2D SHADER DEVELOPMENT

FOUNDATIONS 01

MAKE YOUR GAME UNIQUE IN A WORLD FULL OF LOOKALIKES

FRANCISCO TUFRO

WITH EXAMPLES IN UNITY

Dedicated to Walter Tufró

Who taught me that you have to pursue your dreams in life.

2D Shader Development: Foundations

Published by Hidden People Club

Table of Contents

Introduction to the series

Motivation

During several years of developing my own games as a solo dev, with Nastycloud and, now, with Hidden People Club, I found there were little to no sources of organized information about how to use the power of shader programming specifically in the context of 2D games. Every single shader course or book out there talks about 3D lightning, 3D texturing, shadow, and light mapping, etc. But none of them provided a good section on 2D. I get it though, 2D is kind of a subset of 3D when we talk about computer graphics. Also, in general, computer graphics books are targeted to engine creators, which usually work on 3D.

From giving workshops about this topic in Argentina and the United States I found out that there are a lot of people that are not ready for the 3D math behind computer graphics, but that still can benefit from learning a leaner version of the topic specifically designed for 2D development.

The techniques I describe in the book are the consequence of my own experience in the topic and taken straight from the trenches. Almost all of them have been used in Nubarron: the adventure of an unlucky gnome and other smaller projects. So I thought it would be a good idea to sit down and organize all the information I've been collecting and figuring out during the last 4 years and share it with you all.

As I mentioned, the content in the book series has already been taught in several workshops during 2014-2017. I've updated, expanded, sorted, and enhanced it during these 3 years, and my plan is to continue making it better.

I use Unity, why should I bother learning shader programming at all?

The video game industry is reaching a point where you need something new to stand out. We can't just develop whatever, launch it and expect it to make money for us. Unless you're doing a whatever simulator, those seem to work for some reason. But for the rest of us, trying to stand out in a crowded space, we need to create games that play and look uniquely, at least to some degree.

Shader programming is one of the most important areas of game development that binds visual art and technology. It makes both worlds make sense of each other. Visual artists may have great ideas in mind, they have the means to create fantastic looking worlds, but none of them will run at 60 fps unless a programmer that understands shaders is in the mix.

Not only that, if you know how to program shaders, you can aid the visual arts team in deciding which things make sense and which don't. There are many things that are way easier to achieve using a simple shader than having animators do them. Combining several techniques you can achieve great results without much processing or memory effort, which is key to having high performance in games.

Using Unity is great, but if you limit yourself using stock shaders (the ones that come with Unity), you'll be missing a huge opportunity to make your games look unique and perform at a good frame rate.

Just as an example, in Nubarron: the adventure of an unlucky gnome we used Spine as the main software for skeletal animations. But there is a major issue regarding the amount of processor that the Spine runtime needs and the amount of garbage it creates in memory. From a pragmatic standpoint, we couldn't have more than 20/30 animating objects on screen. We usually surpass that by quite a lot, especially in the background foliage layers, where every single asset is moving. So, instead of animating the foliage using Spine, or a sprite sheet (which would consume too much memory), we created a generic shader that creates a wave that moves the asset from one side to the other, like responding to wind changes. That was a great choice. It works really well visually and does not consume any CPU time.

I created this book using Unity 2017.3.0f3, which may not be the latest one when you read it, so some things may have changed slightly (but it's unlikely!). Please get in touch if something breaks and I'll see to upload a fix in the Github's code or the series website.

Who are these books for?

I think the books are for anyone developing video games. It is of special interest to programmers, obviously, but also to artists and producers, because it gives an introduction to what's feasible, and what kinds of techniques can be used to achieve certain effects.

If you have never ever done any computer graphics programming, this could be a good way to dive into the topic. I'll ignore most of the linear algebra needed to understand 3d transformations and such, there are several resources that cover this topic, and I get nervous when I see computer graphic books starting with one or more 'Linear Algebra recap' chapters.

If you lean more towards the theory, I'm afraid this book may not be for you. There is some theory in the book for sure, but the minimum amount in order to make the reader understand the practical discussion. The book is aimed at the pragmatic programmer that wants information a little bit more digested than regular textbooks.

I need help! What can I do?

First of all, Don't Panic. It's normal to get stuck while learning something new, and I'm here to help you. The first thing I'd suggest is that you join us on our Hidden People Club Discord server https://discord.gg/776BVVD if you haven't already. I am using it to have organized discussions about the book and its content.

Once you're in be sure to use the right channel to send your questions (#2dshader-development). I'll be monitoring the channel to help you on your path to learning these materials.

I also encourage you to share everything you create in the server too, I'm always delighted to see the creations made with my teachings as a starting point.

Be sure to also follow me on @franciscotufro and ping me if you need anything, my DMs are open.

Series Overview

I decided to cover several topics that I think are of special interest when starting working on 2D. These topics were useful to me when working on games in the past, and I consider them part of my everyday developer toolkit.

Foundations

In this book I'll make an introduction to shaders, explaining what the GPU is and what role the shaders play in it.

After understanding what a shader is, we'll dive into how to apply and use shaders in unity. We'll also learn what's the general structure of a ShaderLab program, Unity's own language for shader creation.

Then, we'll dive into Fragment shaders, we'll study the difference between a fragment shader and a vertex shader, we'll talk about colors, RGB color representation, UV mapping, and we'll write a few basic shaders, from a solid color shader to a textured shader with movement.

Finally, we'll discuss blending modes. How we can rely on them to mix between two textures, between a texture and the screen and how to make sprites transparent.

2D Illumination

In the 2D illumination book, we'll focus on figuring out different techniques to give life to our games through the use of illuminations. We'll cover the most basic and widespread techniques for static lights and shadows, that will give us an easy and cheap way to create an ambiance that integrates with our characters.

We'll also cover dynamic 2D lighting. With the aid of specifically crafted normal maps, we can rely on existing 3D lights to create interactive sources of lights, that will provide a really amazing look to our games.

Procedural Texture Manipulation Book

In this book, we'll dive into how we can modify and mix existing textures to create amazing effects or animations inside our shaders. This will give you the tools to be able to implement things that were unthinkable before because you'll be able to do some of the things animators do in After Effects that are impossible to import in a frame by frame basis. We'll use several techniques including sine waves, smoothsteps, color offsetting/chromatic aberration and more.

Full Screen Effects Book

In this book, I'll introduce you to a widely used technique where you apply a shader to the rendered screen. In this shader, you can use all the techniques from the other books to achieve amazing looking full-screen effects.

We'll make a special emphasis in implementing a Bloom effect from scratch, Camera Shake, Retro-looking effects, and more.

Downloading the source code for the exercises

All the source code for the exercises can be found on GitHub, with MIT License (so you can actually use it in your project, except for the assets).

This book's repository is https://github.com/hiddenpeopleclub/2dshaders-book-foundations-exercises

If you are familiar with git, you can clone the repository as usual. If you don't know anything about git or don't want to install it, you can download a zip file containing all the files from https://github.com/hiddenpeopleclub/2dshaders-book-foundations-exercises/archive/master.zip.

Introduction to Shaders

In this chapter, we'll learn a little bit about the context in which shaders exist. What is a GPU and why we need one (or more!). We'll also define what a Shader is, which kind there are and what are they used for.

What is the GPU?

GPU stands for Graphics Processing Unit, in other words, is a piece of hardware designed to handle graphics-specific tasks. We can go as far as the 70s to find that people were already working on graphics-specific hardware, old arcade systems had their own graphics chips to handle rendering in their big screens. But we're more interested in the latest generations, that has been evolving since the nineties when 3d games started to get more and more complex and required too much computation to work on the CPU. That's when we started buying a separate process unit, the GPU. The GPU, as I said before, is a piece of hardware that is specifically designed to handle graphic-related tasks. Well, that's half-true though, since modern GPUs can also be used for other kinds of processing, but their original intent was graphics.

You may be wondering why do we need a specific hardware to do graphics? Why can't our 3 gigahertz quad-core mega powerful CPU handle this? The answer is simple: Concurrency. A standard resolution nowadays is 1920x1080. Doing the math you'll see that you have about 2.073.600 pixels on the screen. We can also agree that we want our games to run at least at 30 fps (usually 60, or more with VR!), which for us programmers means our frames need to be rendered in less than 33 ms. If you take into account all the processing that's needed to render a full-screen world, with say 200-300 objects of many hundreds or thousands of triangles each and fill 2 million pixels, all of that in less than 33 ms, you'll see how easily we run out of time. The major issue is that the CPU can't do much work concurrently. Of course, you can use several cores and processors, but you're always bound to a low number of threads that can go in parallel, say 4 or 8. There is not much difference in splitting our game rendering logic between 4 or 8 concurrent threads. So that's where the GPU comes in.

GPUs are designed to be extremely concurrent. They have less processing power per thread than a normal CPU and have fewer instructions, but they can do millions of operations concurrently. Since most of the processing required to color each pixel can be done in an isolated manner, one can rely on heavy parallel hardware to speed things up. In practice, one can program the GPU to do all the processing for each vertex and each pixel almost at the same time. There are obviously certain restrictions, but the overall speedup is enormous. This parallel nature of the GPU ended up being extremely useful for other things, like scientific calculations and even bitcoin mining back in the day. With a simple google search on General Purpose GPU (or GPGPU), CUDA (Nvidia's library for general purpose GPU) or OpenCL (An open standard that is used to program GPUs among other platforms) you'll find tons of information on ways to use the GPU other than rendering.

Now that you know what the GPU is and why we need it, let's see how to program it using shaders.

What is a Shader?

In the previous section, we learned that in order to do graphics at full speed we need to make use of the GPU. But how? Graphics APIs like OpenGL or DirectX use what is called Shaders. Shaders are small programs that are loaded into the GPU to process the data we send to it. All that is performed concurrently, by the many threads present in the GPU. Using shaders you can apply a texture to a model, show a 2D sprite, do special effects, perform deformation on models and an infinity of things more. In this course we'll pay special attention to one type of shader, the fragment shader, that is the shader that runs at least once per pixel on the screen.

Just so that you are aware, there are other types of shaders. The most used along with fragment shaders are the vertex shaders, which run once per vertex of your 3D or 2D models. You can also find tessellation shaders, that are used to create or remove triangles from existing ones, geometry shaders, that work on patches of several triangles, and compute shaders that are used outside the pipeline to calculate things in parallel. As I said, we'll only make use of fragment shaders, and mention vertex shaders a couple of times for different reasons, but we won't dive into other types of shaders at all.

As a programmer, you'll need to learn a specific language to program your shaders. But don't worry, they're all pretty much C. Different APIs use different languages. For example, OpenGL has GLSL, DirectX has HLSL, and Unity uses a language created by NVIDIA called Cg, which is based on HLSL, but can be compiled into different shader languages (including HLSL and GLSL). You'll find it useful to have NVIDIA Cg Reference at hand while working on shaders in Unity.

Conclusion

In this chapter, we learned that GPU is a piece of hardware specifically designed to perform graphics-specific tasks. In particular, we know now that GPUs are extremely parallel and that provides a huge boost in processing, not only for graphics but for other types of isolated processing as well. We also learned that a Shader is a program that runs on the GPU, they're the tool we have to achieve awesome graphic effects without using the CPU. In this book, we'll focus on fragment shaders because they give us the power to define the final color of the pixel, an essential feature for 2D effects. Now it's time to start learning how to create a shader in Unity, and how to set up all the required objects to get it working.

Using shaders in Unity

This chapter will focus on providing you with enough context so that you're able to start your learning path. I'll guide you through the steps required to get a custom shader up and running inside Unity and we'll discuss the minimum boilerplate structure needed to start working on our custom shaders.

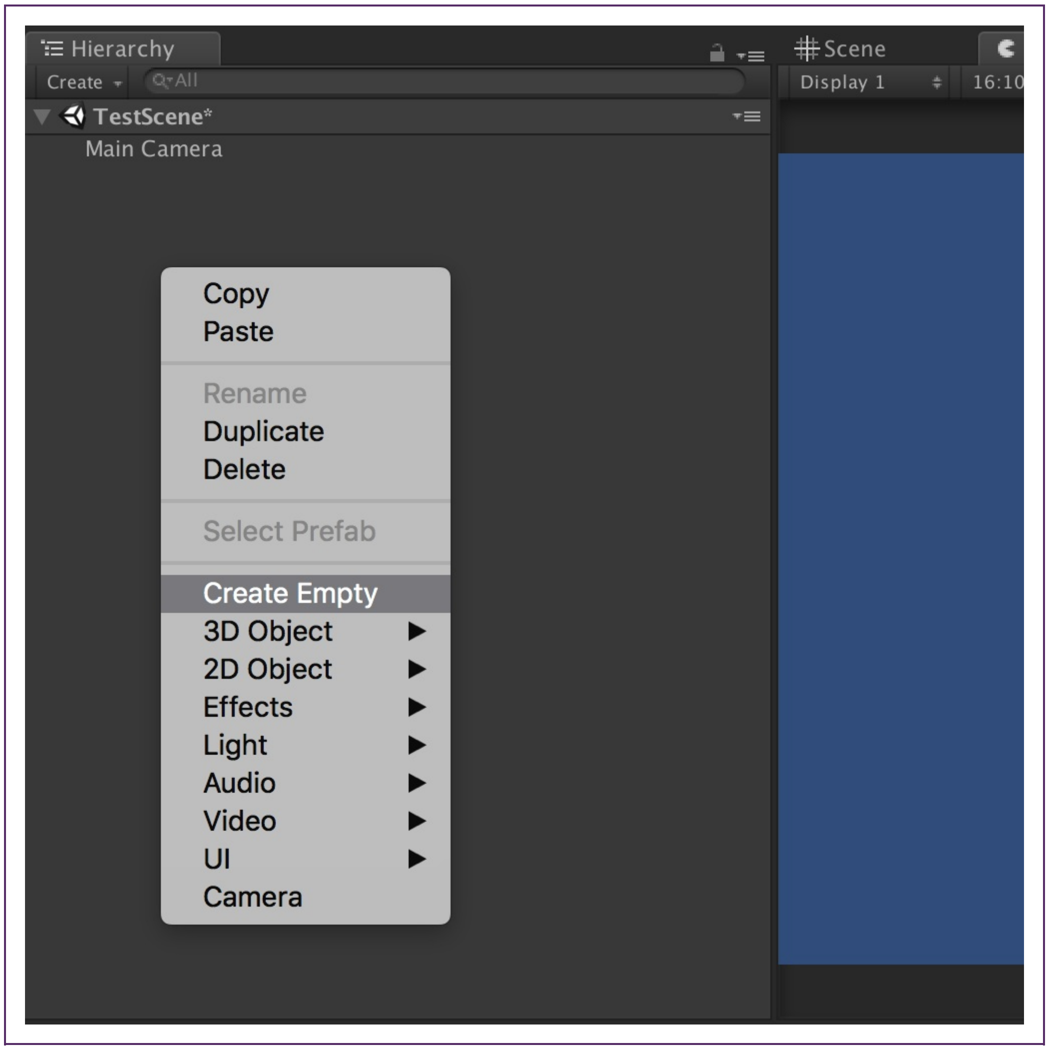
How to apply a shader in Unity

Before we can start playing around with shader code, we'll need to do some groundwork in Unity. We'll need to create an object with a renderer, a material, and a shader. We'll set those things up in this section so that you're ready to start learning.

Create the main GameObject

Unity comes with a shortcut for the steps we're going to use here, and that is using a Sprite object. That object is fine for most uses, but it has one issue that comes up when working with sprites with transparencies (specifically when we work with procedural texture manipulation), so instead of using Sprites during the book we'll create our GameObject from scratch, it's not that difficult and the results are pretty much the same.

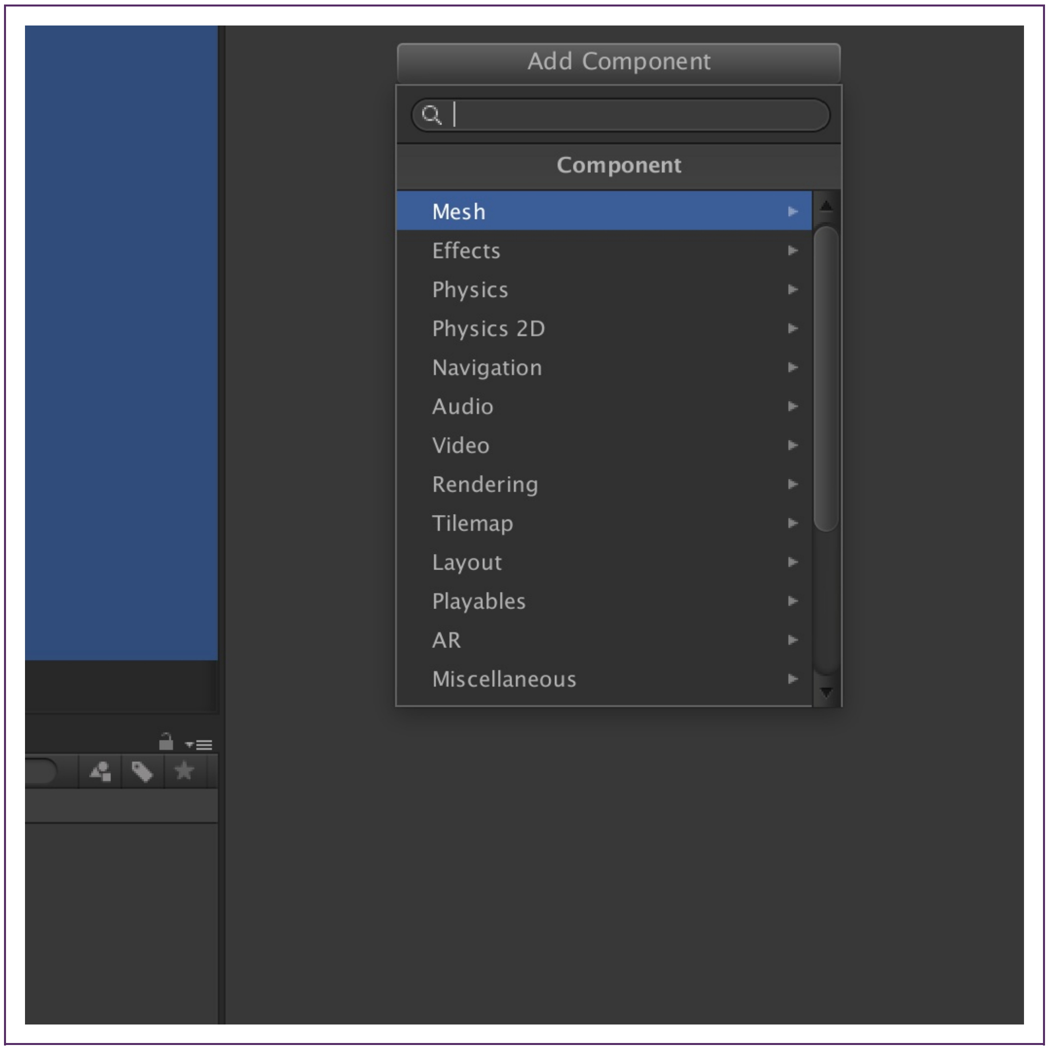
First of all, you'll want to create an empty GameObject by right-clicking the Hierarchy and clicking Create Empty. Call the object TestObject.



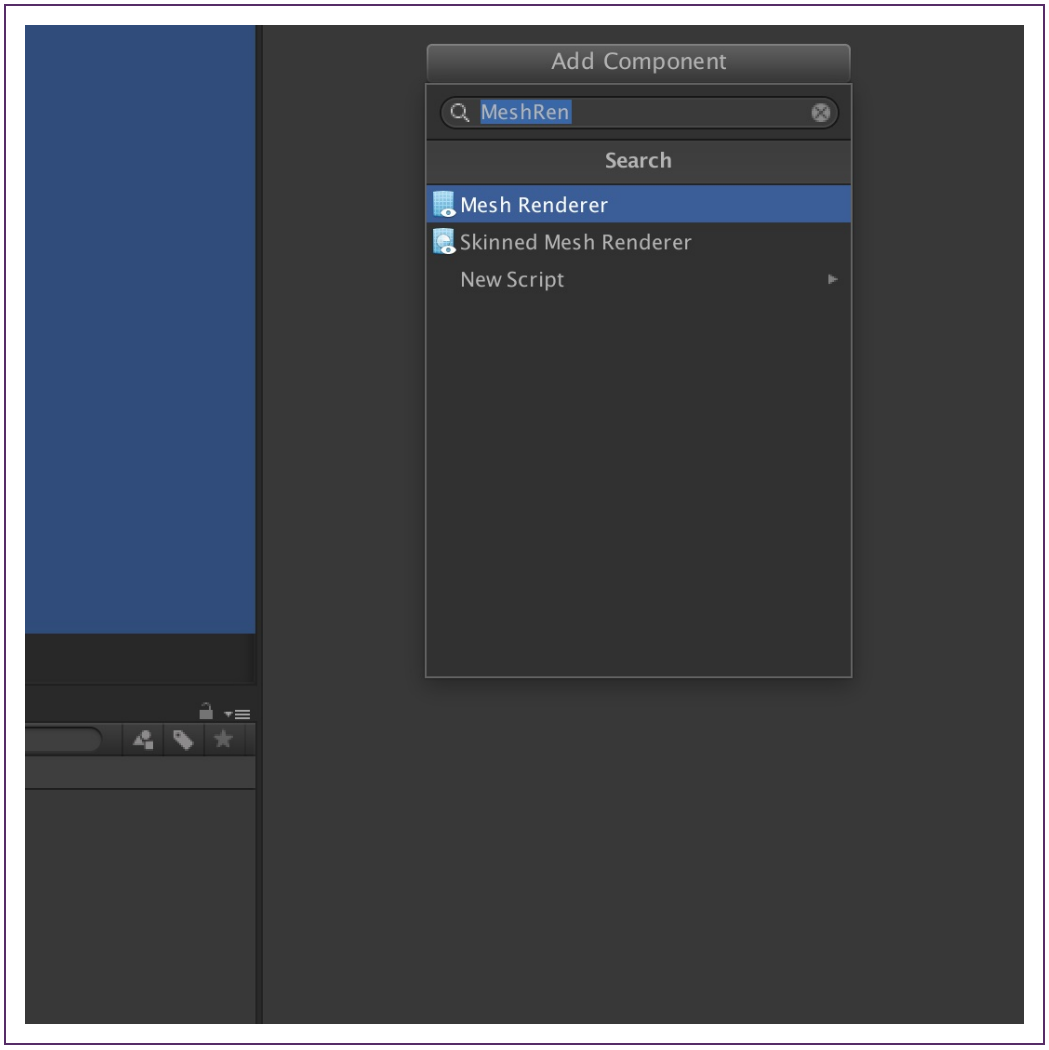
Add a renderer

We want to be able to render something to the screen. For this, we'll use a MeshRenderer and assign a Quad mesh. Some of you may be thinking why are we using a MeshRenderer instead of a SpriteRenderer (that automatically sets a quad mesh). That's a valid question. There is one case where the SpriteRenderer is going to be problematic, and that is when we want to deform a texture that has alpha (transparency) in it. If you're using a SpriteRenderer, Unity creates a mask with the original alpha position, and when you move the pixels inside the sprite it won't honor that movement, creating things that don't look good. That said this problem won't show up until we do procedural manipulation of textures, which is not in the current book, but I thought it would be good to provide with a configuration that Just Works™. So let's move on and get the MeshRenderer in place.

