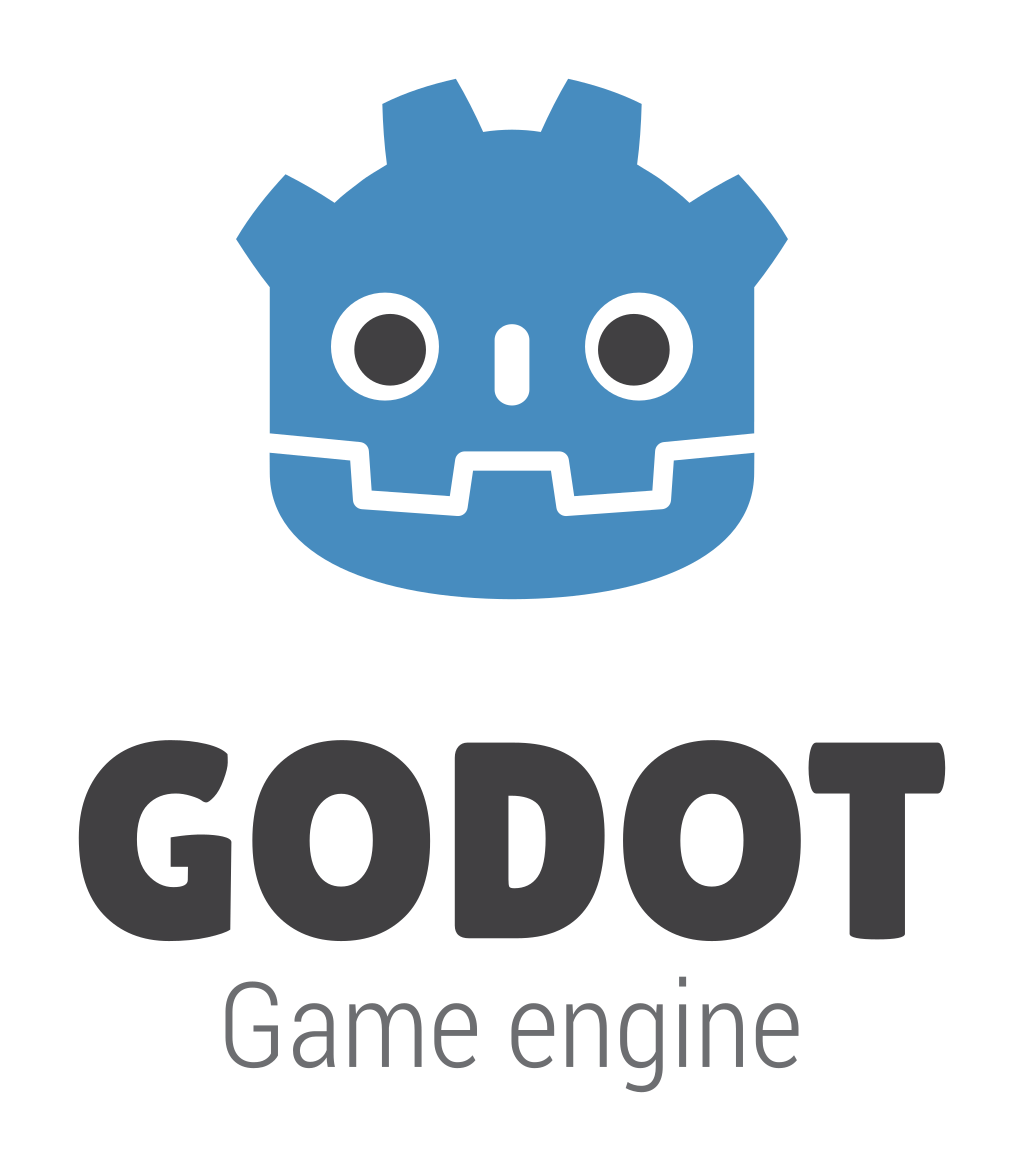
**How to Write Shaders in Godot (GLSL)**



