Unity for Toddlers

JANUARY 03, 2021

Learning Objectives: Visual Effects

- Adding visual effects with **URP**

- Basics of 2D lights
- Creating and using **shader graphs** for 2D rendering: glow and outline,
- Adding **post processing effects**: Bloom filter
- **Parallax background** effect: using Texture scrolling and secondary Cameras, setting culling mask
- Basics of **particle systems**: textured sheet particles
- Creating breakable prefab

Introduction

By now, you should be more or less equipped to implement the basic game logic (at least it will work), for example create platforms, create enemies, create consumable power-ups, count scores, and restarting the game. The next important element in a game is feedback: both visual and auditory feedback. To prove this point, download the asset in the course handout and import it to your Mario project or a new project (your choice). We will be working with this side project for awhile first before returning to our Mario project.

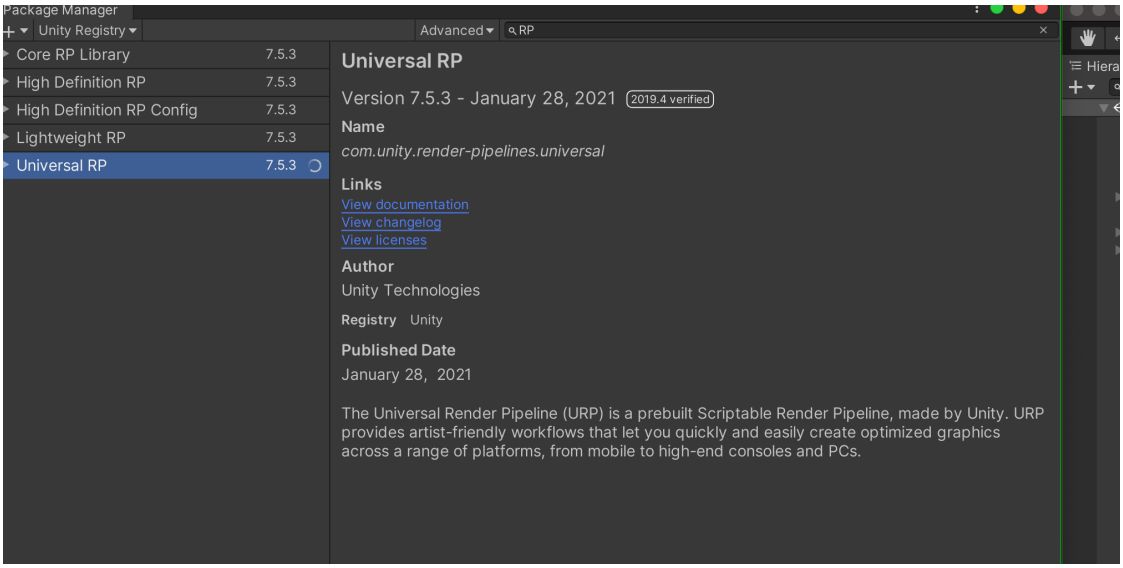
> The complete free asset can be downloaded from GothicVania Church Pack from Unity asset score. The content of this section is obtained from the wonderful Brackeys tutorial.

Universal Render Pipeline

URP package allows us to easily create optimized graphics. We need to enable it first in our project before we get started.