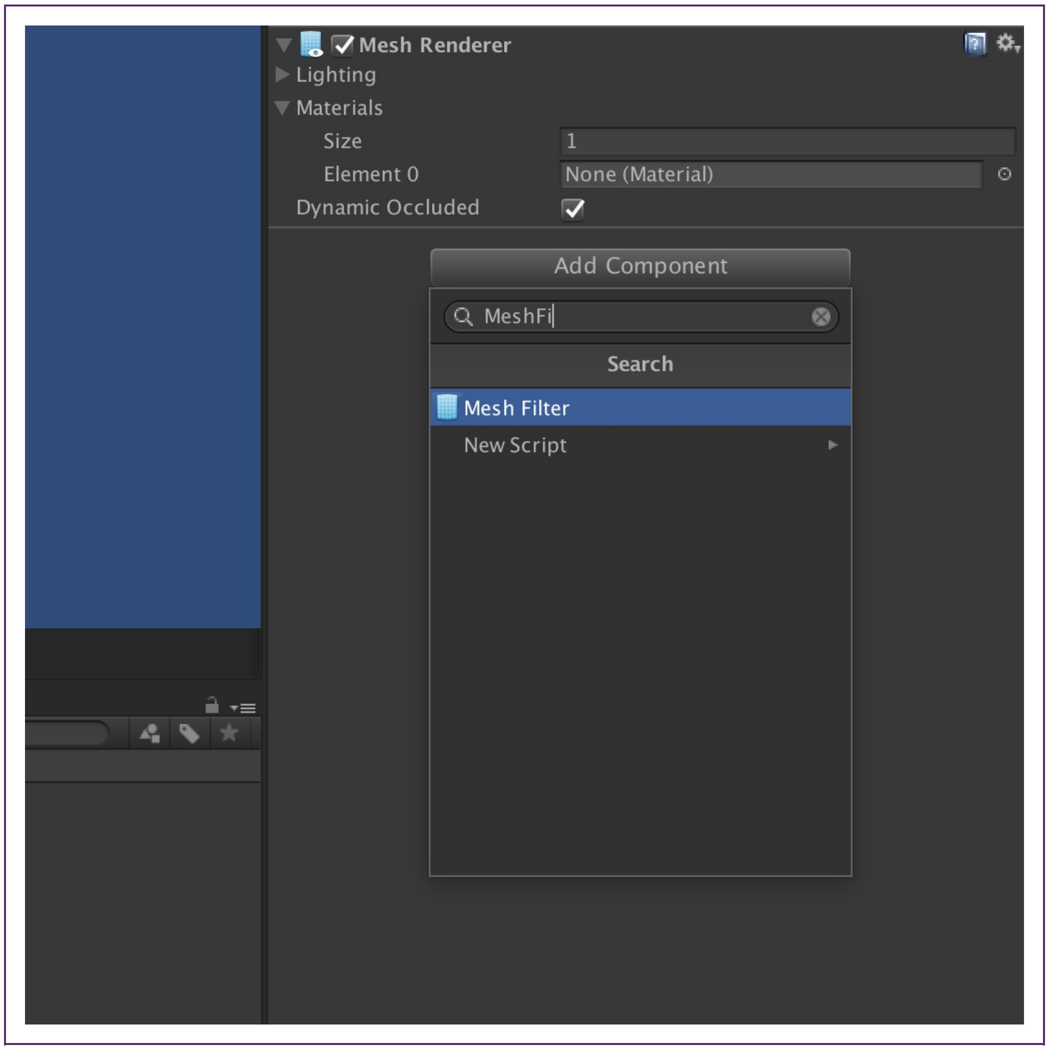
For that, we'll go to the Inspector and choose Add Component.



We'll write Mesh Renderer in the search box and hit Enter.

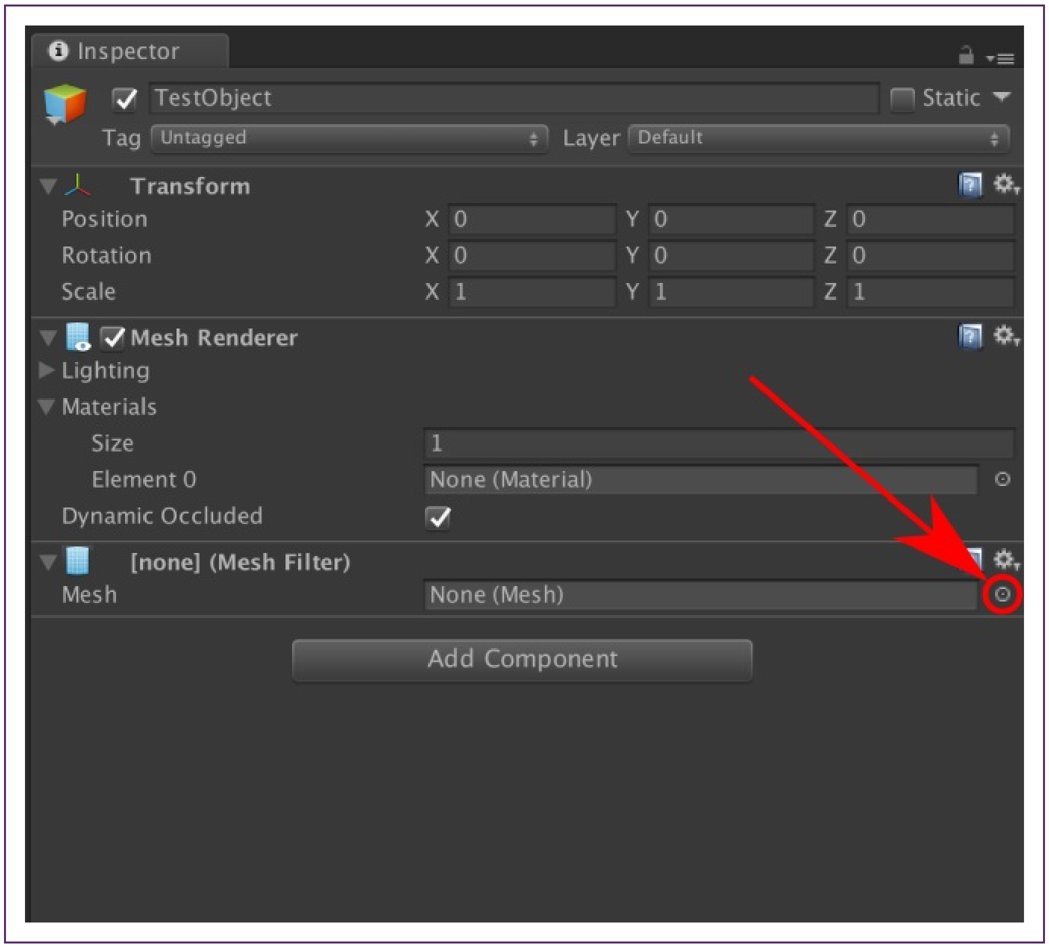


For the Mesh Renderer to work, we need also to add a Mesh Filter, to add it we repeat the process of clicking Add Component and search for Mesh Filter.

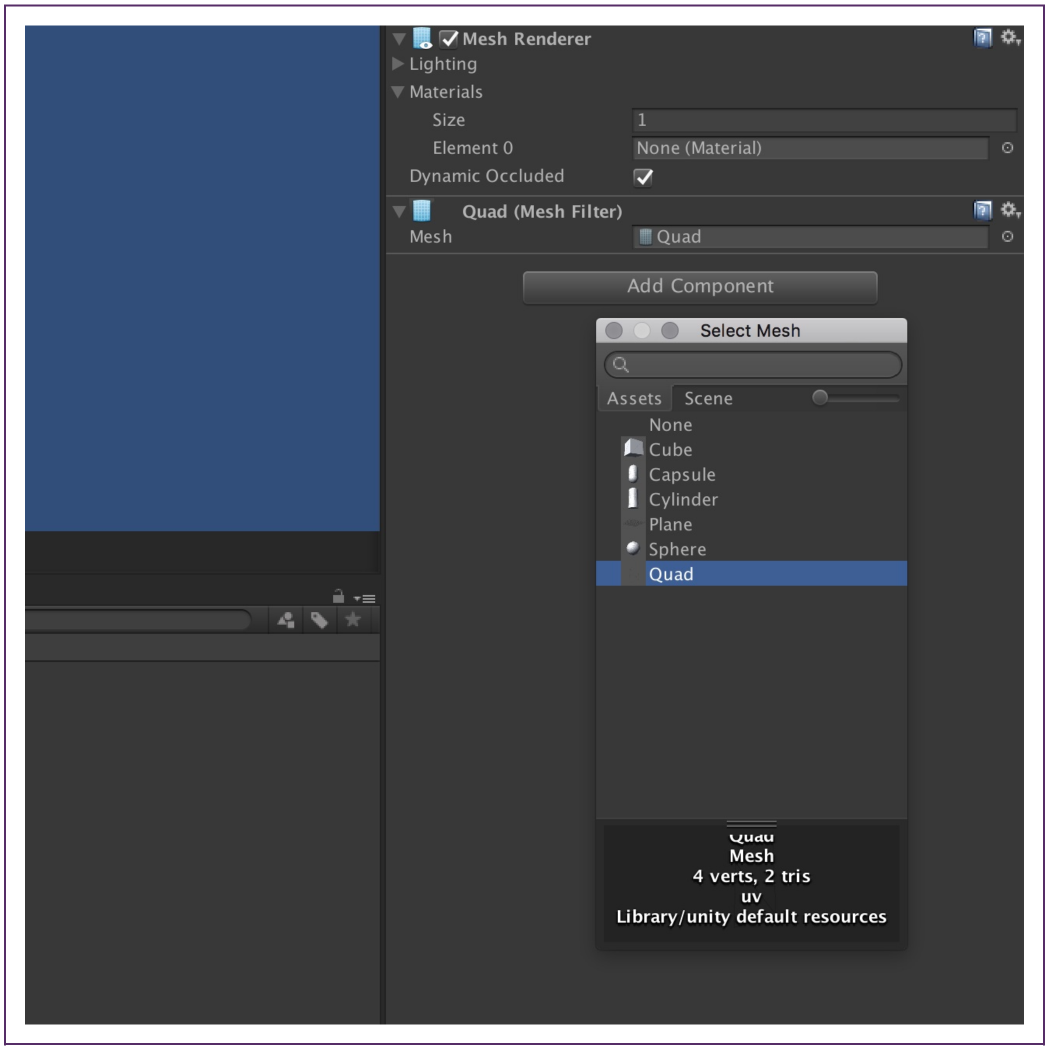


When working on 2D, we usually render textures inside of quadrangle meshes. This is not by any means a rule, you can work in 2D with meshes of any shape, but a quadrangle is the most basic figure that is useful for rendering images. This is what we usually call a Sprite. For specific deformation effects, you can use custom meshes. In fact, 2D animation suites, like Spine, allow you to define custom meshes and transform them at will. In our case though, we'll use plain old quads. We'll have to set the Mesh Filter to use a Quad, a built-in mesh that comes with Unity already.

For that, we'll click on the little circle by the Mesh field.



Then we select Quad from the mesh list. You'll notice that a magenta quad will show up on screen. That means we're on good track.


Mesh Renderer
Lighting
Materials
Size 1
Element 0 None (Material)
Dynamic Occluded ✓
Quad (Mesh Filter)
Mesh Quad

Select Mesh
Assets Scene
None
Cube
Capsule
Cylinder
Plane
Sphere
Quad

Quad
Mesh
4 verts, 2 tris
uv
Library/unity default resources

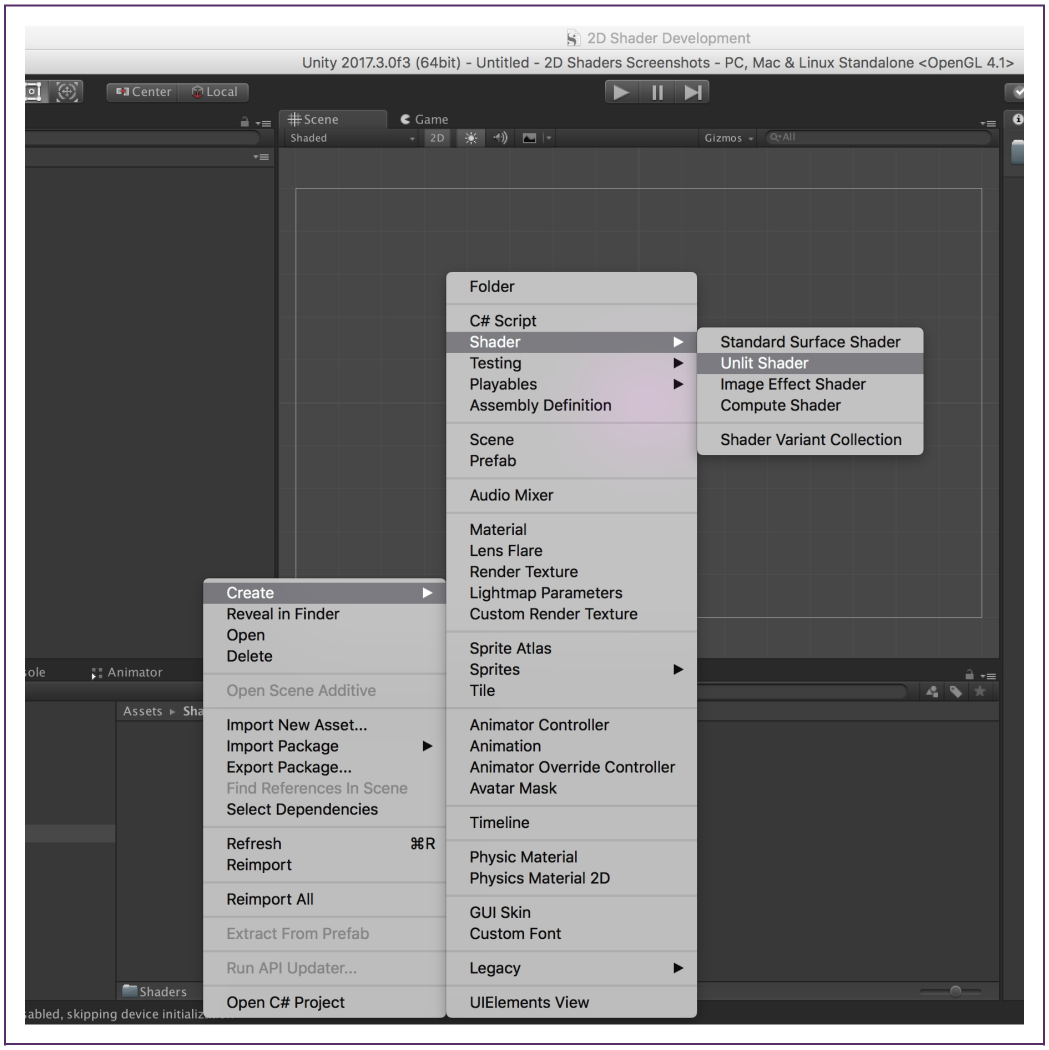
The reason we see it in magenta is that Unity uses that color to provide visual feedback that a renderer is not using any materials or that those Materials have Shaders that could not compile. Magenta is a color that is seldom used, at least in its pure form, and turns out to be a nice and evident way of telling us ERROR!!!

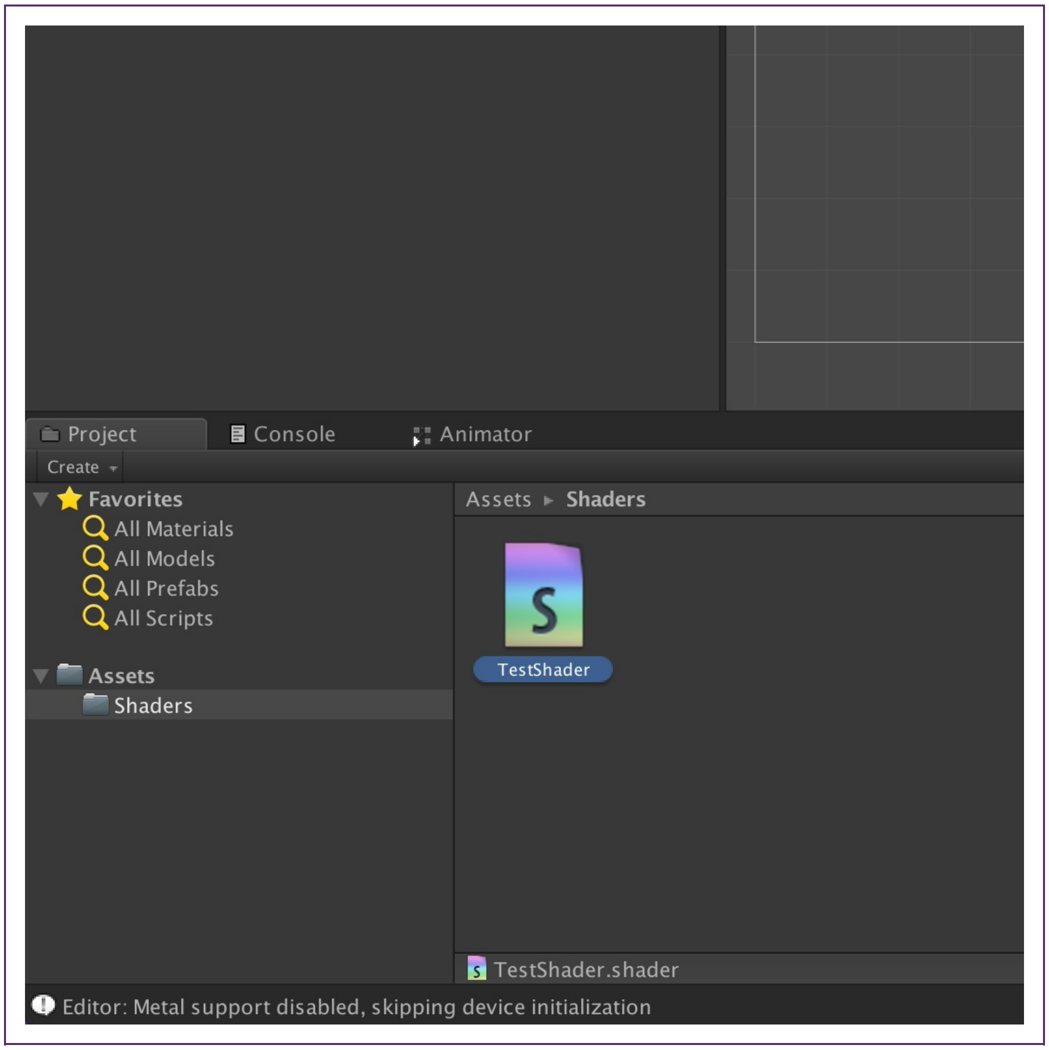
As I mentioned, the reason why we are seeing our quad in magenta is that we don't have any proper Material assigned to it. So let's do that now.

In Unity's context (and other engines as well), a Material is a special type of asset that combines a Shader and a set of

Settings for that shader in the form of parameters. Because of that, in order to create a Material, we first need a Shader.

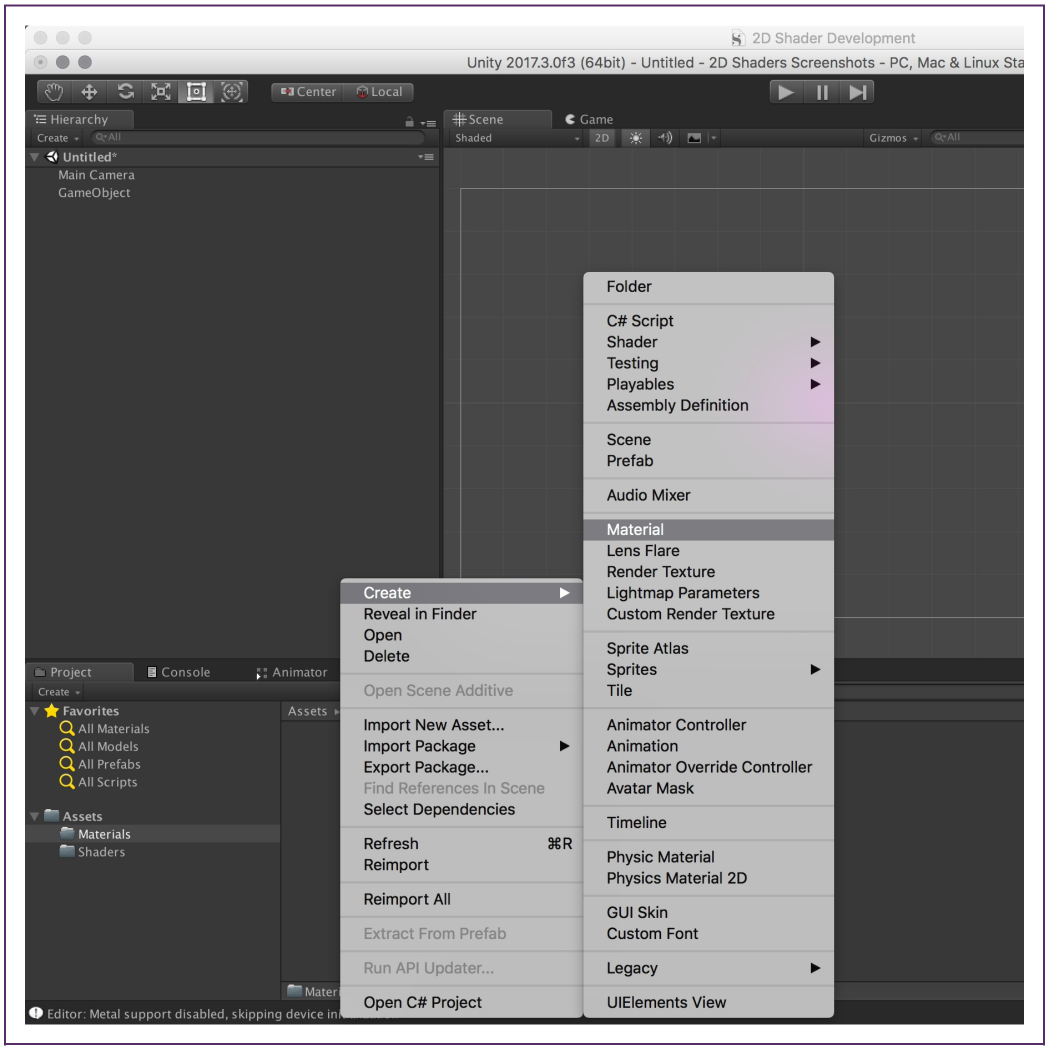
In the Project tab, go to the root of your project and create a folder called Shaders. Right-Click inside that folder and select Create > Shader > Unlit Shader. A new Shader asset will be created for you, give it a name, say TestShader, and hit Enter.



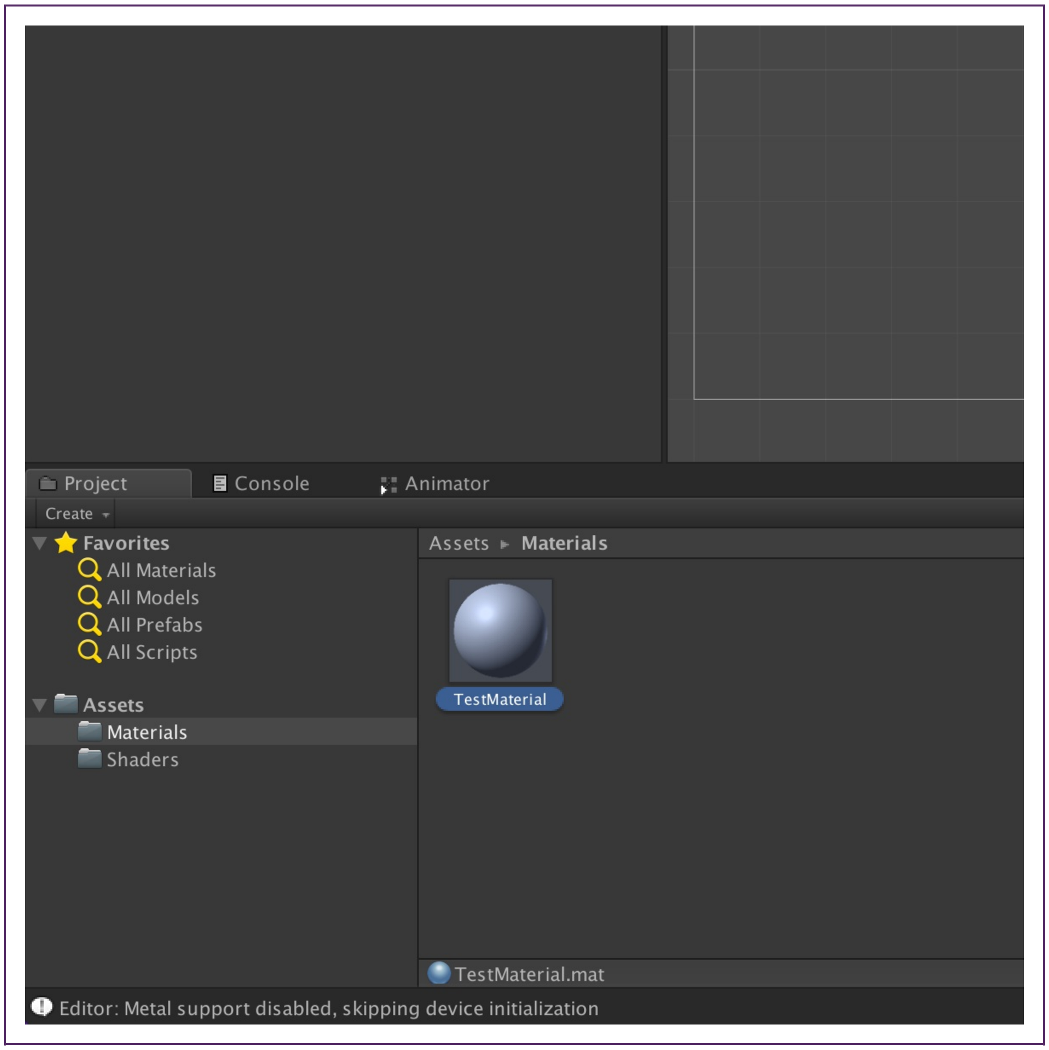


Editor: Metal support disabled, skipping device initialization

Now that we have our Shader (which we'll open and analyze later), let's go ahead and create a Material. For that, we'll create a new folder in the root of our project called Materials. Again, Right-Click inside the folder and choose Create > Material. Give it a name, like TestMaterial.