By Benjamin Brown

Contents

[Introduction 3](#_Toc184759238)

[Shader types 3](#_Toc184759239)

[Processors 3](#_Toc184759240)

[2D Shader Samples 4](#_Toc184759241)

[2D Colour picker 4](#_Toc184759242)

[Version 1: Simplistic Colour Picker 4](#_Toc184759243)

# Introduction

Shaders are an extremely versatile tool which can be used to create VFX or to programmatically automate tasks that would otherwise take up valuable visual art resources such as recolouring assets. Different game engines have their own methods of exposing shaders to the developer, Unity, for example, presents the user with a GUI in the form of their shader graph that allows the creation of shaders without any extra code. But Unity, and most other game engines, also support “GLSL” or “OpenGL Shading Language” for users to write their own more custom shaders.

In Godot’s case, there is no GUI, only “Godot shaders” which is based on GLSL with some added quality of life features and Godot specific integration functionality. Godot has a dedicated page for converting GLSL into Godot shaders. This guide will be aimed at how to use Godot Shaders in Godot but it is good to know that this knowledge is transferrable to other engines. It is also important to not be intimidated by it being a different programming language. The basic syntax is identical to the C family of languages which should make it quite accessible to people with prior programming experience.

## Shader types

Within Godot, there are 5 types of shaders. The first line of a shader must state the type of shader that the file is, but this line will be automatically generated when creating a Godot shader from the editor.

The 5 types are:

* “spatial” for 3D rendering
* “canvas\_item” for 2D rendering (this includes UI/control nodes)
* “particles” for particle systems
* “sky” for sky rendering
* “fog” for rendering fog volumes

This guide will not be covering sky and fog shaders but the principles are transferrable.

## Processors

“Processors” are processing functions in the shader called by Godot. Within these functions, you need to call your logic to apply it to the material. They are not all relevant, and their behaviour may differ depending on the type of shader. Each processor could have its own section, but I will cover them as they become relevant to samples.

The 7 different processors are:

* “vertex()” runs over all the vertices in the mesh and sets their positions and some other per-vertex variables. Used in canvas\_item and spatial shaders
* “fragment()” runs for every pixel covered by the mesh
* “light()” runs for every pixel and light, used in canvas\_item and spatial shaders
* “start()” particle system only, runs once for every particle when it is first spawned
* “process()” particle system only, runs per frame per particle
* “sky()” not relevant, only used in sky shaders
* “fog()” not relevant, only used in fog shaders

# 2D Shader Samples

This section contains a couple of sample shaders with the logic behind them. As with all samples, the accompanying project can be found and downloaded [here](https://github.com/BenDBrown/Godot-Shaders-Tutorial) for you to look through as an example. It is important to note that “Control” the base UI node in Godot, and “Node2D” the base 2D transform node, both inherit “CanvasItem” which means that if you are making a 3D game, but you want to use shaders in your UI, you will need a canvas\_item shader for your UI nodes.

## 2D Colour picker

A pixel art of a green object

Description automatically generatedThe need to have multiple colours of an asset, is a common one. This was actually the reason I started researching shaders in Godot. Luckily it is also a great starting point, a minimalistic version can be achieved in just 8 lines of code! I am going to make 2 versions with some pros and cons. I also have a wrapper to show how a shader can be modified from a normal script, but I will cover this in a different section. To find this sample, download the project, open it in Godot and then search for the file “ColourPickerSample.tscn” in the file search, and open it. You should see the 3 sprites below in your scene view:

### Version 1: Simplistic Colour Picker

The right copy of the sprite shows the original version of the sprite, the other 2 are using the 2 different versions of the shader. The first version of the shader is for the sprite on the left. As you can see, it colours the entire sprite, regardless of the base colour. You can open the sample project to look at how this works, but I think it always helps to do it yourself. So, make a new project and create an empty 2D scene. Add a Spite2D and give it a sprite, I recommend copying the sample sprite from the demo project, as I chose it to demonstrate the differences between this shader and the next.