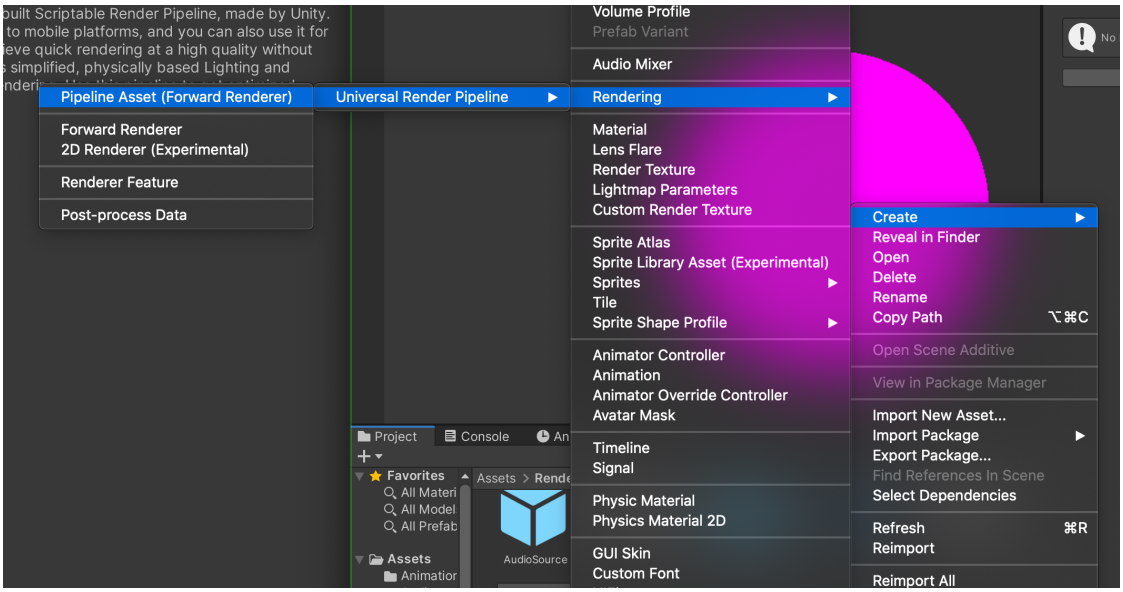
Download URP Package

Go to Window » Package manager, and download Universal RP from Unity Registry as shown.