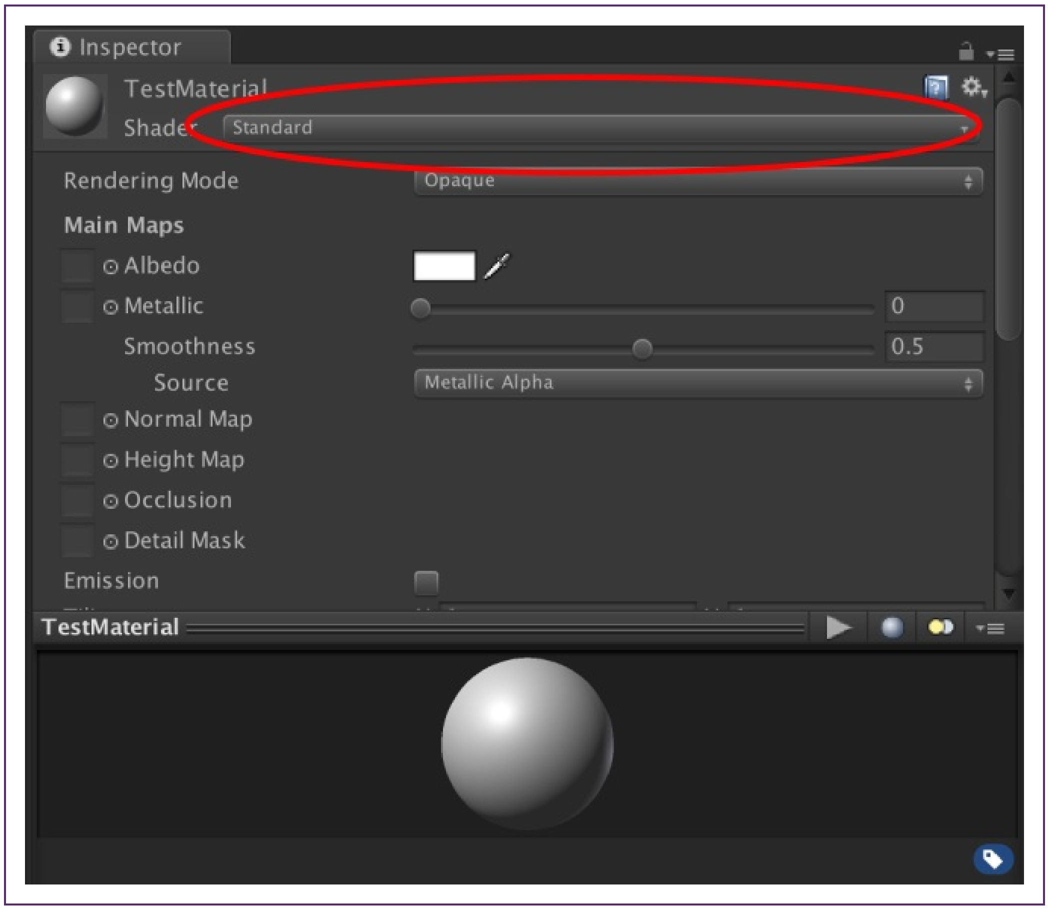


Editor: Metal support disabled, skipping device in

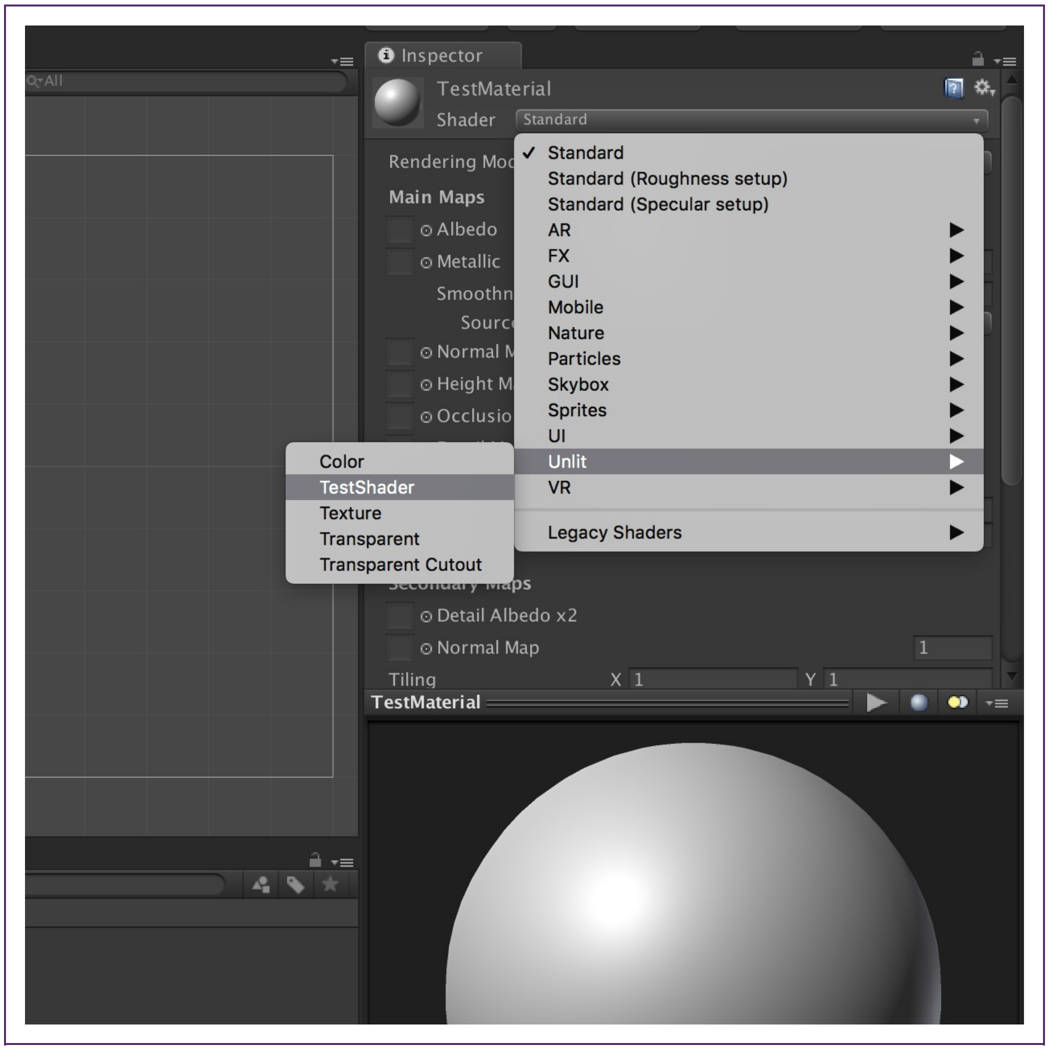


Editor: Metal support disabled, skipping device initialization

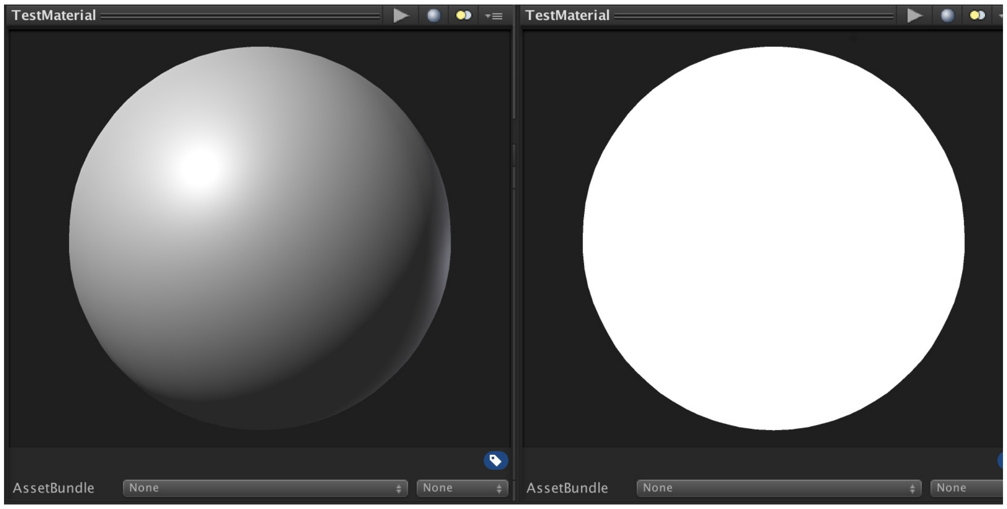
Now we need to assign the Shader we just created, to the Material. To achieve that click on the Material and in the inspector click the drop-box next to Shader as shown in the picture.



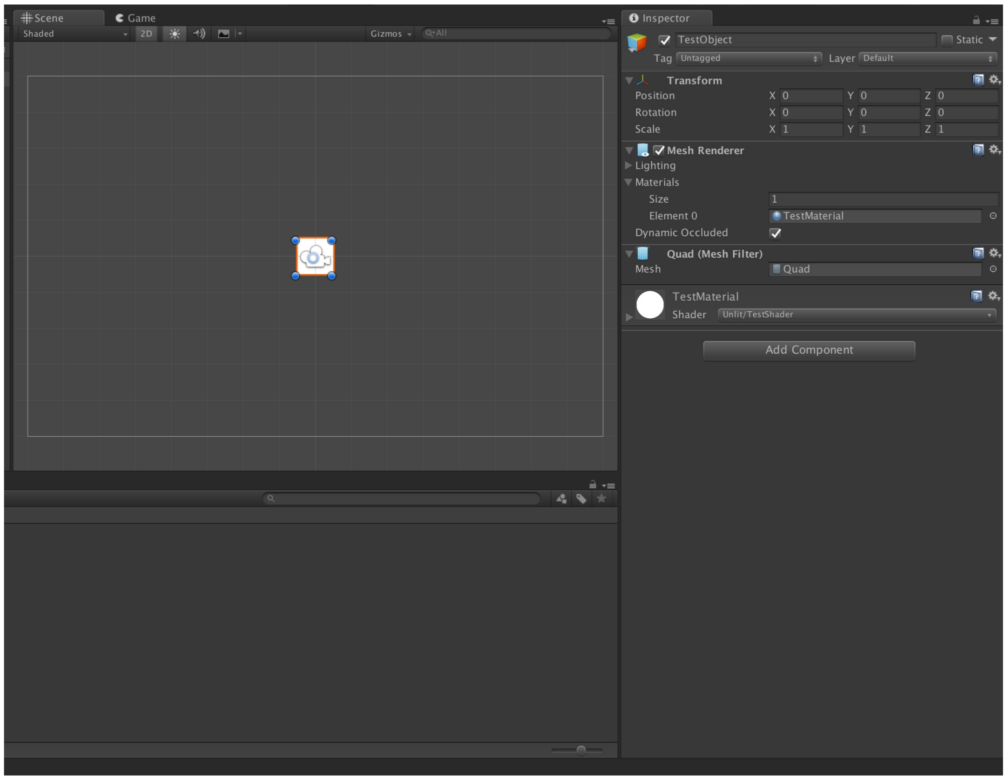
You should now see a big list with several shaders, we'll see how to organize our shaders inside this list in a few moments, but for now, go to Unlit > TestShader.



You'll see that the material will turn white. That's a good sign.



Now, click on TestObject in the Hierarchy. Then in the Inspector, look for the component called Mesh Renderer and open the Materials array. Put a size of 1 and drag and drop our TestMaterial in it.



TestObject
Tag Untagged Layer Default

Transform

Position	X 0	Y 0	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

Mesh Renderer
Lighting
Materials
Size 1
Element 0 TestMaterial
Dynamic Occluded ✓

Quad (Mesh Filter)
Mesh Quad

TestMaterial
Shader Unlit/TestShader

Now our Quad should turn white too. We have our GameObject ready. Now we are ready to start playing with our shaders.

Let’s take a look at our shader.

Now let’s open the shader in the editor by double-clicking the asset we created inside the Shaders folder.

When we created the Shader (by clicking Create > Shader > Unlit Shader), Unity added some default code to it. What you see is the basic code needed to show a texture on the screen. If you scan through it quickly you’ll see there are two functions, `vert` and `frag`. We’ll take a look at them soon, but just as a teaser, those are the vertex and fragment shaders. Before getting into that I wanted to explain how shaders are organized in Unity.

The very first line in the shader file reads:

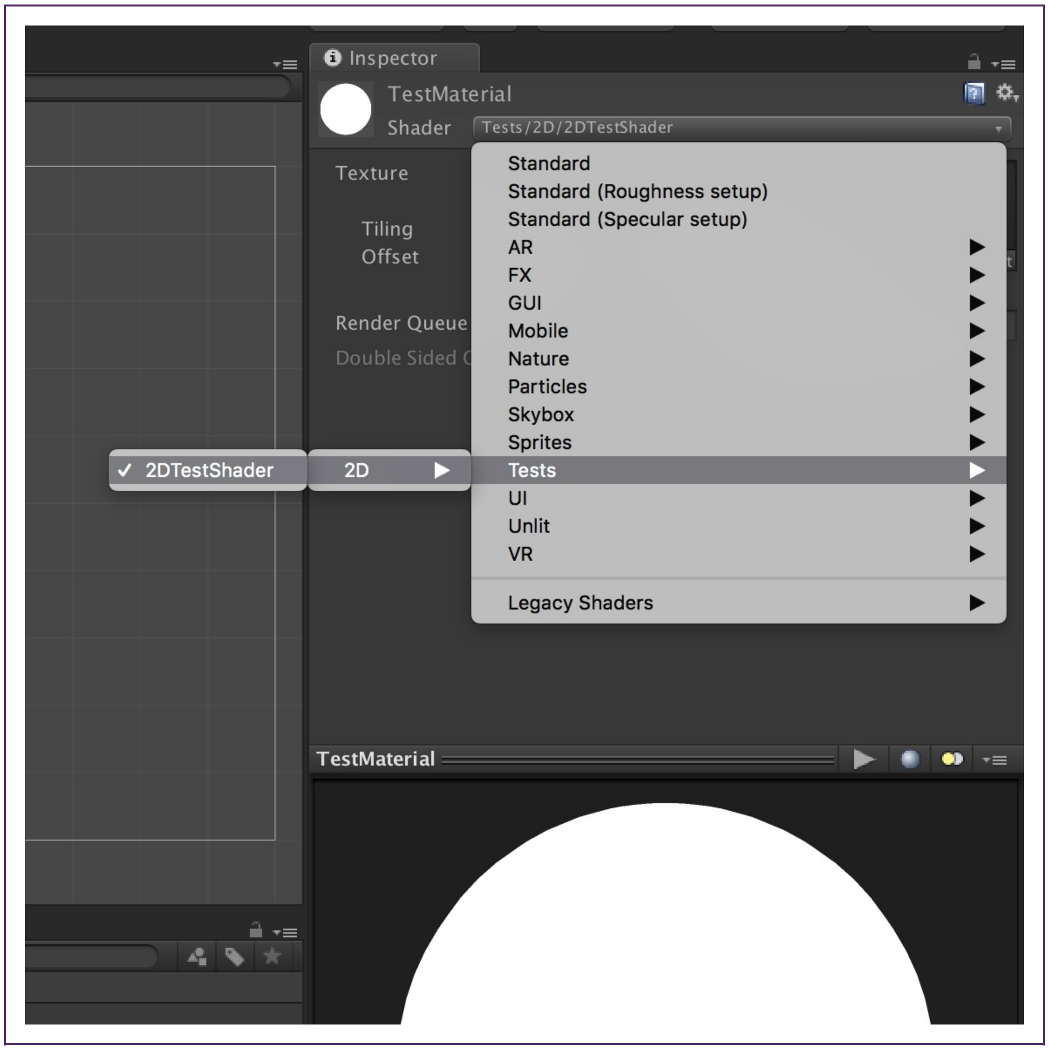
```
Shader "Unlit/TestShader"
```

Which means that this shader will be called TestShader and located in the Unlit folder. This folder and name are the ones you’ll see when you click the Shader dropdown in the material, as we did before.

Go ahead and change the shader name to something like:

```
Shader "Tests/2D/2DTestShader"
```

Go back to Unity and click the material’s Shader dropdown. You’ll see the Tests folder, click it too, and you’ll see a 2D folder and then the 2DTestShader. As you see, you can define the exact path using slashes, and name the folders between them.



Now that you know how to name shaders and organize them, we're going select all the shader code and delete it. Save the file and then go back to Unity.

Now you see the infamous magenta color. Remember this is because there are errors in our shader. In this case, the error is that the shader we were using no longer exists because we just removed it.

We removed everything because I wanted to start from scratch so you can get a general idea of what each section of the shader file does and why we need them.

The structure of a Shader in Unity

Being a generalistic engine, Unity must provide us with a lot of flexibility in terms of how we render our game. One of the features we need is the ability to modify the rendering state machine to a certain degree and that has to be multi-platform.

To solve this issue, Unity created a 2-layer structure to handle our shaders.

The first layer is used to handle a set of configurations required by the rendering engine to perform several tasks. We do that using a language created by Unity and called ShaderLab.

As you'll see, we'll have to write some ShaderLab code in order to define our vertex and fragment shaders in the next section, but let's now take a look at the basic ShaderLab structure.

```
Shader "Folder/ShaderName"
{
  Properties
  {
    // …
  }

  SubShader
  {
    Pass
    {
      CGPROGRAM
      // …
      ENDCG
    }
  }
}
```

The first thing you'll write was already discussed in the previous lesson.

The `Shader` Command

The `Shader` command tells Unity that all the code inside brackets defines a shader. We also need to name the shader, and if you recall we define a path in the shader Dropdown of the Material using slashes. For example:

```
Shader "Test/2D/2DTestShader"
{
// …
}
```

Inside the brackets, we'll define several sections that are used to use certain features of the rendering engine.

The `Properties` Command

One of the most used are Properties: a set of variables that get exposed to Unity's material editor, and that you can use to tune your shader without touching the shader code itself.

Those are defined using the `Properties` command, and each property will have a line defining its name, type and default value.

```
Shader "Test/2D/2DTestShader"
{
  Properties
  {
    _MainTex ("Texture", 2D) = "white" {}
  }
}
```

In this example, we're defining a property that will allow us to access a 2D texture from our shader. We're referencing that texture with the name `_MainTex`, and if it's not set through the material configuration or by code, it will be a white image.

The `SubShader` Command

To be able to write shaders for different platforms or GPUs with different capacities, Unity created a section called SubShader. What Unity does, is it picks the first SubShader that will run on the target GPU. In the book we won't dive into platform-specific nightmares, so we won't be dealing with SubShaders, but is worth noticing that they exist.

```
Shader "Folder/ShaderName"
{
  //…
  SubShader
  {
  // You shader code here…
  }
}
```

The `Pass` Command

You may now think I'm playing with you, but we're getting there. Inside the SubShader, we can define what is called Passes.

When using a specific SubShader, Unity renders the object once per Pass, this could be used to, for example, grab the rendered screen in the back of the current sprite and do some processing while blending with the current object. This is useful to create things like a water diffraction effect.

```
Shader "Folder/ShaderName"
{
  //…
  SubShader
  {
    Pass
    {
      // You pass code here…
    }
  }
}
```

Again, we're not going to be messing with Passes in this book, we'll usually have just one SubShader and one Pass.

`CGPROGRAM` and `ENDCG`

Inside the Pass block, we need to tell Unity that we want to Cg to write our shader code. This is done using the `CGPROGRAM` and `ENDCG` tags inside each Pass.

```
Shader "Folder/ShaderName"
{
  //…
  SubShader
  {
    Pass
    {
      CGPROGRAM
      // . . . Cg code for each shader goes here
      ENDCG
    }
  }
}
```

We do this because we're going to use Cg to make our code multi-platform. If you were targeting Linux, Android or some other OpenGL-only target, you could use GLSL directly. To do this you have to write the `GLSLPROGRAM` and `ENDGLSL` tags instead.

Same goes for HLSL.

Again, we'll be only using Cg in the book since it will compile into GLSL or HLSL (or the console equivalent) depending on the platform we're exporting to.

Is that it?

Mostly, this is the base structure you need to create a shader, but it is not functional yet, if you copy and paste the previous code you'll get a compiler error. We still need to implement the actual code for the shader (what goes between `CGPROGRAM` and `ENDCG`) and that is what we'll do in the next chapter.

Unity's built-in shaders

Unity has a bunch of built-in shaders that you can select and use. You can check them all by clicking the Shader dropdown in the material inspector. The bad news for newcomers is that you can't modify those, you can't even see the source code, so you don't know what's going on behind the scenes.

The good news is that the code of those shaders is available to download in Unity's web page.

To get them:

- Head to http://unity3d.com/get-unity/download/archive
- Search for your Unity's version
- Pick your platform and click Downloads ([Platform])
- Click Builtin shaders

You'll get a zip file with all the code for the Builtin shaders. I strongly recommend that after finishing this course you go through all of them and try to understand what they all do. That's a good way to learn new shader types and understand how shaders work in Unity, not only in 2D but in 3D as well. The Image Effects pack that comes with Unity is another good source of learning material.

Conclusion

In this chapter, we learned how to create a Shader script in Unity, how to apply it to a material and finally to a sprite.

We analyzed the basic structure of a Shader written using ShaderLab and created the structure for a shader that does nothing.

We also found out that Unity shares its built-in shader code in their webpage.

Now that we finished with the boilerplate that Unity requires to create a shader, we can actually start writing our first useful shaders. In the next chapter, we'll learn the difference between Vertex and Fragment shaders, and will code a few shaders from scratch.

Fragment Shader basics

This is the first chapter with information that will be useful for the rest of our shader programming career. We'll write two key shaders, one that is the most basic shader possible, a Solid Color shader, that will allow us to draw a colored mesh on screen. For that, we'll learn about RGBA and how we can modify colors. Then, we'll write another shader that maps a texture into our mesh, successfully rendering a sprite. For that, we'll also need to learn how UV Mapping works.

What's the difference between Vertex and Fragment shaders?

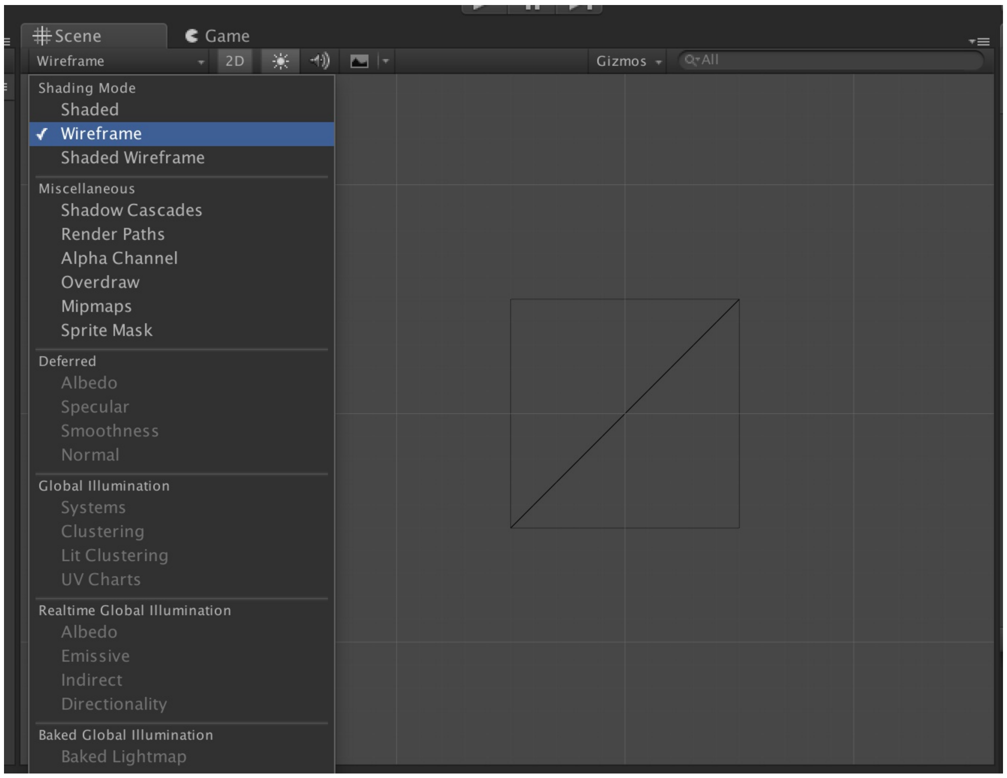
To display something on a screen, if you're using a modern graphics API, you'll need at least a vertex shader and a fragment shader.