1. Open the newly created Sprite2D in the inspector and expand the “Material” tab under CanvasItem.
2. Click on where it says <empty> and then select “New ShaderMaterial”
3. Click on the material preview and then next to the “Shader” field click where it says <empty> and select “New Shader”
4. If you created the shader with a template you will see the vertex(), fragment() and light() processor functions. These are the only ones called on the Shader so if you make new functions that you want to use they need to be called in the corresponding processor function or they will have no affect.
5. If you have the template, delete the light and vertex functions, as we won’t need them for a colour picker. Else if you did not select the template, add the following to your file:

A screen shot of a computer code

Description automatically generated

1. Under the shader\_type, make a new variable of type vec4 and make it uniform like this:



#### Uniforms

Uniform variables in gdshader, are variables that can be passed in via either code or the inspector. They are treated as constants within the shader themselves, so they can only be modified at runtime via an external script. Most of the time though, the values are set in the inspector during development, and do not need to be changed. Changing uniform values in code will be covered later in the document.

#### vec4

“vec4” or Vector4D is a struct made up of 4 float values. In shaders these are used to represent colours as RGBA values which stands for Red, Green, Blue and Alpha. Each value is a range from 0-1. This means that something which is pure green can be represented as (0, 1, 0, 1) with the second 1 being the alpha/opacity of the colour.

1. If you save your shader and look in the inspector (you may need to press build first), under the material you will see a new drop down of shader parameters, this is where you set uniform values in the inspector. You may also notice that the field you added requires 4 numbers. You can fill in all inspector colours as RGBA values but there is a better way. Go back to your shader and replace the semi colon at the end of your variable with “: source\_color;”. If you save and look at the inspector again, you will see that the inspector now shows a colour that you can click to set the value via a colour picker menu that supports a load of quality-of-life features like colour hex codes.
2. Now moving into the Fragment function, we need the colour of the currently processed pixel to do this we need a variable inside the function, you can call it whatever you want but in the example I called it pixelColor we then need to make that variable equal to the result of the texture() function which is built into gdshader. It should have the variables TEXTURE and UV like so:

A black background with white text and symbols

Description automatically generated

#### Texture

The function “texture()” takes two arguments, The first argument is the texture to sample and the second argument is where to sample it. The variable TEXTURE is a constant that simply refers to the texture which the connected material is on which is just the texture assigned to the Sprite2D in our example.

#### UV

The second argument of the “texture()” function is the coordinates to sample on the texture. UV is a variable who’s value refers to the coordinates of the pixel that is currently being processed. Remember that gdshader follows the same coordinate system as Godot which means that 0,0 is the top left of the image. It is also good to know that much like RGBA values, texture coordinates are also a value between 0 and 1, which means that if you passed in your own vec2 into the texture function with values above 1 or below 0, it would just round the values to 1 or 0 respectively.

1. To preserve shading and empty space on the original asset we need to modify the provided colour value before assigning it. Which means we need the grayscale value of the current pixel, and we then need to multiply that with our uniform “color”. To do this we first need the brightness of the pixel which we get like so:



Note that this is just the average of the colours so 3.0 is just the count of red green and blue.

1. Then we convert these values into a vec4 like so:



Note any colour with equal RGB values will be a shade of grey, also note that we do not modify the alpha value as we do not want to change the opacity of anything in this shader.

1. Final step: we make the desired colour by multiplying the grayscale value with the uniform “color” and then we set the “COLOR” value to be equal to that new colour like so:



#### COLOR

This variable refers to the colour of the current pixel. Note that we read the colour using the texture(function) as we may want to determine the value of a pixel on the basis of the value of another pixel. For pure recolouring this isn’t relevant, but this is very useful to keep separate in a lot of use cases.

#### Check your work

The final result should look something like this:

A screen shot of a computer code

Description automatically generated

You may notice when recolouring with this shader, that it recolours not just the grey parts but also the details that were in different colours. That is fine if you want to just recolour an entire asset, but if you want to have some colours preserved, keep reading into the next sample.