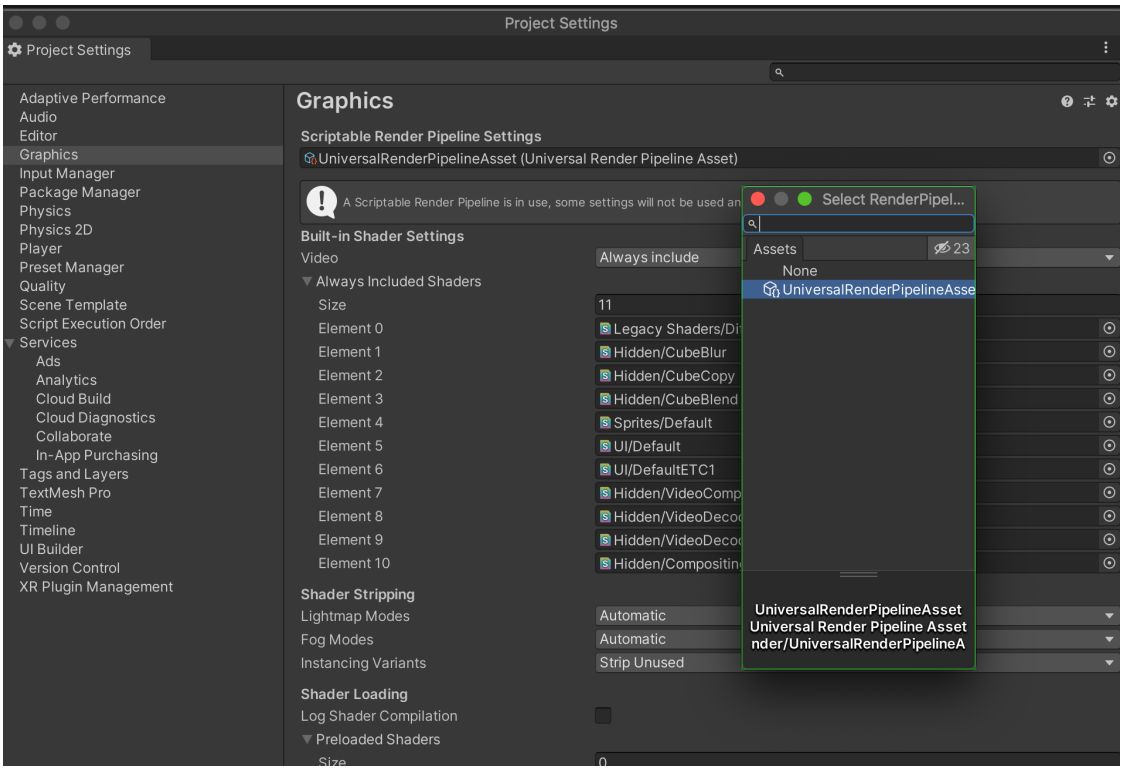
Create Pipeline Asset

This will add the URP package to your project. Afterwards, create a folder in the Assets folder called “Rendering”. Inside it, create Rendering » Universal Render Pipeline » Pipeline Asset (Forward Renderer).



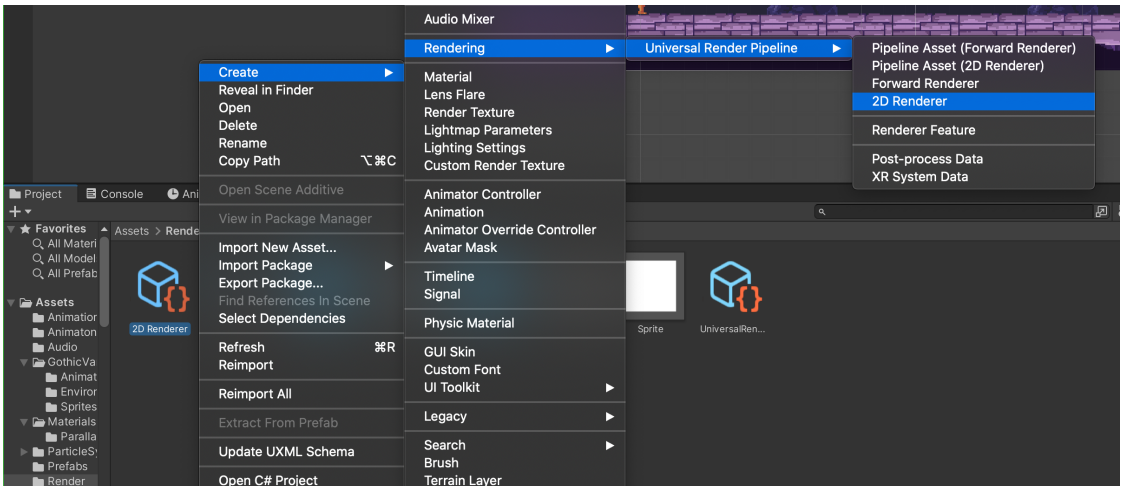
Modify Project Graphics Setting

Go to Edit » Project Settings » Graphics and select the UniversalRenderPipelineAsset you have created in the previous set as shown:

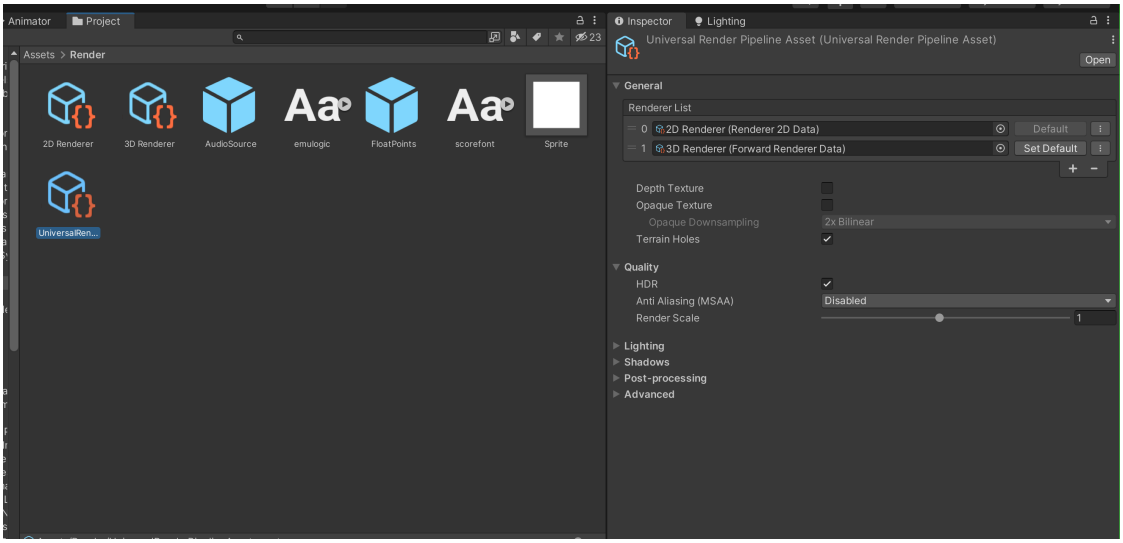


Create 2D renderer

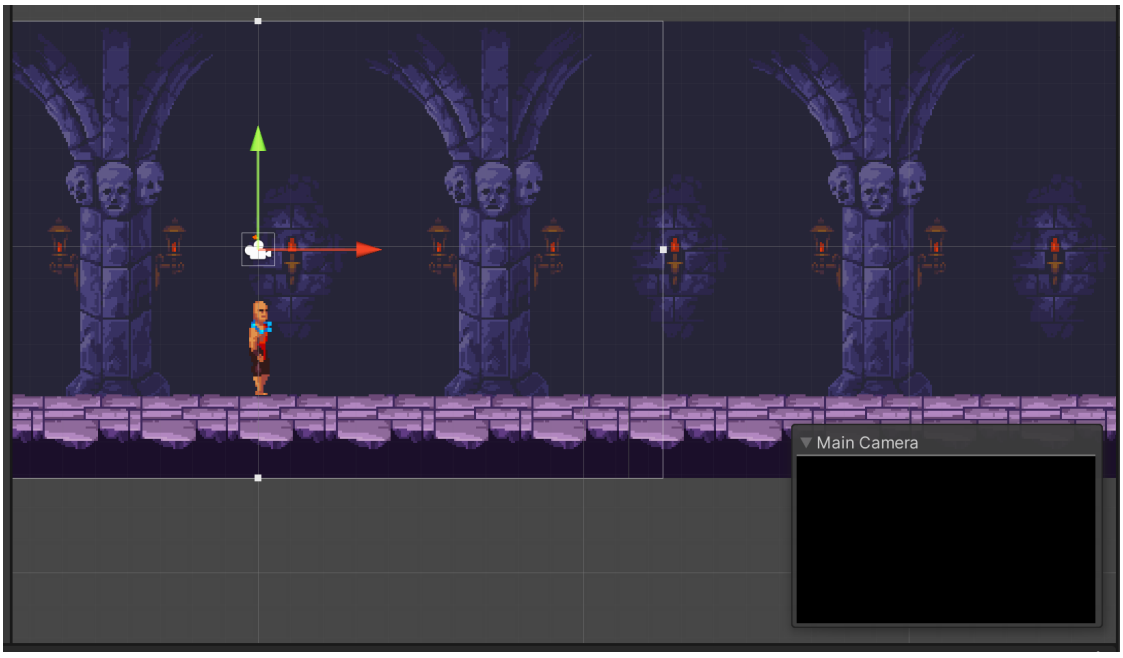
Since we are working with 2D games now, we need to create a 2D renderer as our Universal Render Pipeline Asset's Renderer. Right click inside the Render folder and Create » Rendering » Universal Render Pipeline » 2D Renderer as shown:



Click on the UniversalRenderPipelineAsset you created before and load the 2D Renderer you just created under its Renderer List in the inspector and we are **done** with setting up URP for your project.