I want to keep things simple in this book, but I don't want to omit important information. There is a concept in most modern graphics APIs called Graphics Pipeline. The Graphics Pipeline is a series of steps that are performed one after the other to render an image. That's all I will say about it because there is a lot of information on the internet and other books on this topic. Learning about it is the right way to get the full picture of how the data gets transformed from mathematical models into the final pixels.

For now, let's focus on the data transformations that happen between vertex and pixel shaders.

The first thing you need to know is that when we're working on 2D games, we tend to use quad meshes to render our sprites.

A quad is created using 4 vertices, and two triangles. You can easily debug that using the wireframe view mode in the Scene tab as I'm showing.



When it's time to render the screen, those four vertices are sent to the GPU and get processed by the Vertex Shader.

In this shader, you have to take care of moving the vertices from model space into world space (with a simple instruction that we'll see soon) and set several options that will be linearly interpolated between connected vertices during the Rasterization stage. Linear Interpolation is an algorithm that is widely used by programmers so I'm not explaining it here. If you are not familiar with it, please take a look at the Linear Interpolation Appendix of the book.

After running the Vertex shader, we get into the Rasterization stage. In this stage our geometry from mathematical models, that is our vertices and triangles, get transformed into fragments (potential pixels). During Rasterization all the values that are set in the vertices (colors, normals, texture positions, etc) are linearly interpolated.

After rasterization, we get into the Fragment Shader stage, where we run our fragment shader on every single fragment that was created during Rasterization.

During this stage, we’ll read the texel (the corresponding pixel from the texture) that should go in this screen pixel. We can also modify the logic by which this texel is retrieved, or even do some processing like slightly colorize it to achieve illumination or visual effects. We can do a lot of things when we start thinking of ways of modifying the color of each pixel.

After the Fragment Shader runs we’ll return a final color for our pixel, using different algorithms according to what we want to do.

After running all of our fragment shaders, we end up with the buffer of colors that will go into our monitor in this frame.

This process of running the Vertex Shader, then Rasterization and Fragment Shader is done for each visible geometry in the scene and will result in our rendered frame.

I hope you see why fragment shaders are so important for us now. For 2D games, vertex shaders can do a set of effects where you can modify the shape of the quad, but that’s not as useful as being able to control the color of each pixel. When working on Fragment shaders, we have control of the final color of the pixel, and there are several techniques that achieve really interesting effects by leveraging this feature.

Solid Color: Writing our very first shader from Scratch

Now that you have enough background, we can start programming our first shader. It will be a simple one. We'll only render a solid color on our quad mesh.

Let's start by creating a new empty Shader and call it SolidColor, as we saw in the previous chapter. We'll remove all the default code and write the boilerplate code seen in the previous chapter too:

```
Shader "2D Shaders/SolidColor"
{
  Properties
  {
    // …
  }

  SubShader
  {
    Pass
    {
    CGPROGRAM
    // …
    ENDCG
    }
  }
}
```

Defining the vertex and fragment methods

The first thing we need is to tell Unity the names of the methods that will be used as Vertex and Fragment shaders. This is achieved by setting pragma comments for the preprocessor.

After the `CGPROGRAM` and before the `ENDCG` commands, we'll add the following:

```
CGPROGRAM
#pragma vertex vert
#pragma fragment frag
// …
ENDCG
```

The `#pragma` directive is used to let the rendering engine know which functions we want to use as vertex and fragment shader. In this case vert and frag, respectively. This is a standard name for vertex and fragment functions, you can call them whatever you want though, as long as you define the right `#pragma` directive.

Some required data structures

Our vertex function receives data from the application in a custom-built struct written by us. For now, we only want to pass the vertex position, so let's add a `float4` called position.

```
#pragma fragment frag

struct appdata {
  float4 position : POSITION;
};
```

When using Cg or HLSL, we need to tell Unity what we want to do with each member of the struct, for that, we use shader semantics (For a complete list of the Vertex Input and Fragment Output semantics check https://docs.unity3d.com/Manual/SL-ShaderSemantics.html).

In this case, the `float4` position needs to be bound to the `POSITION` semantic. That's done using a semicolon and then the semantic itself.

The appdata struct, then, connects our application with the vertex shader. Now we need to create another struct that is used as a return value for the vertex shader, and it's going to be used by Unity to define where that vertex goes in the final clip coordinates, and also to pass the fragment shaders interpolated data. We'll learn more about that data later in the book when we talk about texturing our mesh.

We'll call that struct vertex to fragment or `v2f`, and will include a `float4` position with the `SV_POSITION` semantic. This semantic is required by Unity and we'll have to fill it with the clip space position of the vertex. Don't run in desperation if you don't know what this is, Unity provides a built-in matrix that does this for us.

```
struct v2f {
  float4 position : SV_POSITION;
};
```

Now that we have our appdata and `v2f` structs, let's write the vertex shader.

The vertex shader method

```
struct v2f {
  float4 position : SV_POSITION;
};

v2f vert ( appdata v ) {
  v2f o;
  o.position = UnityObjectToClipPos(v.position);
  return o;
}
```

First of all, we define the output value of the method as `v2f`, then we define the name of the method to be `vert`, and finally we pass an appdata struct to it. Inside the method, we create a `v2f` instance.

Then, we set the position using Unity's helper `UnityObjectToClipPos`. This transforms our vertex from model space into clip space. Internally it multiplies the vertex by the MVP matrix. That's the only requirement Unity (or graphics APIs to be more correct) has for the vertex shader, so we can safely return our `v2f` instance and we're done.

Don't worry if you didn't get the full picture of what's going on here, understanding that matrix multiplication is not completely needed for us to produce nice effects, although I recommend you go through some 3D math basics to understand what the MVP matrix is and how it works. Check the Where to go now? Chapter to find references.

MVP stands for Model View Projection, and is a matrix that contains three concatenated affine transformations, one that moves the vertex from model space to world space, then view space and finally projects it into a plane, called clip space. But that's all I'll say about this since it goes out of the scope of the book.

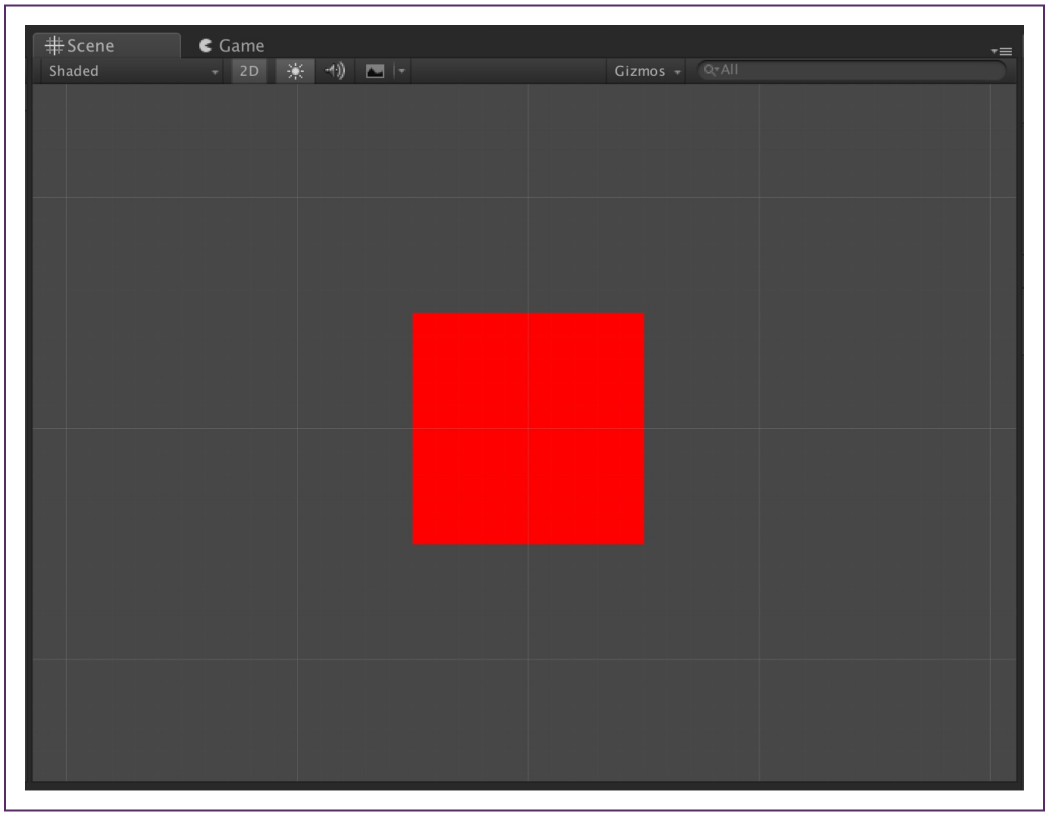
The fragment shader method

```
fixed4 frag (v2f i) : SV_Target {
   return fixed4(1,0,0,1);
}
```

Our frag method receives a `v2f` struct, and returns a color in the way of a `fixed4`.

In order for it to work (and this is really important) we need to tell Unity that the result of this method is a color, we do that using semantics again. This time, we use a colon and the `SV_Target` semantic. As with the Vertex Input semantics, you can check Unity's documentation to find out more about Fragment Output semantics.

Now to finish this, let's make the frag method return a fixed color. For that, we create the `fixed4` value by hand. Using `(1, 0, 0, 1)` as the parameters we should get a solid red quad in Unity. If you go to Unity you should see a red square on the screen.



Yay! It works. If it didn't work for you, please go to the book's forums and post your errors, there always be someone there to help you.

If you know what RGBA color space is, that's exactly the format that this function is returning.

For those of you who are not familiar with RGBA, the next section will explain how it works.

The full shader code is as follows:

```
Shader "2D Shaders/SolidColor" {
    SubShader {
        Pass {
            CGPROGRAM
            #include "UnityCG.cginc"

            #pragma vertex vert
            #pragma fragment frag

            struct appdata {
                float4 position : POSITION;
            };

            struct v2f {
                float4 position : SV_POSITION;
            };

            v2f vert (appdata v) {
                v2f o;
                o.position = UnityObjectToClipPos(v.position);
                return o;
            }

            fixed4 frag (v2f i) : SV_Target {
                return fixed4(1,0,0,1);
            }

            ENDCG
        }
    }
}
```

Understanding RGBA

Before we continue working on shader code let's stop to talk about color representation in computer graphics. As we saw in the `SolidColor` shader, we represent a color with a vector of four normalized floating point numbers.

Each vector component, that is `x`, `y`, `z` and `w`, are bound to colors, Red, Green, Blue, and Alpha, or transparency. Since it's a normalized vector, each value goes from `0` to `1`. Having these components normalized makes color math easier since we can combine them by multiplying them, or even modulating with other functions and always stay in range.

```
float4 red = float4(1,0,0,1);
```

In this example, we are defining a `float4` variable called red and initializing it to `(1,0,0,1)`. The reason why this represents the color red is that the value for the red channel (the first component) of the vector is one, while the green and blue channels are zero. The alpha channel (the fourth component of the vector) determines the opacity, which is not so useful now since we haven't talked about Alpha Blending. A value of one in the alpha channel means the color is fully opaque, with no transparency applied. In the future, we'll be able to use this channel to give transparency to our pixel and blend it with existing colors from previously rendered objects.

As an aid to programmers, we can retrieve vector components not only by using `x`, `y`, `z` and `w`, but also `r`, `g`, `b`, `a`. In that way is much easier to read the functions we write to process colors.

```
float4 red = float4(1,0,0,1);
red.x; // returns 1

red.r; // returns 1 too.

red.g; // returns 0
red.b; // returns 0
red.a; // returns 1
```

Let's see some examples:

```
float4 red = float4(1,0,0,1);
float4 green = float4(0,1,0,1);
float4 blue = float4(0,0,1,1);
float4 white = float4(1,1,1,1);
float4 black = float4(0,0,0,1);
float4 magenta = float4(1,0,1,1);
float4 yellow = float4(1,1,0,1);
float4 cyan = float4(0,1,1,1);
```

Those are the colors you get by using channels to their full capacity, but if we use numbers between zero and one, we can represent way more colors, let's take a look.

For example, if you want to make a gray that is halfway between black and white.

```
float4 mid_gray = float4(0.5,0.5,0.5,1);
```

Then, for example, you can tone that mid gray to get a reddish mid-gray increasing the red channel in `0.1`.

```
float4 redish_mid_gray = float4(0.6,0.5,0.5,1);
```

I suggest you explore the RGBA space modifying the `SolidColor` shader we created in the previous section. In this way, you'll get a better picture of what's going on.

Playing with colors

The `lerp` function

Now that we have a better understanding of how colors work in the context of computer graphics, let's introduce some useful functions we can use to play with them.

First of all, we want to talk about the lerp function. This function allows you to do linear interpolation between two values. That is mixing two values using a weight variable. You control the amount of each value that goes into the final mix by setting the weight to a number between `0` and `1`. Let’s test this out in our `SolidColor` shader.

Inside the fragment shader we’re going to define two variables: `col1` and `col2`.

```
float4 col1 = float4(1,0,0,1); // Red
float4 col2 = float4(0,1,0,1); // Green
```

Then, we want to replace the return line from:

```
return fixed4(1,0,0,1);
```

to:

```
return lerp(col1, col2, 0.0);
```

Now what we see is… still red. That’s because our first color is red, and the weight variable is set to `0.0`, so the result is full red. If we change the weight variable to `1`, we get full green.

But if we put `0.5`, or `0.25`, we get a color that is the weighted average of those two colors.

If you’re working with OpenGL, in GLSL this method is called `mix()`.

The `_Time` struct

This is our first approach to being able to create colors by code, so let’s introduce a concept that will be really useful in the future. Unity provides some built-in variables that you can access to in the shaders, a really important one is the `_Time` struct. It is a `float4` struct, that holds several precomputed time scales inside. If t is the current time since we started the game:

- The `x` component contains `t` divided by `20`.
- The `y` component contains `t`.
- The `z` component contains `t` multiplied by to `2`.
- And the `w` component contains `t` multiplied by `3`.

These values can be used to create animations inside our shaders.

Let’s modify our weight value in the shader to use `_Time.y` instead of a fixed number. To visualize this change, we need to run our project. Go ahead and hit play.

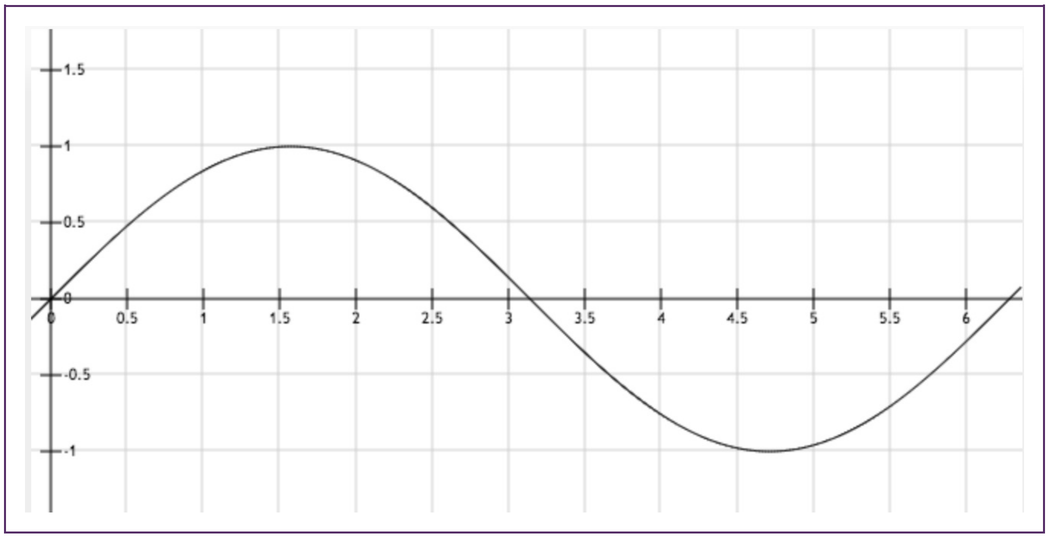
```
return lerp(col1, col2, _Time.y);
```

That was fast. What is happening is that it takes one second to go from red to green, since at the beginning `_Time.y` is `0`, and goes up frame by frame, until the first second, when `_Time.y` becomes `1`. After that `_Time.y` is greater than one, so the linear interpolation returns the last value (green).

We can make this animation twenty times slower by changing `_Time.y` to `_Time.x`. When we click play we get 20 seconds of slowly changing from red to green because `_Time.x = t / 20`.

The `sin` function

What if we want to make this animation bounce forever so that it slowly comes back to red after hitting green? If you remember your trigonometry classes, you'll know that the sine function transitions softly and infinitely between minus one and one.



We can rely on it to add perpetual animations to our shaders.

```
return lerp(col1, col2, sin(_Time.y));
```

Nice! It's animated, but… it stays in red for a long time, can you identify why? The reason is that since `sin` goes from minus one to one, the negative part of it is always showing red because the linear interpolation shows intermediate values only when the parameter goes from zero to one. In this case, we're below zero, so the linear interpolation returns red always.

In order to make this work as we may expect, we'll have to normalize this sin and make it go from `0` to `1`.

You do this by applying some simple math let’s see, we know sine goes from minus one to one.

```
-1 <= sin(x) <= 1
```

Then, if we add one to it, it will go from zero to two.

```
0 <= sin(x) + 1 <= 2
```

Now, if we divide this by 2 (which won’t invert the signs since two is positive), we end with our sine going from 0 to 1. That’s what we wanted.

```
0 <= (sin(x) + 1) / 2 <= 1
```

When we apply this math to our shader, we get a nice smooth transition between red and green.

```
return lerp(col1, col2, (sin(_Time.y) + 1) / 2 );
```

For this specific case, Unity also provides `_SinTime` and `_CosTime` built-in structs, that contain precomputed sines and cosines of `t`. You can read more about Unity’s built-in variables in the documentation: http://docs.unity3d.com/462/Documentation/Manual/SL-BuiltinValues.html

You’ll learn extensively about the sine function in the Procedural Texture Manipulation book.

UV Mapping

The shader we'll write in the next section will display a Texture in our mesh. This is the first step required to show a 2D sprite. But before we can do that, we need to discuss one more thing: UV Mapping.

UVs are two-component vectors with a special semantic: they represent the 2D coordinates of a texture.

Think of a squared image as a rectangle in a coordinate system, the `x` axis (U) represents the horizontal position, and the `y` axis (V) the vertical position.

In order to be useful to operate on them, UV vectors are also normalized, so their components go between `0` and `1`. The vector `(0,0)` represents the bottom-left pixel of the texture. The vector `(1,1)` is the top-right pixel. `(0.5, 0,5)` is the pixel in the center of the texture.

This is really handy because we don't have to know the resolution of the texture in order to access it, instead, we can calculate the specific pixel we want to retrieve by using values between `0` and `1`.

The technique is called UV Mapping because we use UV coordinates to map a texture into a mesh. In order to do that, we define a UV value for each vertex of the mesh, that value is then passed to the vertex shader, and finally to the rasterizer to get it interpolated and tell the fragment shader which pixel in the texture corresponds to a given fragment.

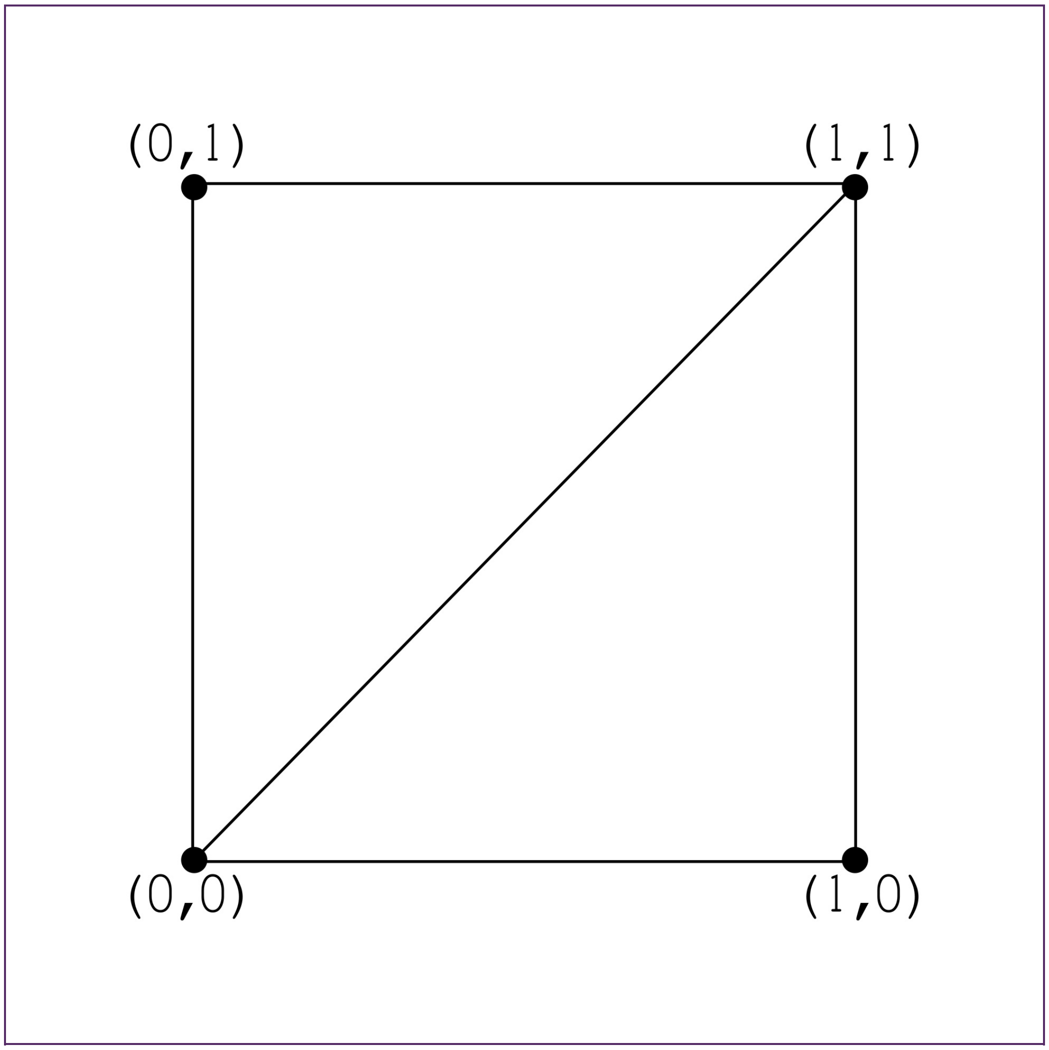
You will also find that books or articles refer to pixels in a texture as texels, which is more correct since we're not talking about screen pixels, but something stored in a texture that in the future may become a pixel.

Quad UV mapping

As we just saw, every mesh you want to project a texture into needs UV coordinates to be set in the vertices. This is a standard practice across all the different graphics APIs, so our Quad mesh that Unity provides already has UV values assigned to its vertices. Let’s take a look.

A Quad mesh is constructed using two triangles, typically defined by linking 4 vertices.

Each of these vertices have a UV value assigned to them, the bottom left vertex, is assigned with the UV `(0, 0)`, the top left with `(0, 1)`, the top-right with `(1, 1)` and the bottom right with `(1, 0)`.



Those values are interpolated in the rasterization stage, and the fragment shader will receive interpolated UV values that will provide all the intermediate UVs automagically.

It's useful to notice that you can actually modify those UVs in the vertex and/or fragment shaders in order to alter how those texels are retrieved (or sampled in computer graphics jargon).

Wrapping Modes

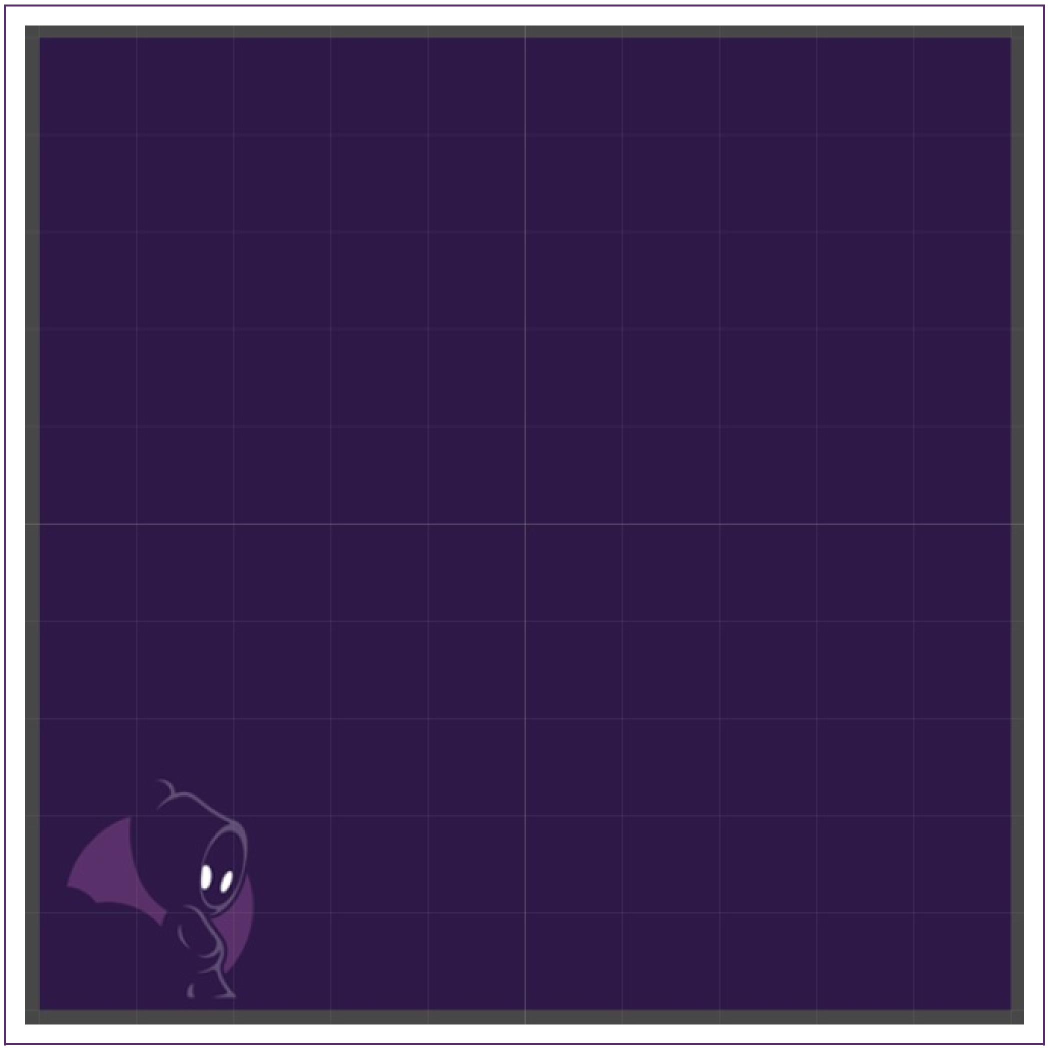
One more important thing to discuss is what happens if we overflow a UV component. By overflowing I mean, we sample a texture using a number that is greater than `1` or smaller than `0`.

In the RGB space, this makes no sense, there is nothing beyond `1`, because, as you may recall, `1` is the maximum value an RGB channel can take. So if we exceed `1`, it just gets clamped to `1` again, and the same if we go below `0`. But in UV mapping the story is different. We can actually tell the graphics card to behave in some useful ways.

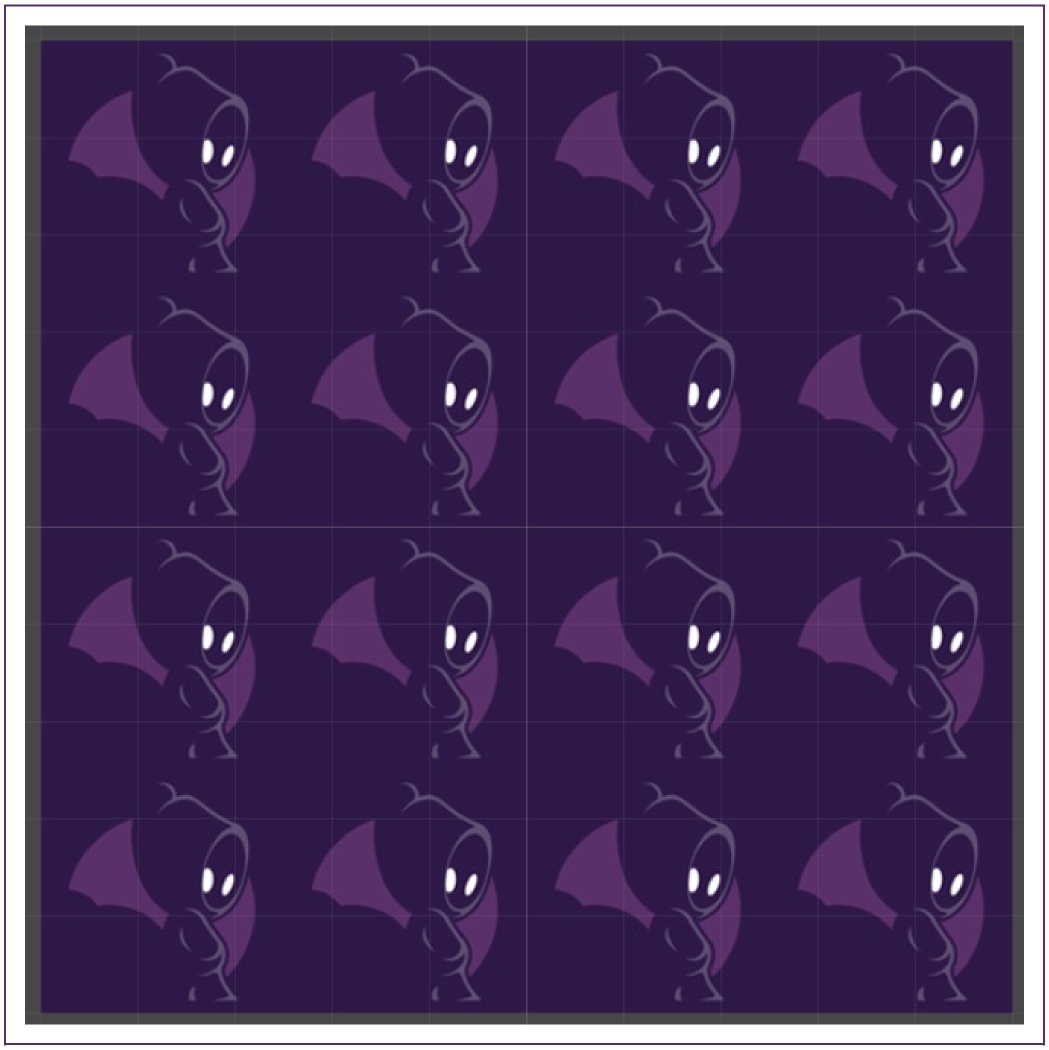
If you select a sprite or texture in the Project tab and take a look at the inspector, you'll see there are many more options to play with. For now, we want to focus on Wrap Mode.

Wrap Mode is used to make a decision about what to do when we exceed the bounds of a texture's UVs.

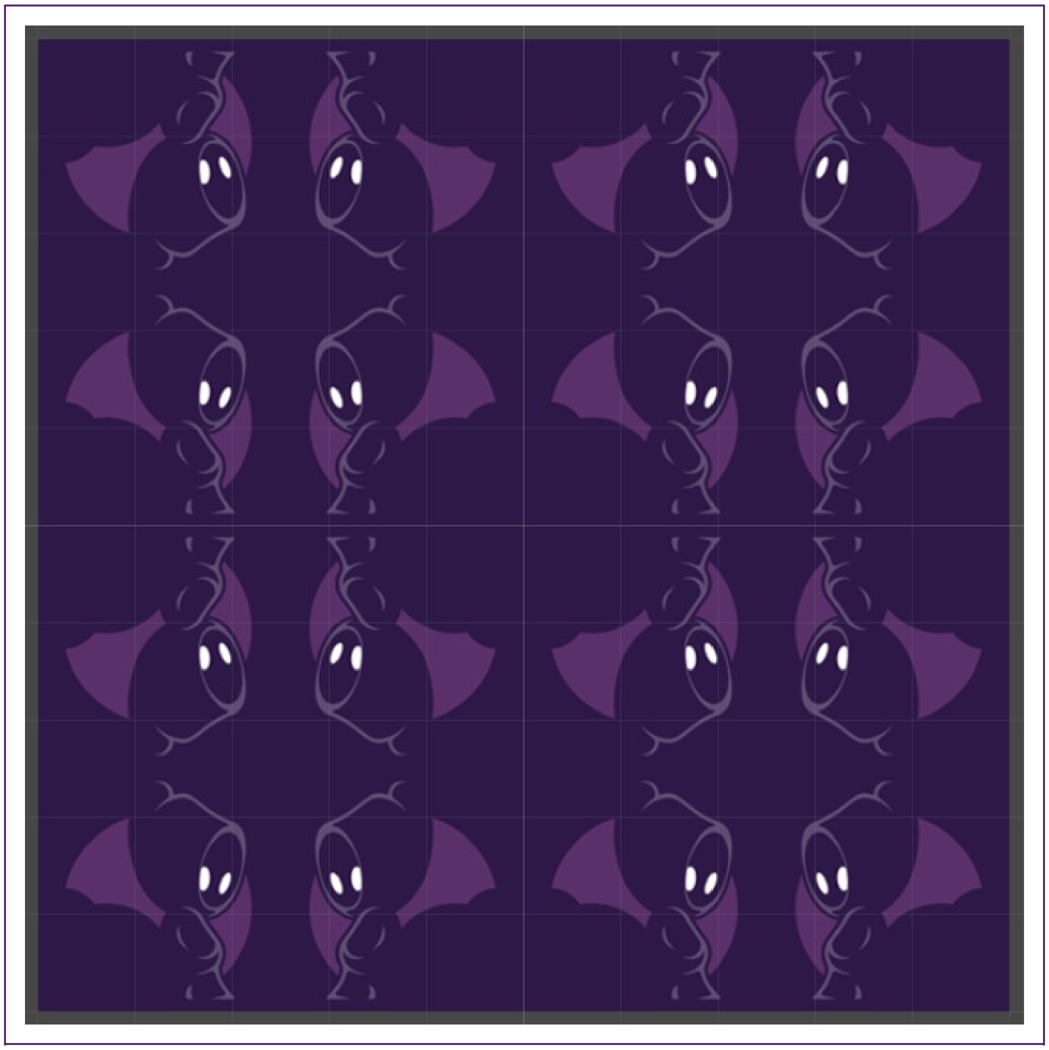
By default, it is set to Clamp mode. This mode will repeat the last pixel when we exceed `1` or the first pixel when we go below `0`. Similar to what happens with the RGB color space.



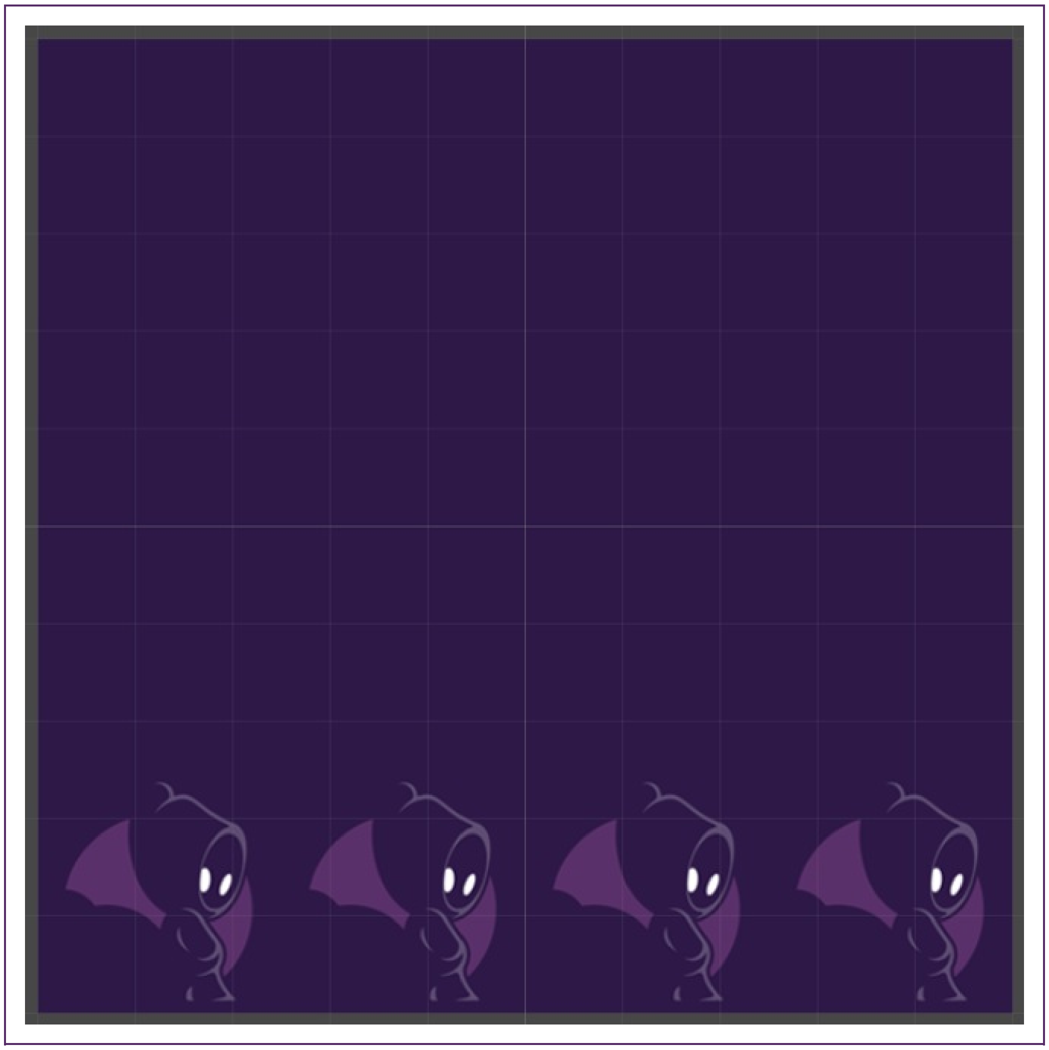
An interesting thing happens when we change Clamp to Repeat, which instead of using the last pixel, it will go back to the first one, creating an infinite loop.



Another option is Mirror, which will invert the image whenever you exceed `1`. This can be really useful to quickly break repetition in a sprite.



In recent versions of Unity, you can even set a different wrapping mode for each axis, being able to, for example, Repeat in `u` and Clamp in `v`.



In this way, we can increase the UV sampling parameter constantly and always have something coherent to show. This will be really useful for the first exercise you'll work on at the end of this chapter.

In these examples, I'm multiplying the uv by `4`, so that instead of sampling from `(0,0)` to `(1,1)` I'm sampling to `(4,4)`. In this way, I exceed the `[0,1]` range and I can show you how the Wrap Mode behaves when overflowing.

Now you know all you have to know to write a shader that renders a texture, let's do it!

Writing a Shader that displays a Texture

This is the first step towards our long journey creating unique looking games. We're going to create a shader that receives a texture as a parameter and displays it on the screen. This is the basic functionality we need to display a sprite.

First of all, let's duplicate our `SolidColor` shader and name it Texture and open it. Change the name to `"2D Shaders/Texture"`.

Main Texture parameter

When we made an introduction to the ShaderLab language, we introduced the concept of Properties. In order to pass data to our shader (in this case the texture itself), we have to rely on a Property. We'll have to create a property for this texture inside the `Properties` block.

We'll call this property `_MainTex` which is the standard name for main textures in Unity.

```
Shader "2D Shader/Texture"
{
  Properties
  {
    _MainTex ( "Main Texture", 2D ) = "white" {}
  }
```

The structure is pretty simple, we write the name of the property, `_MainTex`, then between parenthesis we define the external name and the property type, and finally the default value.

The external name is used to identify the property in Unity's editor in a more human-readable way, any string should be fine for this as long as it provides the user of the shader an understanding of what the property is.

There are several property types. You can take a look at all of them in Unity's documentation. In our case, we want to use the type called `2D`, which is used for two-dimensional textures like the sprites we want to render.

Then, we define a default value of white and we put brackets. Those brackets were used before Unity 5 to pass some parameters to the texture which we won't be doing.

Now that we have our property let's write the shader code.

Changes to the `appdata` struct

In the previous shader we wrote, we used appdata to get the position of the vertex. We'll do that here as well because it's required by Unity, but we will also add a `float2` field called `uv`, which will be the uv coordinate for the vertex. In the previous section, we learned what UV coordinates were. Here is where we put that theory in practice.

```
struct appdata {
    float4 position : POSITION;
    float2 uv : TEXCOORD0;
};
```

We want to tell Unity that the `uv` member is actually the place where we want it to put the uv coordinate from the mesh, for this we use the semantic `TEXCOORD0`. Remember that our Quad mesh already comes with a built-in set of uvs. This semantic is the glue between our shader and the mesh. That's it for the appdata struct.

Changes to the `v2f` struct

Our `v2f` struct, the parameter to our frag function, needs the uv information as well, so we'll add it there again as a `float2`, using the same semantic `TEXCOORD0`.

```
struct v2f {
    float4 position : SV_POSITION;
    float2 uv : TEXCOORD0;
};
```

Defining the `_MainTex` variable inside our shader

This is a really common source of errors, so pay special attention. When we define properties in ShaderLab (in our case `_MainTex`), they're just exported to Unity's editor but are not defined inside the shader (that is between `CGPROGRAM` and `ENDCG`) to be used. You'll have to define it yourself.

```
struct v2f {
    float4 position : SV_POSITION;
    float2 uv : TEXCOORD0;
};

sampler2D _MainTex;

v2f vert(appdata v)
{
```

Since we're using a 2D texture, the data type for it has to be `sampler2D`. The name of the variable needs to match the name of the property you defined in the `Properties` block. In this case, it's obviously `_MainTex`.

The vertex shader

We don't want to do much in our vertex shader, we just want to do a pass-through to the fragment shader. The vert is basically the same thing as the one used in our `SolidColor` shader, but with the addition of passing the `uv` member to the fragment shader.

```
v2f vert (appdata v) {
    v2f o;
    o.position = UnityObjectToClipPos(v.position);
    o.uv = v.uv;
    return o;
}
```

The fragment shader

This is where the magic will actually happen. Assigning the `uv` value in the vertex shader set everything up so that the rasterized could interpolate the intermediate uv values required for each fragment. Now it's time to use those interpolated values in the fragment shader to get the corresponding pixel. This process is called a texture lookup. For this we need to make some changes to our `frag` function:

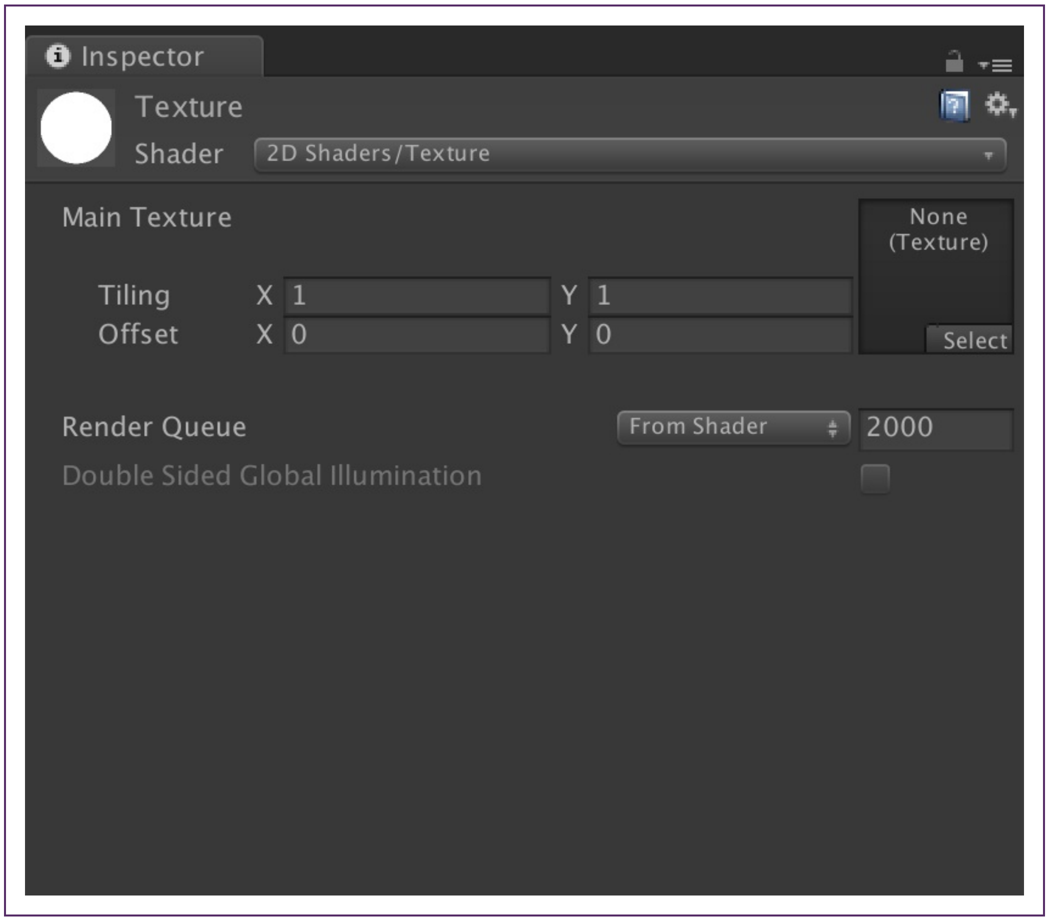
```
fixed4 frag ( v2f i ) : SV_Target {
    fixed4 col = tex2D(_MainTex, i.uv);
    return col;
}
```

The texture lookup is achieved in the fragment shader by using the `tex2D` method, that expects a `sampler2d` and a `uv` pair as a `float2` and returns a `float4` that contains the color for that texel. There are several possible parameter combinations in the `tex2d` method, we're only going to use this one in this book series, but be sure to check Nvidia's documentation on tex2d.

In terms of the shader, we're done now. There are still some things that we need to do in Unity to see the sprite though.

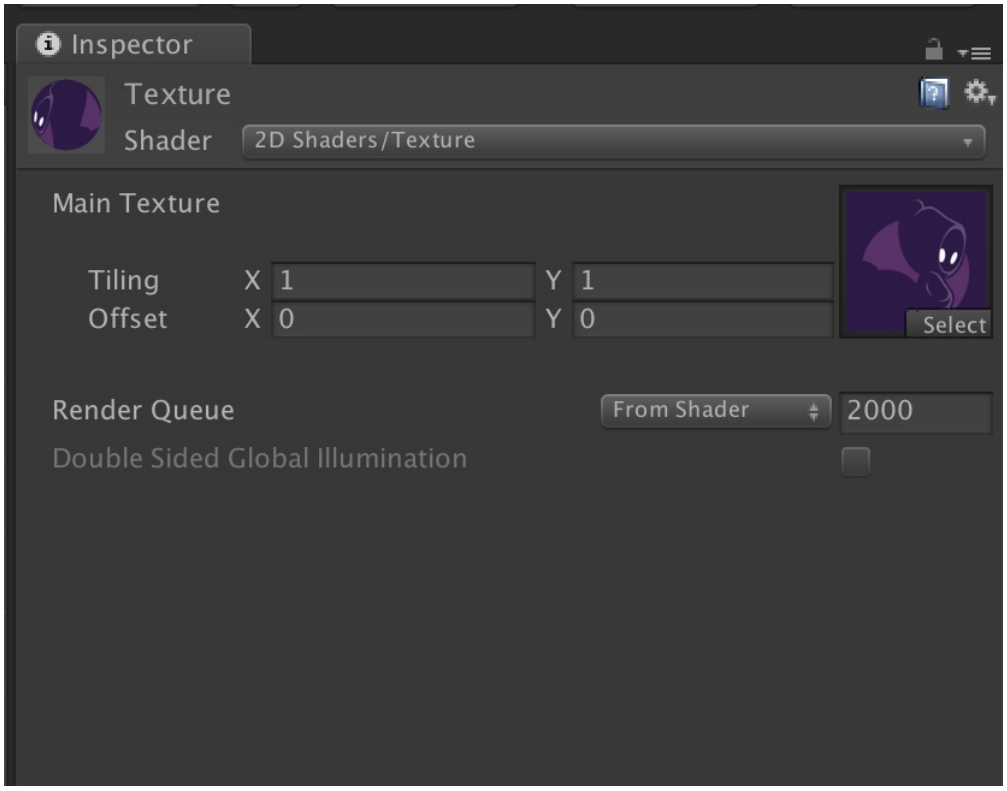
Setting up the material

Let's create a new material called Texture and select the `2D Shaders/Texture` shader.