Open Fighter.scene and notice that everything in the Game window is dark, although you can clearly see in the Scene that there are some stuffs in the world as shown below. If you can't see anything on the Scene either, toggle the light bulb icon to *disable Scene lighting* (for our view when prototyping, but not the camera's).



Time to add some Lights, Camera, Action!

Adding 2D Lights

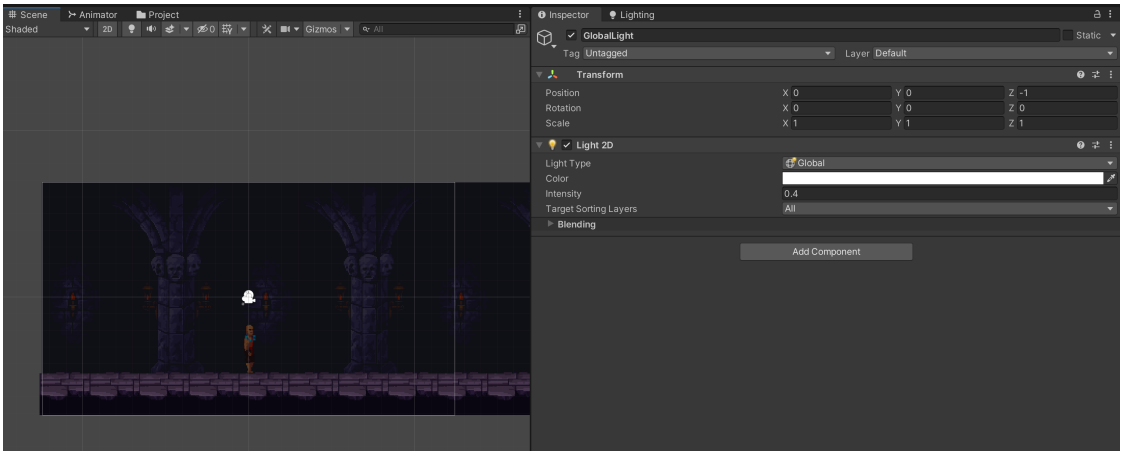
The fun thing about enabling URP is that you can now lit up your sprites, thus creating a bit of a *mood*. Right click on your project hierarchy and observe that there are a few 2D Lights GameObjects that you can create, namely FreeformLight2D, SpriteLight2D, GlobalLight2D, and SpotLight2D.

GlobalLight2D

Create a GlobalLight2D Game Object, and set the properties:

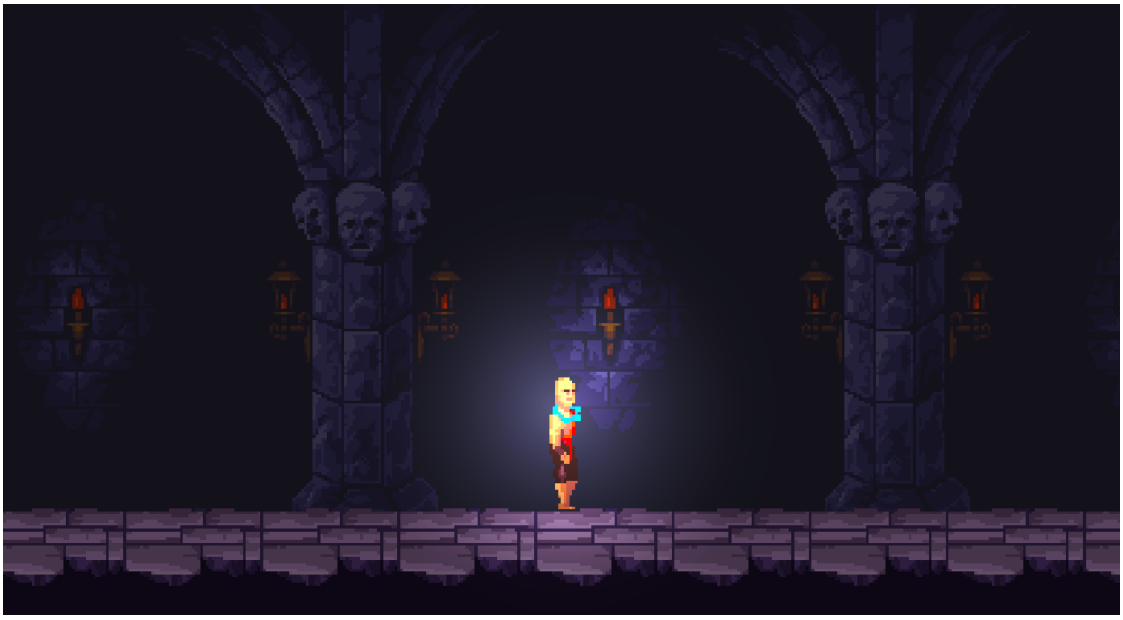
- Intensity of 0.4
- Target Sorting Layers: All (this determines which Sprite layers to lit up)
- Position at `(0, 0, -1)`