Texture
Shader: 2D Shaders/Texture
Main Texture: None (Texture)
Tiling X 1 Y 1
Offset X 0 Y 0
Render Queue: From Shader 2000
Double Sided Global Illumination

In the inspector, you should now see that there is an option to set up the Main Texture from the editor, and the material should look white (remember we set white as the default value for `_MainTex`). You can drag and drop a texture from the Project tab into the square that says None (Texture).


Inspector

Texture

Shader: 2D Shaders/Texture

Main Texture

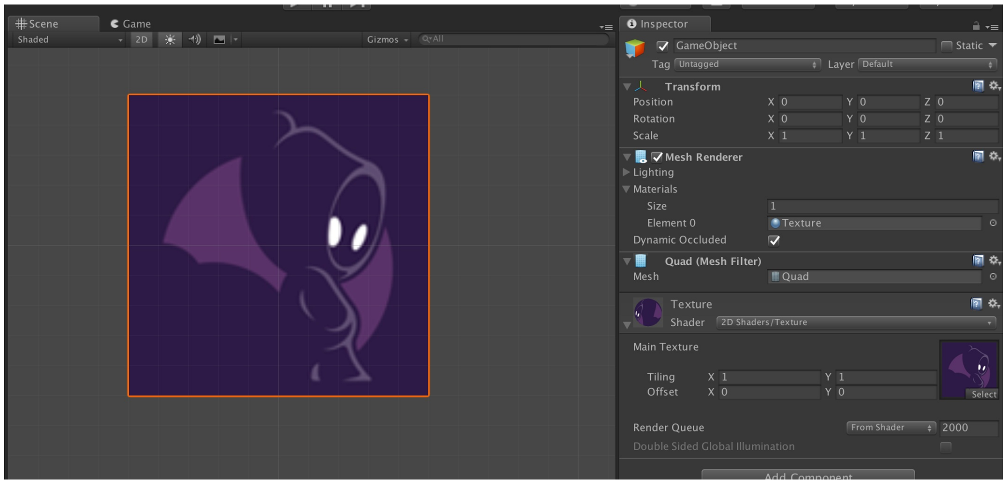
Tiling X 1 Y 1

Offset X 0 Y 0

Render Queue: From Shader 2000

Double Sided Global Illumination ☐

Then is just a matter of telling our GameObject to use this new material.



GameObject ☐ Static

Tag: Untagged Layer: Default

Transform

Position X 0 Y 0 Z 0

Rotation X 0 Y 0 Z 0

Scale X 1 Y 1 Z 1

Mesh Renderer

Lighting

Materials

Size 1

Element 0: Texture

Dynamic Occluded ✓

Quad (Mesh Filter)

Mesh: Quad

Texture

Shader: 2D Shaders/Texture

Main Texture

Tiling X 1 Y 1

Offset X 0 Y 0

Render Queue: From Shader 2000

Double Sided Global Illumination ☐

We now have a sprite rendering on our screen. I know what you're thinking: this is a lot of work for something you can do in two clicks using the Sprite GameObject in Unity, and I would normally agree if we were only looking for this. But understanding how the shader that the Sprite Renderer uses works is essential to getting into the good stuff later. Now let's try to put the stuff we learned into practice with an exercise.

Exercise 1: Side Scroller Background

Congratulations! Now you have enough knowledge to write a shader that is useful for a game! In this case, I'll ask you to take your time to create the background of a side scroller game.

I assume you already downloaded the source code for the exercises if you haven't this is the time to go ahead and do it.

Once you have the source code downloaded, open the project called `Exercise 1 - Side Scroller Background`.

The idea is to use the texture shader we created in the previous section and modify it to create a background that scrolls infinitely.

Also, you'll need to be able to set the speed by which it moves using a shader Property.

You can find the solution to this exercise at the end of the book.

Conclusion

This chapter was full of new and interesting stuff, wasn't it? We covered the basics of fragment shader usage. We learned the difference between Vector and Fragment shaders, how colors are represented in the graphics card in RGBA, how to mix them to create new colors and how textures get mapped to vertices using UV coordinates.

We also wrote some shaders. We implemented a shader that creates a solid color for the whole sprite and created a shader that displays an opaque texture. You also worked on an assignment to create a side-scroller background with a variable to handle its speed. You can consider your shader programming path started for real!

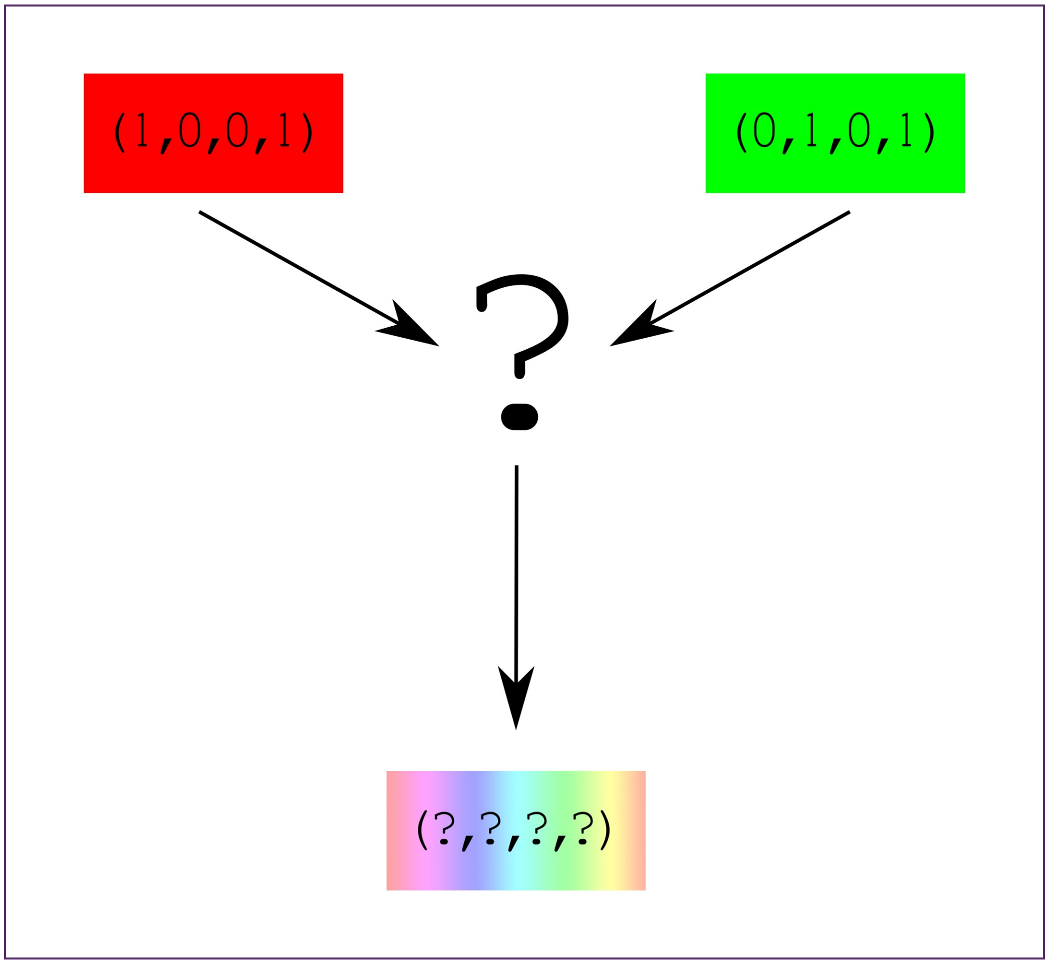
In the next chapter, we’ll start learning about how to mix colors in different ways, this is called Blending modes. We’ll learn how different math operations affect the mixing of colors, and we’ll also learn how to make our sprites use transparency.

Blending Modes

Up until now, we've been dealing with just one source color (one created by us or fetched from a texture) but now it's time to start mixing more than one color. For that, we have Blending Modes. This chapter will cover how to blend colors inside a shader (that is two colors created by us or two fetched from textures) and blending with the existing contents of the screen (the background).

The theory behind Blending Modes

Now that we have a decent amount of background on how to create basic shaders, is time to learn about Blending Modes. Imagine you have two colors, let's say, red and green, and you want to combine them. How many ways do you think there are to combine them?



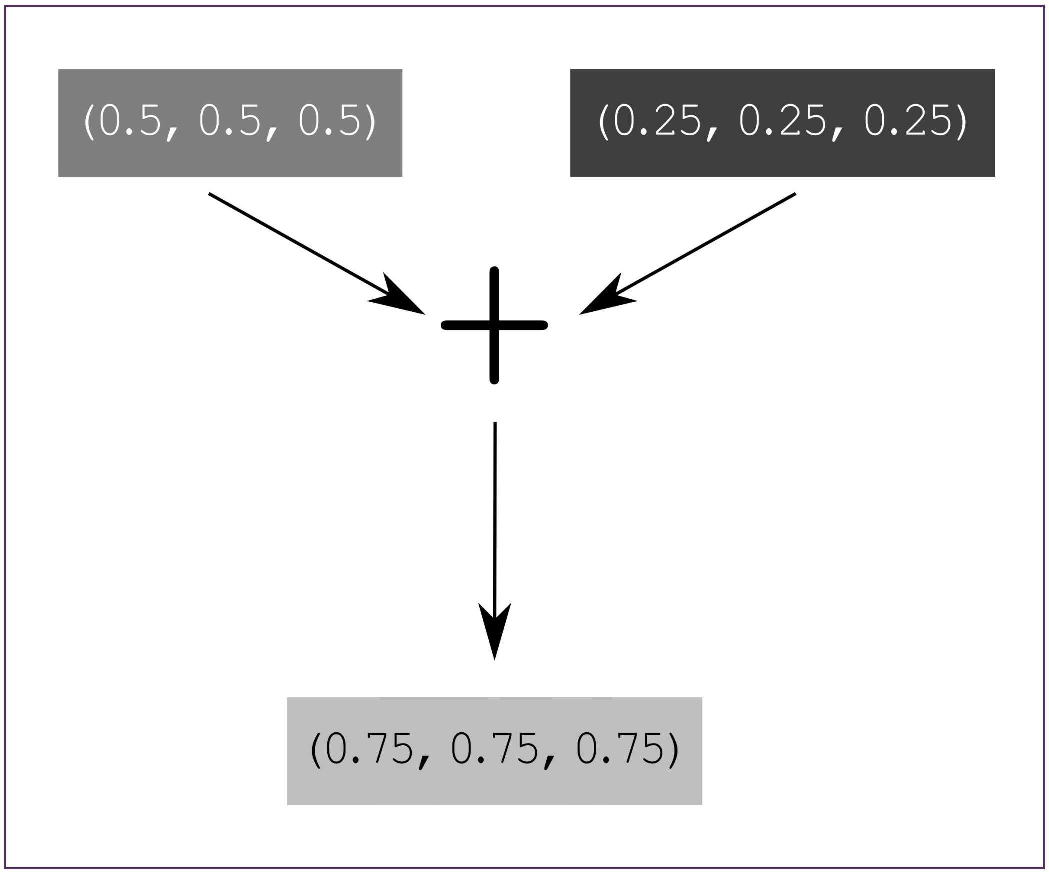
Stop counting already. There are infinite. In computer graphics, we represent colors as vectors of 3 or 4 components, and, as you may already have guessed, we can use any mathematical operation we normally use with vectors. Those operations will have different results in what colors we see.

Blending modes are nothing more than a bunch of mathematical functions that ended up being really useful to mix colors and create effects. Let's see what are the typical ones.

Additive

The most basic blending mode is additive, we add the colors, component by component, and end up having a third color that is the sum of the previous two. If you think about what happens numerically when you sum two colors, you'll notice that the result is always closer to white because colors go between `0` and `1` (they're positive) so adding them will yield something that is closer to `1` than each color alone.

For example, if we add a mid gray (0.5, 0.5, 0.5) with a quarter gray color (0.25, 0.25, 0.25), you'll get a three quarter gray (0.75, 0.75, 0.75), which is closer to white.

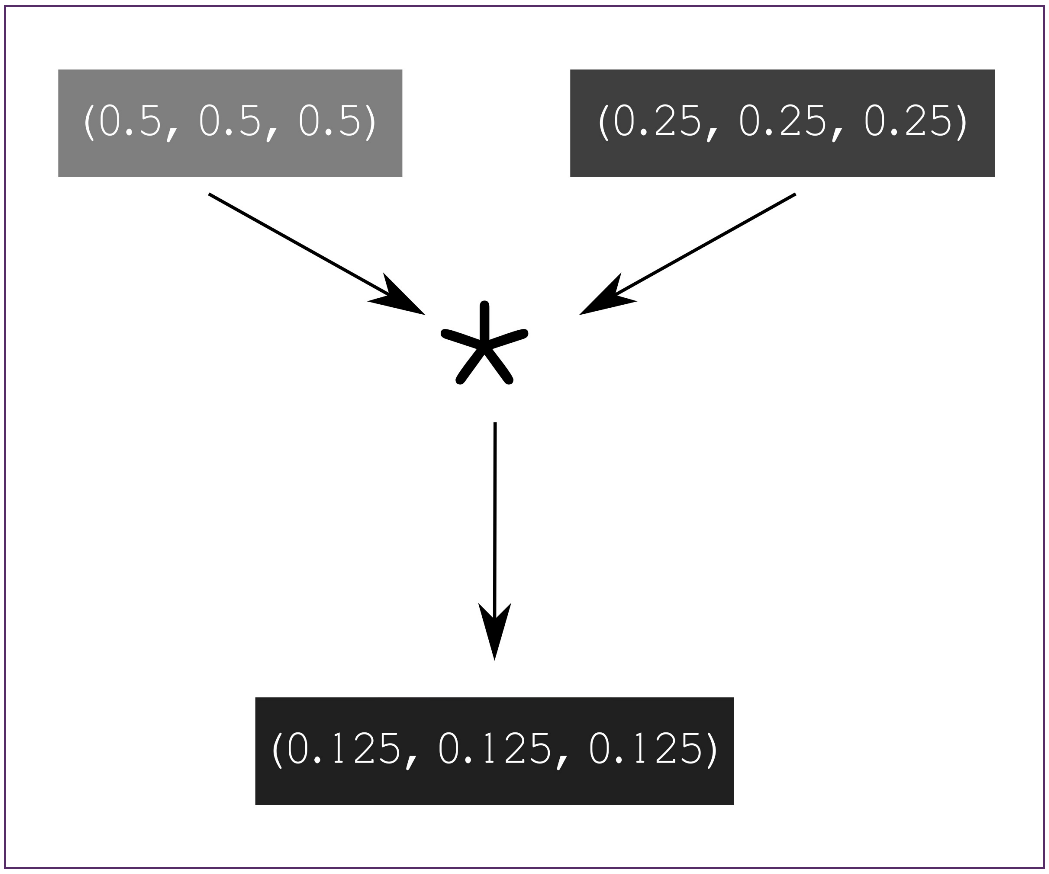


The consequence of this property is that using an Additive blending mode is really useful to light things up. When you see particle systems in games that look like really bright (like bolts of lightning, fireballs, magic missiles, etc) they're almost always using additive blending mode.

Multiply

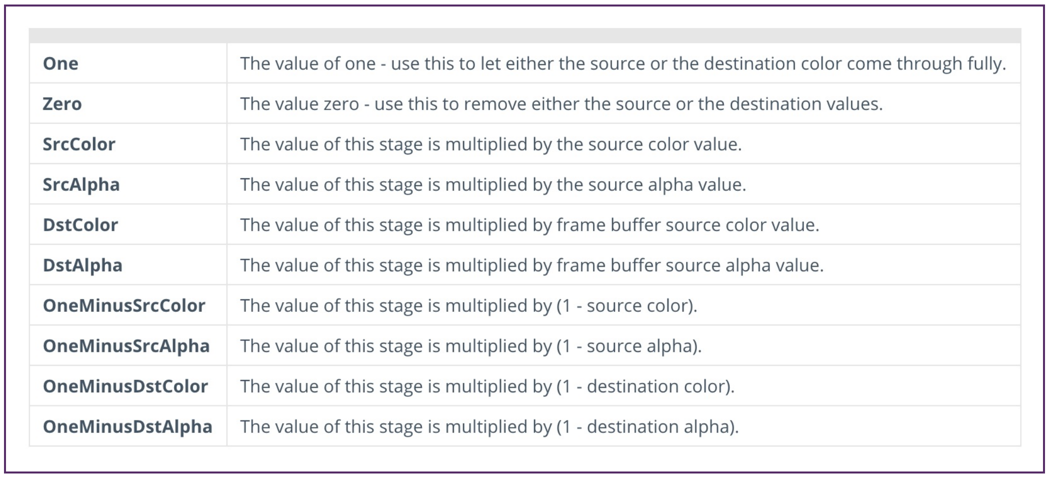
Another useful blending mode is Multiply, in which we multiply both colors component by component. Again if we analyze what happens numerically, you'll notice that the result of multiplying two colors is almost always making the color closer to black. This is because multiplying a number by something between zero and one always returns a smaller number (excluding one, that doesn't change the original number).

Using the example above, (0.5, 0.5, 0.5) by (0.25, 0.25, 0.25) results in (0.125, 0.125, 0.125) which is closer to black. This blending mode is super useful to create shadows.



Other Blending Modes

There are several other blending modes that are quite common. One thing I usually suggest to my students is that besides knowing several blending modes, you learn how they're called in Photoshop since artists usually refer to them with that name. For example, in Photoshop, Additive is known as Linear Dodge. Here is a table of the math behind some of the blending modes available in Photoshop (Source: http://photoblogstop.com/photoshop/photoshop-blend-modes-explained A really nice article to understand blending modes in Photoshop) you can always refer to this when asked from an artist to apply a certain blending mode on a layer.



One	The value of one - use this to let either the source or the destination color come through fully.
Zero	The value zero - use this to remove either the source or the destination values.
SrcColor	The value of this stage is multiplied by the source color value.
SrcAlpha	The value of this stage is multiplied by the source alpha value.
DstColor	The value of this stage is multiplied by frame buffer source color value.
DstAlpha	The value of this stage is multiplied by frame buffer source alpha value.
OneMinusSrcColor	The value of this stage is multiplied by (1 - source color).
OneMinusSrcAlpha	The value of this stage is multiplied by (1 - source alpha).
OneMinusDstColor	The value of this stage is multiplied by (1 - destination color).
OneMinusDstAlpha	The value of this stage is multiplied by (1 - destination alpha).

Blending Two Textures

The easiest thing we can do to experiment with Blending Modes is extending our `Texture` shader to receive two textures and combine them. Go ahead and duplicate the Texture shader and rename it to `TwoTextures`. Don't forget to change the name inside the shader after the `Shader` command.

First of all, add a new property called `_SecondTexture`.

```
Properties {
    _MainTex ( "Main Texture", 2D ) = "white" {}
    _SecondTexture ( "Second Texture", 2D ) = "white" {}
}
```

After that, we also add the `sampler2D` variable associated to the `_SecondTexture` property.

```
sampler2D _MainTex;
sampler2D _SecondTexture;
```

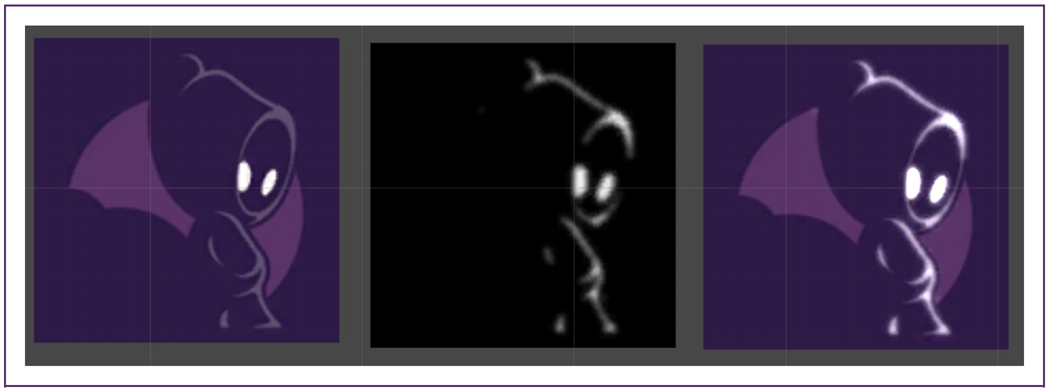
In the fragment shader, we'll want to sample a second color, taken from `_SecondTexture` so that we can blend them.

```
fixed4 frag (v2f i) : SV_Target
{
  fixed4 col1 = tex2D(_MainTex, i.uv);
  fixed4 col2 = tex2D(_SecondTexture, i.uv);
  return col1;
}
```

In `col1` we have the color of the texel in `_MainTexture`, in `col2` we have the color of the texel in `_SecondTexture`. Now we can apply the different blending modes we learned in the previous section.

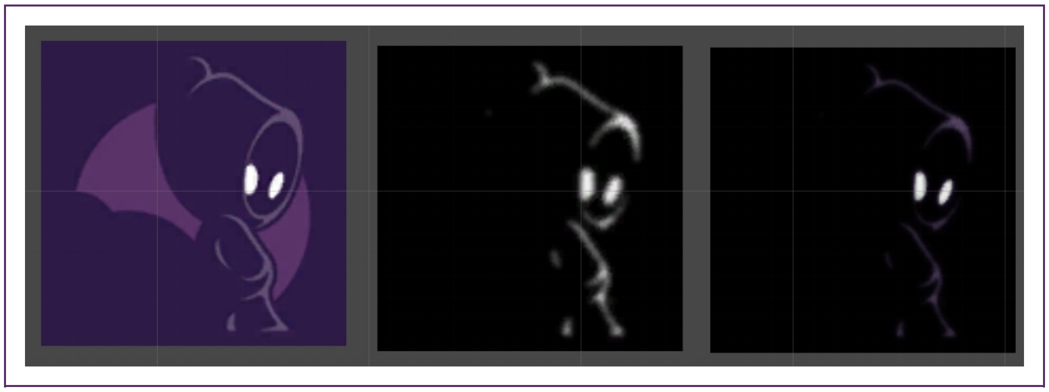
For example, we can add both textures and see what happens.

```
fixed4 frag (v2f i) : SV_Target
{
  fixed4 col1 = tex2D(_MainTex, i.uv);
  fixed4 col2 = tex2D(_SecondTexture, i.uv);
  return col1 + col2;
}
```



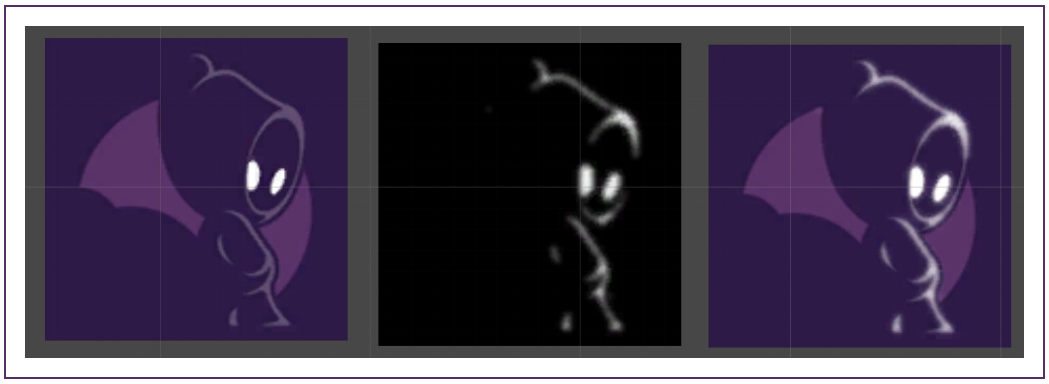
We can also multiply them:

```
fixed4 frag (v2f i) : SV_Target
{
  fixed4 col1 = tex2D(_MainTex, i.uv);
  fixed4 col2 = tex2D(_SecondTexture, i.uv);
  return col1 * col2;
}
```



Or we can apply screening, which is a soft additive blending mode.

```
fixed4 frag (v2f i) : SV_Target
{
  fixed4 col1 = tex2D(_MainTex, i.uv);
  fixed4 col2 = tex2D(_SecondTexture, i.uv);
  return 1 - (1 - col1) * (1 - col2);
}
```



In the next section, we'll learn how to blend one texture with the pixels already rendered behind it using ShaderLab `Blend` and `BlendOp` commands.