Toggle enable lighting at your Scene view and you should observe a somewhat dimly lit environment like this:



SpotLight2D

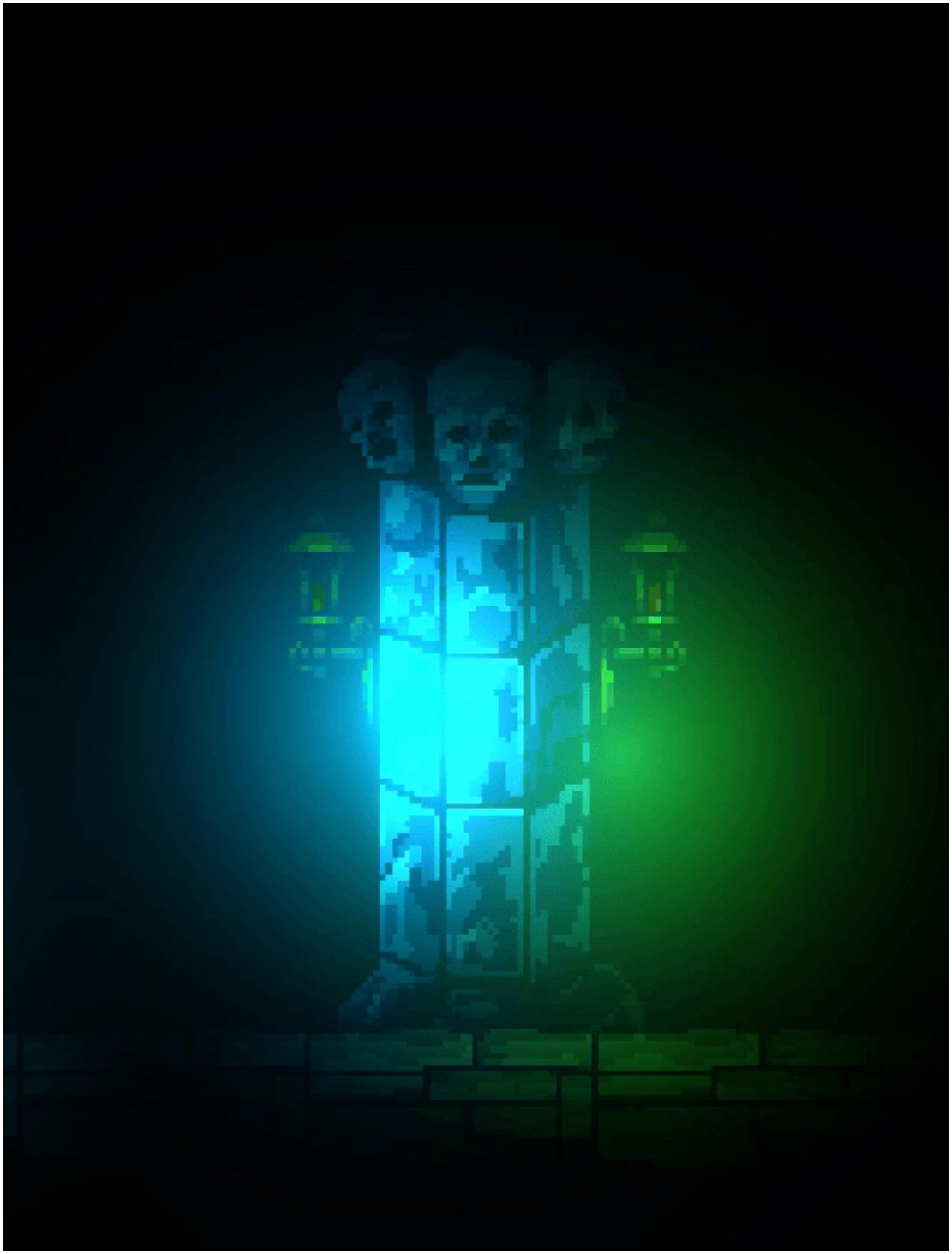
Another type of light that you can add to the game os SpotLight2D. Create a gameObject of SpotLight2D type and place it as a child gameObject of the player. Place it somewhere near the Player's head to brighten up that area. You should see something like this in your Scene:



Head over to the Light2D inspector and you observe that you can adjust its properties such as light **color**, **intensity**, **radius**, and **target sorting layers** (yes, you can ask light to just lit up certain areas and not the others).

Expand the **blending** section and observe a few settings: this dictates how two or more light source should blend when they're near one another. Overlap operation of **additive** will cause the overlapping region to be very bright, while **alpha blend** allows the light with higher order to be rendered on top of the other. The gif below demonstrates at first the blue light to be rendered after the green light, giving an overall blue-ish appearance. Afterwards, the light order is swapped so you see that the overall overlapping region has a green-ish appearance. Both lights are set to be "Alpha Blend".

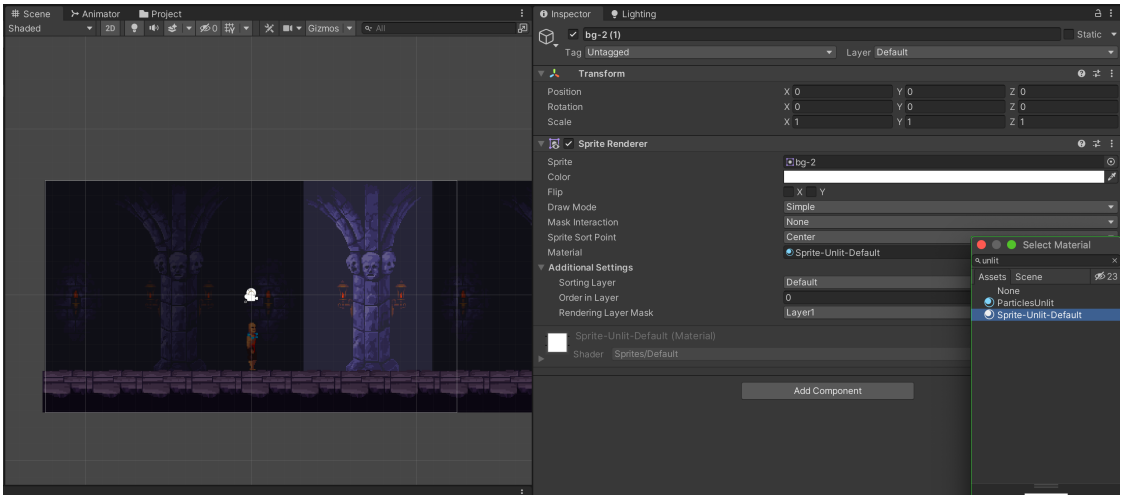
> Wait for a few seconds, the gif is quite long.