Blending with the Screen

After experimenting a bit with blending two textures inside an object, we will want to extend this to the screen. It's a shame that we don't have the same amount of flexibility for this, but we can do a lot anyway.

When the fragment shader is done, Unity's graphics pipeline has to decide how to merge the colors we returned in it with the existing colors in the buffer. For that, we use ShaderLab's `Blend` command. Let's take a look at how this works, but first, we need to do some setup.

First of all, duplicate the `Texture` Shader and call it `BlendingModes` and remember to change its name in the shader code to `"2D Shaders/Blending Modes"`.

Now duplicate the `Texture` material and rename it to `BlendingModes` too. Also, change the shader of the material to the one we just created.

The last thing you'll have to do is duplicating the GameObject we're using to test things. We'll also have to assign the new material to it in the `MeshRenderer`.

The Blend command

Open up the `BlendingModes` shader in your text editor and before the `Pass` command, let's add `Blend Off`. That's the default behavior, which won't do any blending, it will overwrite the existing pixel in the buffer with the one we're returning in our frag function.

```
SubShader {
    Blend Off

    Pass {
```

If we check at the reference for the [`Blend` command](), we'll see there are several parameter combinations to it. Let's take a look at the first one, which is `Blend SrcFactor DstFactor`.

As the documentation says, the color that we return in our frag shader is multiplied by `SrcFactor`, the color on the screen is multiplied by `DstFactor` and then they're added together.

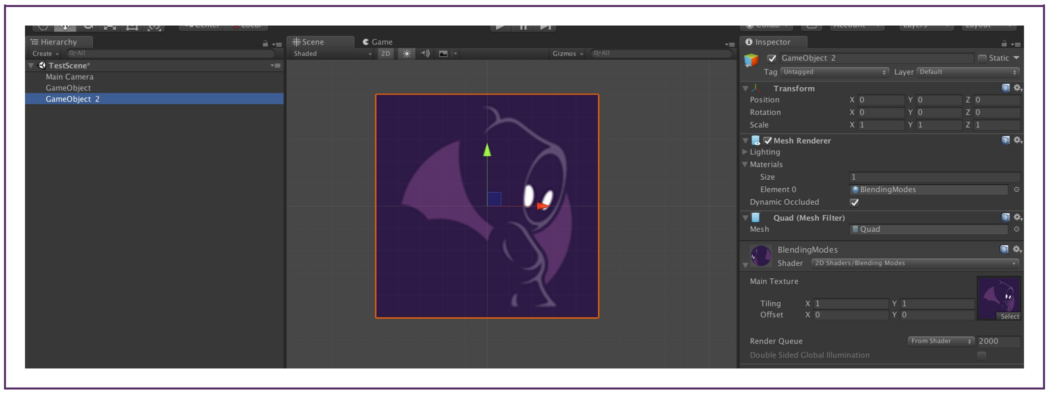
It is helpful for me to think about the math function that we end up having if we choose one factor or the other.

```
final_color = my_color * SrcFactor + screen_color * DstFactor
```

For example, to make additive blending, what we want is to have both colors added together. So writing the equation shows that both factors have to be one.

```
final_color = my_color * One + screen_color * One
```

The default value is `Blend Off`, which should look like this:



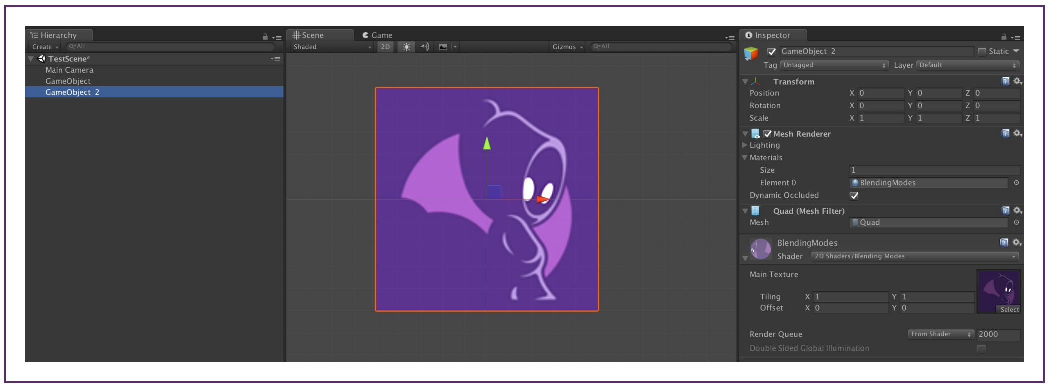
Hierarchy
TestScene*
Main Camera
GameObject
GameObject 2

Inspector
GameObject 2 — Static
Tag: Untagged — Layer: Default
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Mesh Renderer
Lighting
Materials
Size 1
Element 0 BlendingModes
Dynamic Occluded ✓
Quad (Mesh Filter)
Mesh Quad
BlendingModes
Shader 2D Shaders/Blending Modes
Main Texture
Tiling X 1 Y 1
Offset X 0 Y 0
Render Queue From Shader 2000
Double Sided Global Illumination

As you can see `GameObject 2` is on top of `GameObject` and it hides it, because `Blend Off` will just overwrite the existing pixel value.

If we change `Blend Off` to `Blend One One`, we'll see how our image on top is added to the one we already had.

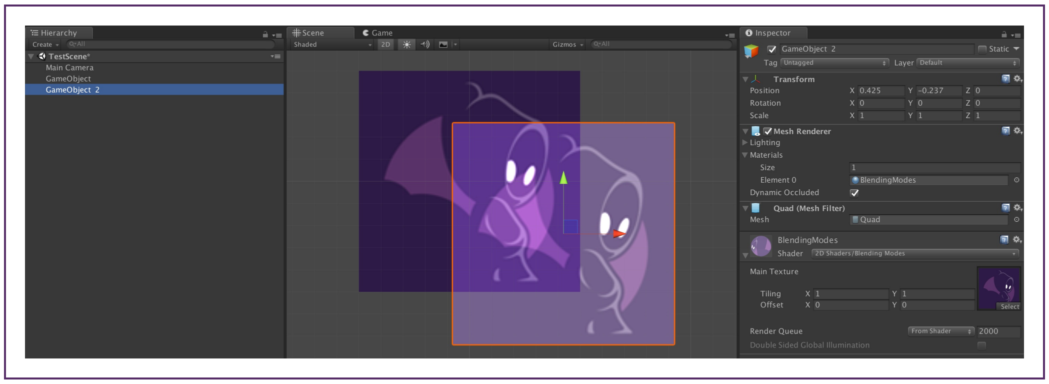
```
SubShader {
    Blend One One

    Pass {
```



Hierarchy
TestScene*
Main Camera
GameObject
GameObject 2

Inspector
GameObject 2 — Static
Tag: Untagged — Layer: Default
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Mesh Renderer
Lighting
Materials
Size 1
Element 0 BlendingModes
Dynamic Occluded ✓
Quad (Mesh Filter)
Mesh Quad
BlendingModes
Shader 2D Shaders/Blending Modes
Main Texture
Tiling X 1 Y 1
Offset X 0 Y 0
Render Queue From Shader 2000
Double Sided Global Illumination

As you can see, the image is lighter, this is because we're adding it to itself. Let’s move the top one to check the intersection and see how this works.



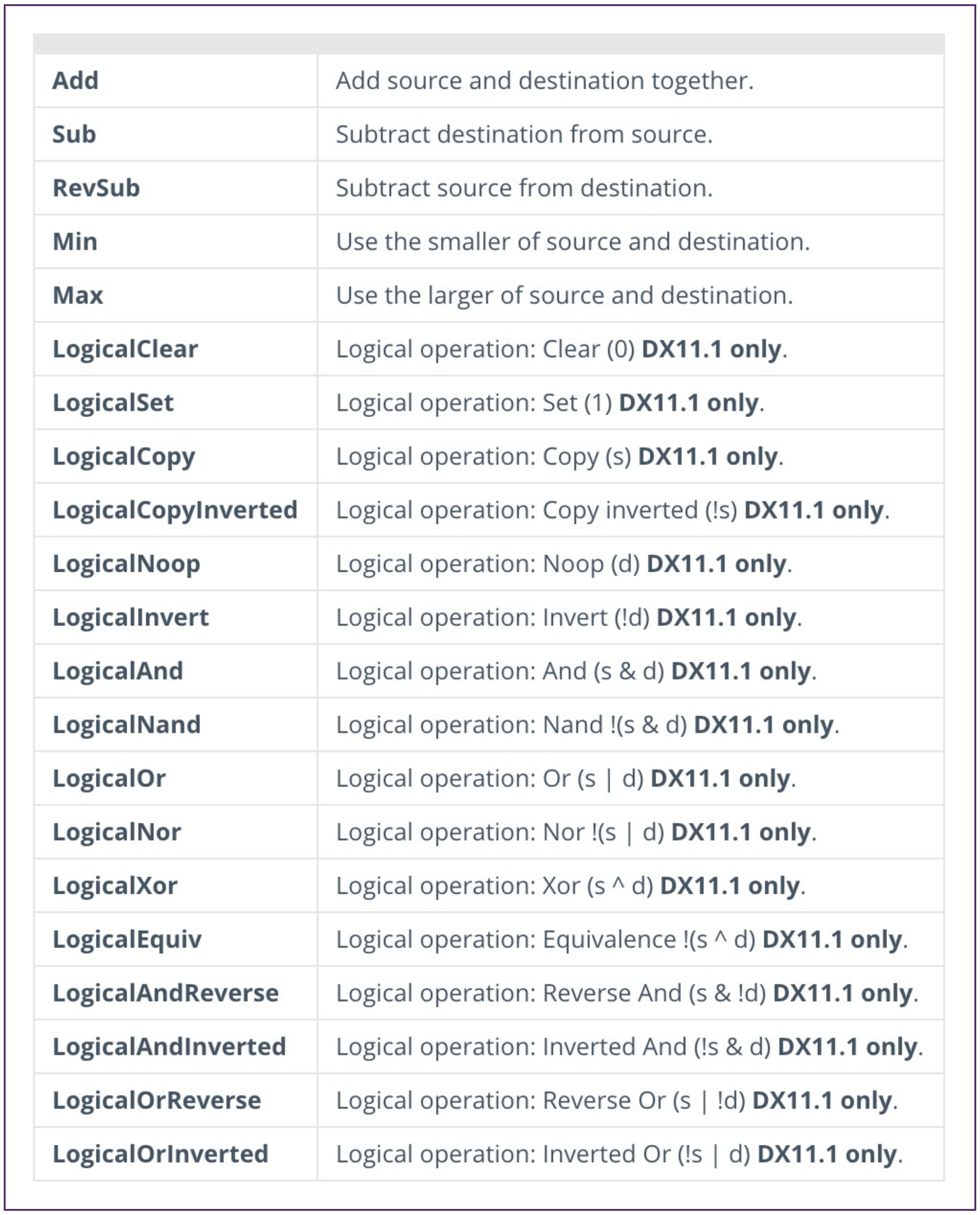
GameObject 2 Static
Tag Untagged Layer Default
Transform
Position X 0.425 Y -0.237 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Mesh Renderer
Lighting
Materials
Size 1
Element 0 BlendingModes
Dynamic Occluded ✓
Quad (Mesh Filter)
Mesh Quad
BlendingModes
Shader 2D Shaders/Blending Modes
Main Texture
Tiling X 1 Y 1
Offset X 0 Y 0
Select
Render Queue From Shader 2000
Double Sided Global Illumination

I suggest you experiment reviewing the reference and see what you can find, to understand the effects on the image, you can always set one factor to `Zero` and try stuff with the other, and then combining them. It’s interesting how much you can achieve.

There are other versions of the `Blend` command, for example, you can also pass 4 different factors and it will be the same as before, but differencing alpha, this expands our possibilities, because alpha is one thing on its own and it’s nice to be able to do computations separately. I encourage you to experiment with this.

The `BlendOp` command

Adding the result of the multiplications is just the default behavior, but we may want to do another operation between both colors. There is another command called `BlendOp` that allows you to change this behavior.



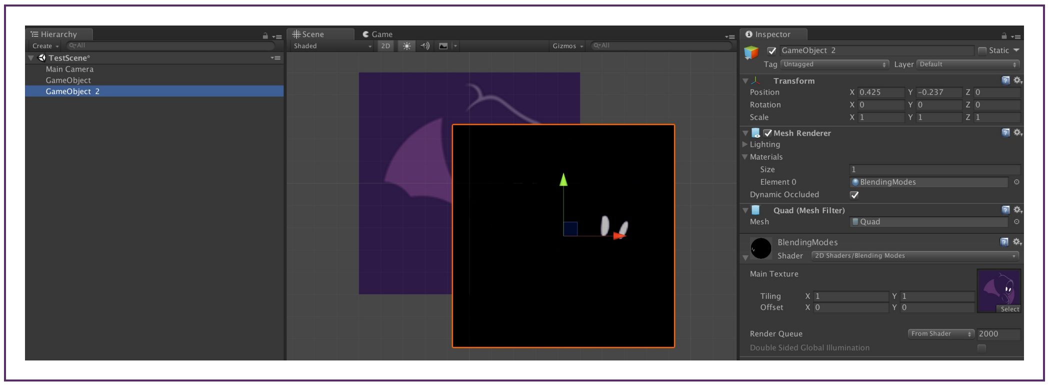
Add	Add source and destination together.
Sub	Subtract destination from source.
RevSub	Subtract source from destination.
Min	Use the smaller of source and destination.
Max	Use the larger of source and destination.
LogicalClear	Logical operation: Clear (0) **DX11.1 only**.
LogicalSet	Logical operation: Set (1) **DX11.1 only**.
LogicalCopy	Logical operation: Copy (s) **DX11.1 only**.
LogicalCopyInverted	Logical operation: Copy inverted (!s) **DX11.1 only**.
LogicalNoop	Logical operation: Noop (d) **DX11.1 only**.
LogicalInvert	Logical operation: Invert (!d) **DX11.1 only**.
LogicalAnd	Logical operation: And (s & d) **DX11.1 only**.
LogicalNand	Logical operation: Nand !(s & d) **DX11.1 only**.
LogicalOr	Logical operation: Or (s | d) **DX11.1 only**.
LogicalNor	Logical operation: Nor !(s | d) **DX11.1 only**.
LogicalXor	Logical operation: Xor (s ^ d) **DX11.1 only**.
LogicalEquiv	Logical operation: Equivalence !(s ^ d) **DX11.1 only**.
LogicalAndReverse	Logical operation: Reverse And (s & !d) **DX11.1 only**.
LogicalAndInverted	Logical operation: Inverted And (!s & d) **DX11.1 only**.
LogicalOrReverse	Logical operation: Reverse Or (s | !d) **DX11.1 only**.
LogicalOrInverted	Logical operation: Inverted Or (!s | d) **DX11.1 only**.

Instead of adding let's subtract both colors. For that, below `Blend One One`, we'll write `BlendOp Sub`.

```
SubShader {
    Blend One One
    BlendOp Sub

    Pass {
```

We can also play with the factor here, for example by squaring the `SrcColor` and subtracting.

```
SubShader {
    Blend SrcColor One
    BlendOp Sub

    Pass {
```



GameObject 2 Static
Tag Untagged Layer Default
Transform
Position X 0.425 Y -0.237 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Mesh Renderer
Lighting
Materials
Size 1
Element 0 BlendingModes
Dynamic Occluded ✓
Quad (Mesh Filter)
Mesh Quad
BlendingModes
Shader 2D Shaders/Blending Modes
Main Texture
Tiling X 1 Y 1
Offset X 0 Y 0
Render Queue From Shader 2000
Double Sided Global Illumination

With `BlendOp` we also have the option to pass a second parameter for the alpha operator. Which opens up way more possibilities.

I insist on taking your time to explore this space of possibilities, keeping in mind what is the actual mathematical function that is being applied and get a better understanding of the impact of each combination. This will help you create complex effects in the future.

Alpha Blending

Now that we learned how blending works with the existing screen buffer, we're in the position of being able to create transparent sprites. We'll do this by applying what's called Alpha

Blending.

Alpha Blending is a technique in which we mix the colors proportionally to the opacity of the color in the top. This means for example, that if we want to render a red color that's 70% opaque (That would be `(1, 0, 0, 0.7)`), we want to use 70% of that red, and 30% of the existing color. In order to do this, we'll have to configure blending with:

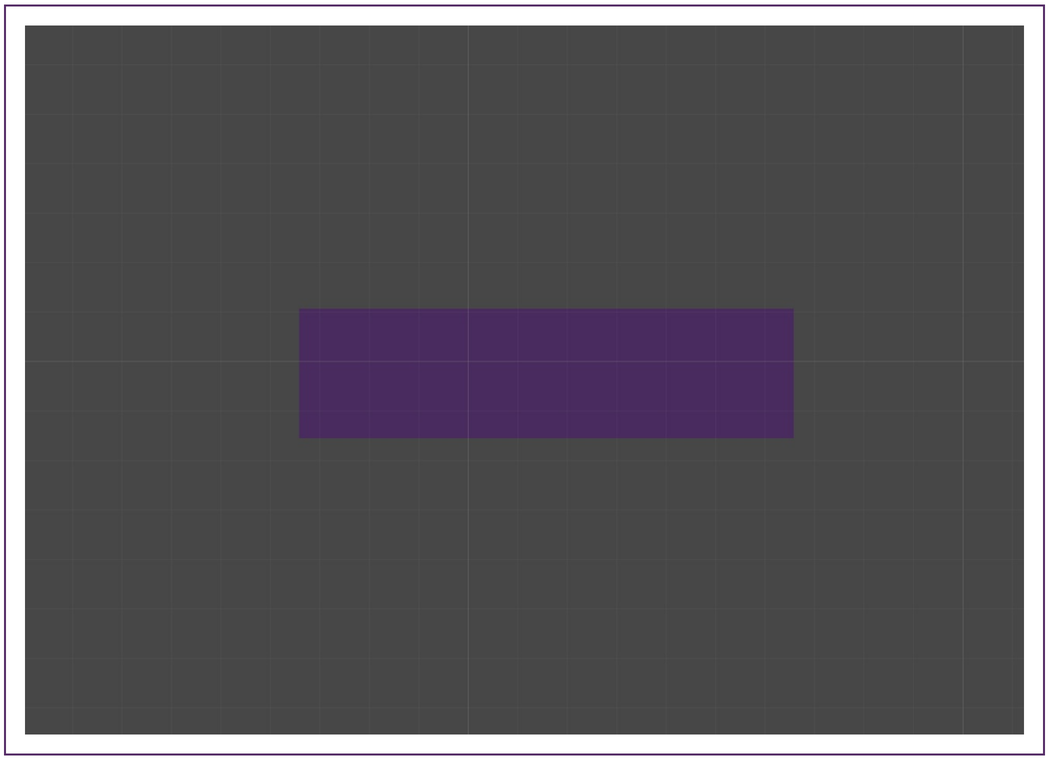
```
Blend SrcAlpha OneMinusSrcAlpha
```

If we take a look at the mathematical function that results from this `Blend` command, we'll understand why this works.

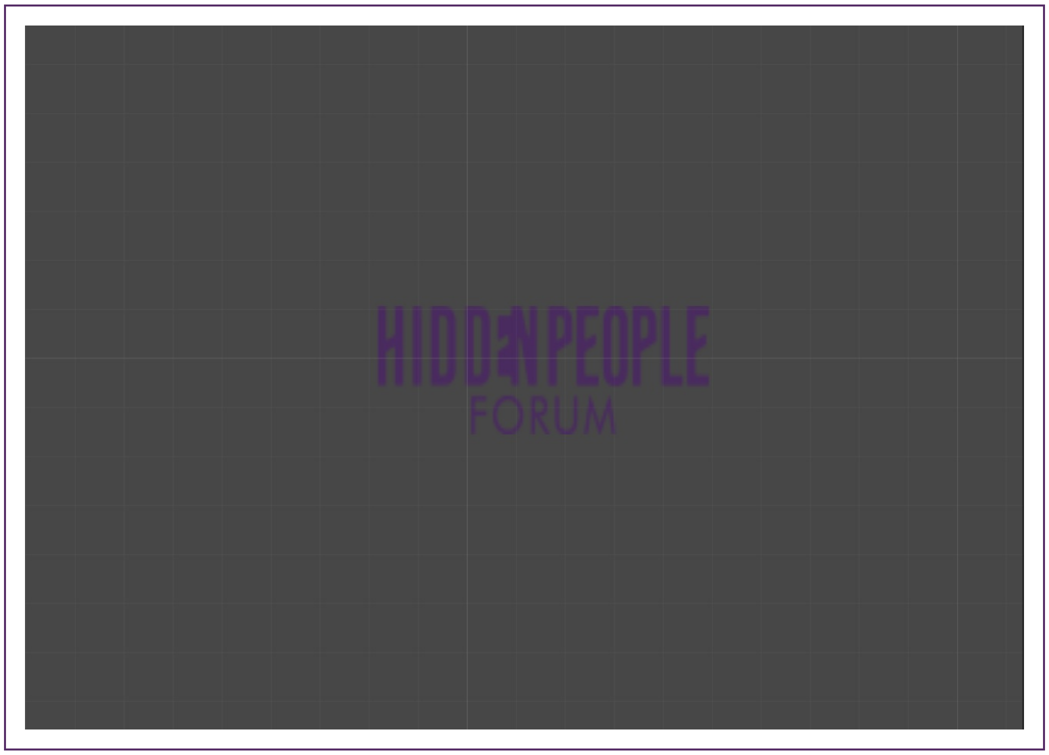
```
FinalColor = SrcColor * SrcAlpha + DstColor * (1 - SrcAlpha)
```

You can see why this works, right? Using `SrcColor` as `(1,0,0)` and Alpha `0.7` as in the previous example, we're multiplying the red color by `0.7` and the existing color by `1 - 0.7`, which is `0.3`.

Let's take a look at an image with transparency without Alpha Blend applied:



The problem is that, since alpha is not being honored, you see the background as a color. Let's add Alpha Blending to it.



Now we can properly see the image because the transparent sections are rendered as such.

Unity also has a specific Render Queue for transparent objects, so that they're rendered after opaque ones. This is a rendering technique used to allow for optimizations at the engine level on opaque objects, we don't care too much about it for now. To set this up, you want to add some Tags inside the `SubShader` block:

```
Tags {
  "Queue" = "Transparent"
  "RenderType"="Transparent"
}
```

If we use Sprites this is done automatically, but if we use a MeshRenderer with a Quad or some other mesh, which is actually really useful, we'll have to write this.

You'll need an image with an Alpha channel, and translucent pixels to test this out. The cool part is that this will work with any mesh we want, so if we're creating some strange topology we can add transparency to it with just a few lines of code, which is awesome.

Exercise 2: Blending Modes

You made it to the second exercise! This is exciting, now you have enough tools to work with more than one texture, both on a single sprite and with the screen.

In this exercise I'll ask you to open the scene called `Exercise 2 - Blending modes` and go through each of the sprites in the screen (except the background) and make them use the blending mode that its name suggests. For example, find the object called `Multiply` and make it use the Multiply blending mode. Do the same for Alpha Blend, Additive and Subtractive.

This exercise should be easy and quick to do, only remember to keep in mind the blending equation that Unity uses, which was presented in the previous sections.

Conclusion

In this chapter, we learned how to combine two or more colors inside a shader and how to mix the color from our shader with the existing colors in the screen buffer (the background). We did that by relying on `Blend` and `BlendOp` commands from ShaderLab.

If you are curious about what else can be done with Blending Modes be sure to check the 2D Illumination book in this series.

Where to go now?

Congratulations! You now have a solid foundation that will enable you to go on a much deeper learning career. Let's figure out what you can do next.

Continue with the other books in the series

This book series is designed to provide an introduction to 2D Shader Development. There are three well-defined branches that you can take now:

2D Illumination

In this book, you'll learn all about how you can create complex scenes using illumination. The book is split in two, static and dynamic illumination.

In the first part, you'll learn how you can effectively give the sensation of light and shadows to all the objects on the screen behind a certain layer. This is a super cheap and useful way to give depth to your game. It can also help you reuse a lot of assets, by illuminating them in different ways you break the monotony.

The second part is all about how you can create a dynamic light model pretty much like the one in 3D but to be used with 2D sprites. For this, you'll craft a separate texture with normals for the sprite and use Unity's lights in combination with it.

Procedural Texture Manipulation

In this book, you'll learn a few techniques that are used a lot in computer graphics to manipulate our textures with code. You'll go from a simple sine wave movement to complex combinations of textures animating other textures and crazy stuff like that.

You'll also learn about noise, you'll use Perlin Noise to animate sprites and create noise inside a shader.

Full-Screen Effects

This book is all about creating screen-space modifications. Using the rendered screen as an input texture you can apply all the stuff we learned in the series to the whole screen, and that's what you'll do in this book. You'll figure out how Bloom works, you'll implement several effects like camera shake, retro-looking filters like pixelating, and other useful things applying some of the theory behind DSP (Digital Signal Processing).

The internet

The second obvious option is to search the internet for examples of existing techniques you would like to learn and read articles about how you can implement them.

Reach out other developers that have done things you are excited about and ask them how they did it. This could be a major source of learning material!

Books

I can't recommend any books that are specific about 2D (That's the reason why I'm writing this!!!) but if you think you're ready to transfer the knowledge from 3D to 2D be sure to check an up to date list in the website for the book at https://www.2dshaders.com/what-to-do-now

Exercise Solutions

Exercise 1 Solution

First of all, I wanted to say thanks to the great community at opengameart.org, the background for this exercise was taken from there (by Alucard: https://opengameart.org/content/city-background-repetitive-3) Hopefully, you managed to get the exercise working fine and you're just validating your work. As we'll see this is not a difficult exercise at all.

First of all, I want to say there is a small trap in the exercise to make it more interesting, and there are two right ways of solving it, one more performant than the other, in the end, it won't matter much in this case. We'll analyze the least performant first.

In the project you downloaded, you'll find a shader called `EndlessScrollerBackground` that is already attached to a sprite through the `EndlessScrollerBackground` material. This shader is a copy of the Texture shader we created in the book.

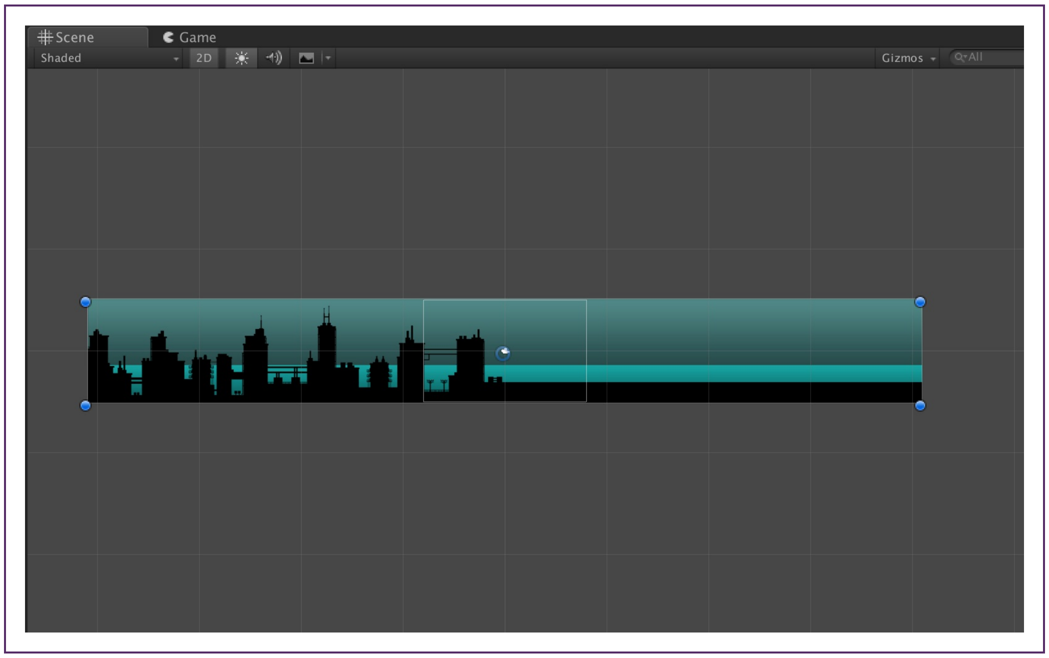
The first thing we want to do is to be able to move that texture to the left infinitely, so let's do that.

As you already know, the UV coordinates map the texture to the quad mesh. So in order to move the texture, we need to modify the UVs in some way.

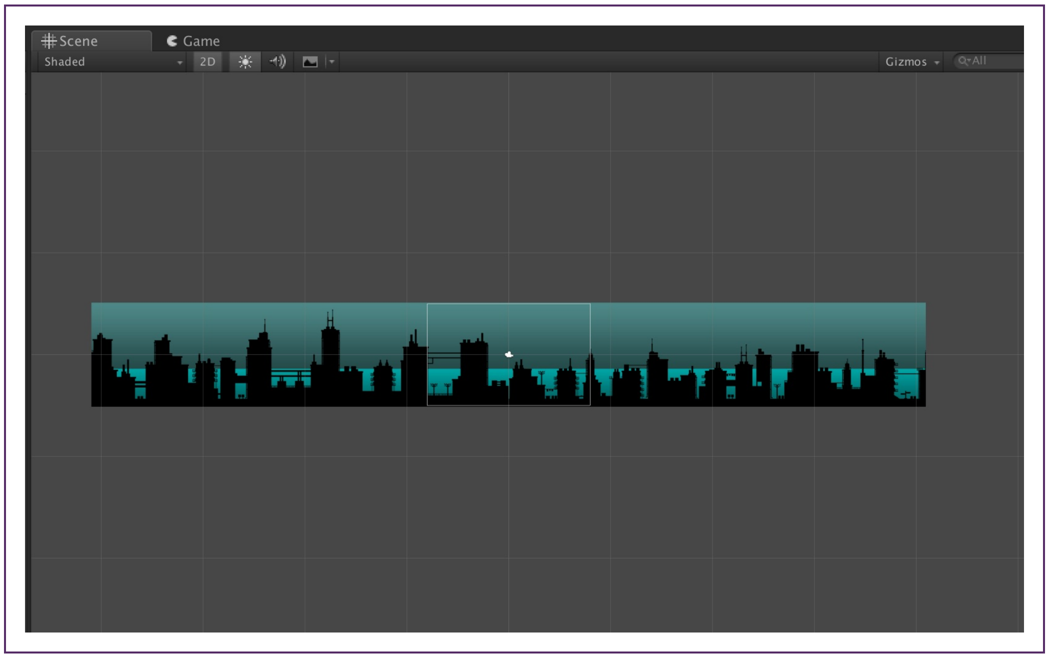
In order to move the texture to the left, we can add some value to the x component of the uv vector. This will shift the texture proportionally to that amount we add.

```
fixed4 frag (v2f i) : SV_Target {
    fixed4 col = tex2D(_MainTex, i.uv + float2(0.5, 0.0));
    return col;
}
```

Let's add a `float2`, with `0.5` in the `x` axis and `0` in the `y` axis and see what happens.



Wow, that image is strange, right? Well.. that is related to the wrapping mode of the texture. Let's fix it, so it does actually loop. Search for the image, click on it, and, in the inspector, click Wrapping Mode and switch from Clamp to Repeat. Then click Apply.



Now you can see that the image shifted correctly. If we change `0.5` to whatever other value, we'll see that the image position changes.

What we need to do is to modify that over time. And here we can use Unity's `_Time` struct.

```
fixed4 frag (v2f i) : SV_Target {
    fixed4 col = tex2D(_MainTex, i.uv + float2(_Time.y, 0.0));
    return col;
}
```

For this to actually work, we need to put play in the scene and yay! We got it moving!

Half of the thing is done. Now, what can we do in order to modify the speed of this?

The speed by which this moves is related to the `_Time.y` value. What we need to do is to scale that value, and we achieve scaling by multiplying it. So let's add a float value multiplied to

`_Time.y`. For example, let's multiply it by three and see what happens.

```
fixed4 frag (v2f i) : SV_Target {
    fixed4 col = tex2D(_MainTex, i.uv + float2(_Time.y * 3, 0.0));
    return col;
}
```

Wow, it's way too fast. What if we multiply it by `0.5` (which is the same as dividing by two, making it smaller),

```
fixed4 frag (v2f i) : SV_Target {
    fixed4 col = tex2D(_MainTex, i.uv + float2(_Time.y * 0.5, 0.0));
    return col;
}
```

It goes slower. What about zero?

```
fixed4 frag (v2f i) : SV_Target {
    fixed4 col = tex2D(_MainTex, i.uv + float2(_Time.y * 0, 0.0));
    return col;
}
```

As we expect, it stopped.

So, the thing here is easy, if we multiply it by `0`, it stops moving. If we multiply it by something greater than `0` and less than `1` it slows it down, by `1` is the default speed, and more than `1` it speeds it up. We can actually make this speed a parameter of the shader so that we can control it from the editor or the game itself.

Let's add a property called `_Speed` as a `float`, and make it be one. so no change by default.

```
Properties {
    _MainTex ( "Main Texture", 2D ) = "white" {}
    _Speed ("Speed", float) = 1
}
```

Now let's define this `float` in the shader.

```
sampler2D _MainTex;
float _Speed;
```

And remove that constant we were multiplying by and add this speed.

```
fixed4 frag (v2f i) : SV_Target {
    fixed4 col = tex2D(_MainTex, i.uv + float2(_Time.y * _Speed, 0.0));
    return col;
}
```

Now let's go to the material, and modify this variable. As you can see the speed changes.

We can do a small interface tweak here, and use a `Range` instead of a `float`, as the property type, and set between which values we want to allow this to work, in our case going more than `5` or `6` will be too fast to even see the background, so let's add a range between `0` and `5`. So instead of using `float` as the type, we have to put `Range` and two floats that define the range.

```
_Speed ("Speed", Range(0,6)) = 1
```

Check the new visual widget that is now in the inspector. Now you have This is useful if you want to set a fixed speed, but if you want to change the speed while the game runs you'll need to do it in some other way.

This was the first possible solution of the shader. Can you identify why I said there is a slightly more efficient way of implementing it? It's not something that will matter in this shader, but if you have to do more complex calculations it can save you some good amount of time. The problem is that we are modifying the uv in the fragment shader. So for each possible fragment, we're creating two variables (the one we add to `i.uv`, and the sum itself) and doing a multiply.

This could be achieved by doing the same thing, but in the vertex shader, because we're doing a linear operation that can be interpolated since float addition is a linear function. So let's move it to the vertex shader.

```
v2f vert (appdata v) {
    v2f o;
    o.position = UnityObjectToClipPos(v.position);
    o.uv = v.uv + float2(_Time.y * _Speed,0.0);
    return o;
}
```

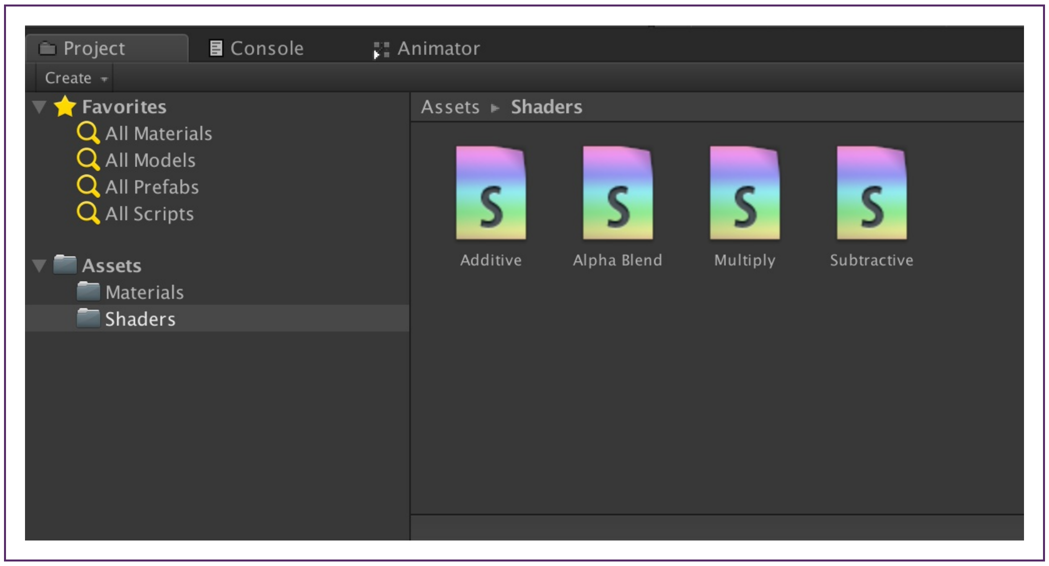
And as you see it works just as well. If you got a little bit lost, you may be wondering why is this faster? The reason is that the vertex shader is processed only once per vertex, in this case, we have just 4 vertices, so we'll do just 4 multiplications and 4 sums, instead of one per pixel.

As I mentioned, a sum and a multiplication are nothing to worry about, it will work super fast either way, but if we were doing something more complex, we could speed up our rendering by only processing this in the vertex shader and let the rasterization step interpolate the values.

I'd like to suggest you go ahead and jump into the forums if you feel lost on the additional exercise, or just to help other fellow students. I'll be around to help there too.

Exercise 2 Solution

This exercise was about implementing four blending modes. Open `Exercise 2 - Blending Modes` and you'll find 4 shaders: `Multiply` shader, then the `Alpha Blend` shader, `Additive` and finally `Subtractive`. There are also 4 corresponding materials with their shaders and textures attached to them.



All of these blending modes have to be implemented using the `Blend` and `BlendOp` commands. You can check the reference manual for this commands to get all the possible options you can use and combine.

Multiply

For the Multiply blend mode, we want to take a look at the default formula for the `Blend` operation. Remember that:

```
Blend SrcFactor DstFactor
```

Represents:

```
Color = SrcColor * SrcFactor + DstColor * DstFactor
```

In this case, what we want to end up since we need multiply blend, is

```
Color = SrcColor * DstColor
```

So, what we need to do is to use `DstColor` as the first parameter of the `Blend` command, and `Zero` as the second one. In that way, the second part of the equation disappears and the first one ends up as we expected.

Alpha Blending

The next mode is Alpha Blending, which is achieved using

```
Blend SrcAlpha OneMinusSrcAlpha
```

Go back to the Alpha Blending chapter if you don’t remember why this is done this way.

Additive

Now we have Additive, which is the easiest one. If you take a look at the equation

```
Color = SrcColor * SrcFactor + DstColor * DstFactor
```

It's pretty much obvious that `SrcFactor` and `DstFactor` have to be one, in order for the colors to be added. So, the solution is

```
Blend One One
```

Subtractive

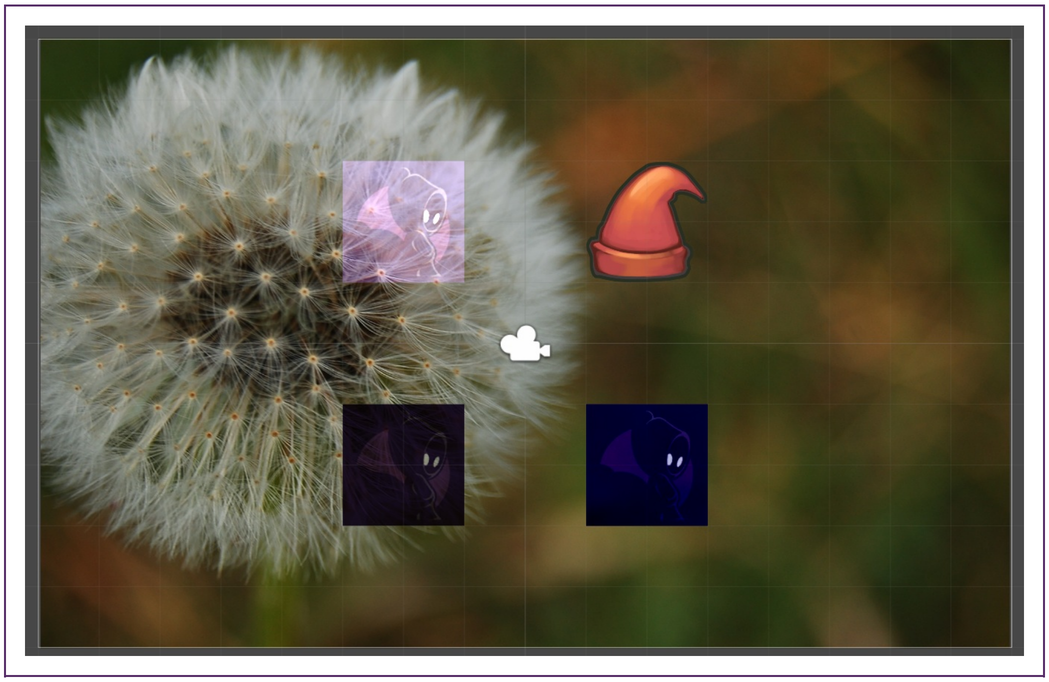
Now for the Subtractive, we need to change the operation in the middle of the Blending equation, the plus between both terms. For that, we have to use the `BlendOp` command. In this case, we want

```
Blend One One
BlendOp Sub
```

which will end up subtracting both colors. Notice that you could use `RevSub` if you want to invert the order of terms in the equation.

Result

In the end, it should look like this:



Appendix I: Linear Interpolation

Mathematically speaking, Linear Interpolation is a method used to define a straight line between two points. It's a simple mathematical function:

```
float lerp(float a, float b, float w){
  return w * a + (1-w) * b;
}
```

Let's analyze it. The method lerp (from _L_inear int_ERP_olation) receives three values, `a` and `b`, in this case, are `float`, but they could be any type that supports addition and multiplication. Then we have `w` (weight), which is a parameter that goes between `0` and `1`.

You can find all the values that are in between `a` and `b`, by passing incremental values of `w`. For example:

```
lerp(10, 20, 0) // returns 10
lerp(10, 20, 1) // returns 20
lerp(10, 20, 0.5) // returns 15
```

There is no more magic to it than this, it's just a matter of internalizing the formula and understanding how it works. When we work on our shaders, the typical parameters are two colors (or texture fetches) for `a` and `b` and one of the components of the `uv` vector for `w`. Also, the `_Time` struct is a popular option for the `w` parameter. But keep in mind that it could be pretty much any value, or array (`float2`, `float3`, `float4`) that can be used. In fact, in Cg, you can even pass one of these values to the `w` parameter for different uses.

In the rasterization stage, interpolators are used to go from one vector to another, step by step, finding out which pixels are covered by a given triangle. Because of this, we have a fast hardware implementation of linear interpolation that we can use. This is exactly what we do when we set the uvs in the vector shader and receive it interpolated in the fragment shader. We're relying on the hardware-accelerated interpolation that comes bundled in the GPU. Whenever you feel like you can interpolate in the vector shader instead of the fragment shader, go ahead and do it, it will save you valuable computing time.

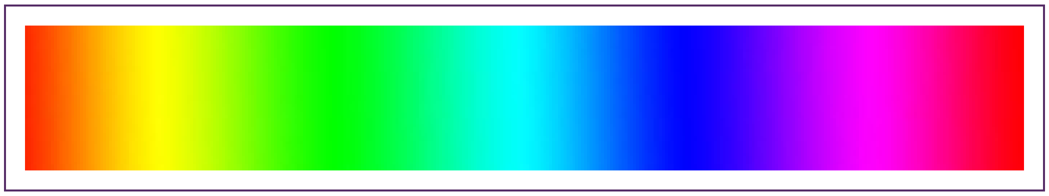
If you want a more in-depth (aka. Mathematical) explanation of Linear Interpolation, be sure to check its Wikipedia entry.

Appendix II: HSV Color Space

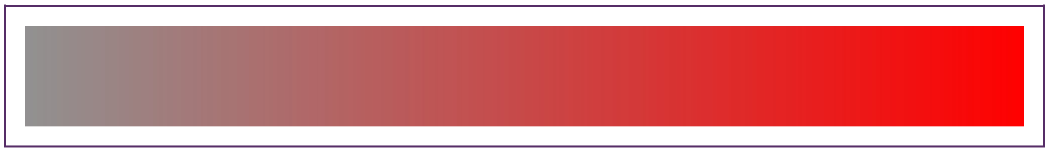
HSV is an alternative representation of the RGB color space. It stands for Hue, Saturation, Value; and it's a useful way to represent colors closer to how we perceive them. What do I mean by that? How easy it is for you to give me a halfway desaturated red color using RGB. Not easy. The reason is that there is not a clear relationship between how we perceive colors and these three channels.

Instead of smashing our heads adding more green, then blue, removing some red, just to find that the result is not the kind of color we need, we can rely on this alternative representation. Let's analyze how it works.

Hue represents the color itself. It's a linear gradient that traverses the whole color spectrum from `0` to `1`.



Saturation is the amount of gray in the color. It also goes from `0` to `1. 0`. `0` is full gray, `1` is no gray at all (full color).



Value (or Brightness) represents how bright the color is. It also goes from `0` to `1` and `0` is full dark, and `1` full bright. Keep in mind that `1` varies according to the saturation value.

With these three values, it's easier to define a color like the one I mentioned before: a halfway desaturated red color could be represented by something like `(0, 0.5, 1)` in HSV. Pretty simple, right? Now, we know that the GPU expects RGB, so we should be able to go from HSV to RGB. Here are the two methods you'd need to paste in your shader if you want to use HSV inside of it (Migrated to Cg from https://www.laurivan.com/rgb-to-hsv-to-rgb-for-shaders/):

```
fixed3 rgb2hsv(fixed3 c) {
    float4 K = float4(0.0, -1.0 / 3.0, 2.0 / 3.0, -1.0);
    float4 p = lerp(float4(c.bg, K.wz), float4(c.gb, K.xy), step(c.b, c.g));
    float4 q = lerp(float4(p.xyw, c.r), float4(c.r, p.yzx), step(p.x, c.r));
    float d = q.x - min(q.w, q.y);
    float e = 1.0e-10;
    return fixed3(abs(q.z + (q.w - q.y) / (6.0 * d + e)), d / (q.x + e), q.x);
}

fixed3 hsv2rgb(fixed3 c) {
    float4 K = float4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    float3 p = abs(frac(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * lerp(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
```

And here how to use it:

```
fixed4 frag (v2f i) : SV_Target {
    fixed3 c_rgb = fixed3(0,1,0); // Green
    fixed3 c_hsv = rgb2hsv(c_rgb); // Green in HSV
    c_hsv.x = 0; // Move Hue to Red
    c_rgb = hsv2rgb(c_hsv); // Red in RGB
    return float4(c_rgb, 1); // Return Red
}
```

Acknowledgements

This book is the result of a long journey that included a lot of people, and I'll do my best to include them here, but I may leave some people out. Sorry if I did.

First of all, I want to thank my girlfriend and eternal partner Aldi, who's always there for me no matter what crazy idea I have plans for. Thanks for all the support, I love you so much. Thanks to the rest of my family too, my mom Elena, my siblings and their couples who also helped in different ways. Jorge and Gaby for always being there for us too.

I want to thank everyone involved in the game development community from Argentina, especially my great friends from Nastycloud and Bigfoot Gaming.

Also David Roguin, Agustin Cordes from Senscape and Daniel Benmergui, all the crew from ADVA who put countless amounts of time in growing our local industry.

Thanks to my dear friends Michael de la Maza, Diane Hsiung and Julian Nadel for being of immense help and support during so many years.

Last but not least, all the amazing developers that helped review the book in its early stages: David Roguin, Mauricio J Perez, Gaston Simonetti, Dan Amador, Juan Sebastián Muguruza, Nahuel and Orlando Almario.

Thank you so much to all of you, I'm eternally grateful for your time investment in making this book better!

Credits

The amazing cover design and Hidden People Club logo were created by German Sanchez from Bigfoot Gaming. I can't be more grateful for having you on board, man.

The City Background for the scroller exercise downloaded from OpenGameArt and created by Alucard

Nubarron Icon (hat in the Blending Modes exercise) created by Juan Novelletto

Dandelion Fruit by Aldana Gonzalez