URP SpriteLit and SpriteUnlit Material

The reason that Unity knows which Sprite should be affected by lighting (or not) is because of the material of the Sprite. To prove this point, expand the Environment GameObjects in the Hierarchy, and click on any background object. As an example, let's use `bg-2(1)` .

Then in the inspector of `bg-2(1)`, change its Material into **Sprite-Unlit-Default**. You can see that part of the background corresponding to `bg-2(1)` isn't affected by the global light, and will just show its regular texture:



The other background objects are affected by light because its material is set to **Sprite-Lit-Default**. Both **Sprite-Unlit-Default** and **Sprite-Lit-Default** are **Materials** that utilizes URP's **Shaders**.

Textures, Shaders, and Materials

Materials, Shaders, and Textures are three different things. We need the first two to render the Camera's view, and sometimes Textures as well if we do not want the object to simply contain solid colors.

When we import images (.png, .jpeg, etc), we are creating **Textures**, also known as *bitmap images*. Textures alone are not enough to dictate how Sprites or 3D Meshes should be rendered.

We need **materials** for this. **Materials** are definitions of *how* a surface should be rendered, including references to textures used, tiling information, colour tints, *normal maps*, mask, and more.

On the other hand, **Shaders** are small scripts that contain the **mathematical calculations** and algorithms for **calculating the colour of each pixel** rendered,

based on the lighting input and the Material configuration. Therefore, it should be obvious that when we create materials, we need to specify the **Shader**.

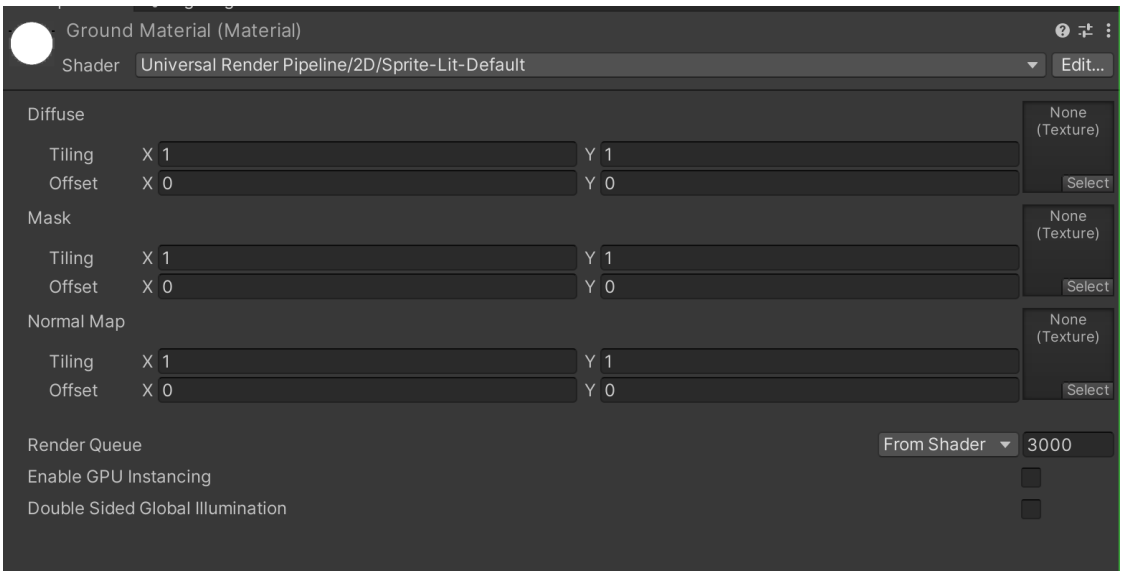
Let's create a *new Material* and demonstrate this knowledge.

- Under Materials folder in your Assets, right click Create » Material and name it GroundMaterial.
- Set its shader to be Universal Render Pipeline/2D/Sprite-Lit-Default. Then, click on GameObject `tileset_15` and then,
- Load this GroundMaterial under the Sprite Renderer component of `tileset_15` : the Material property.

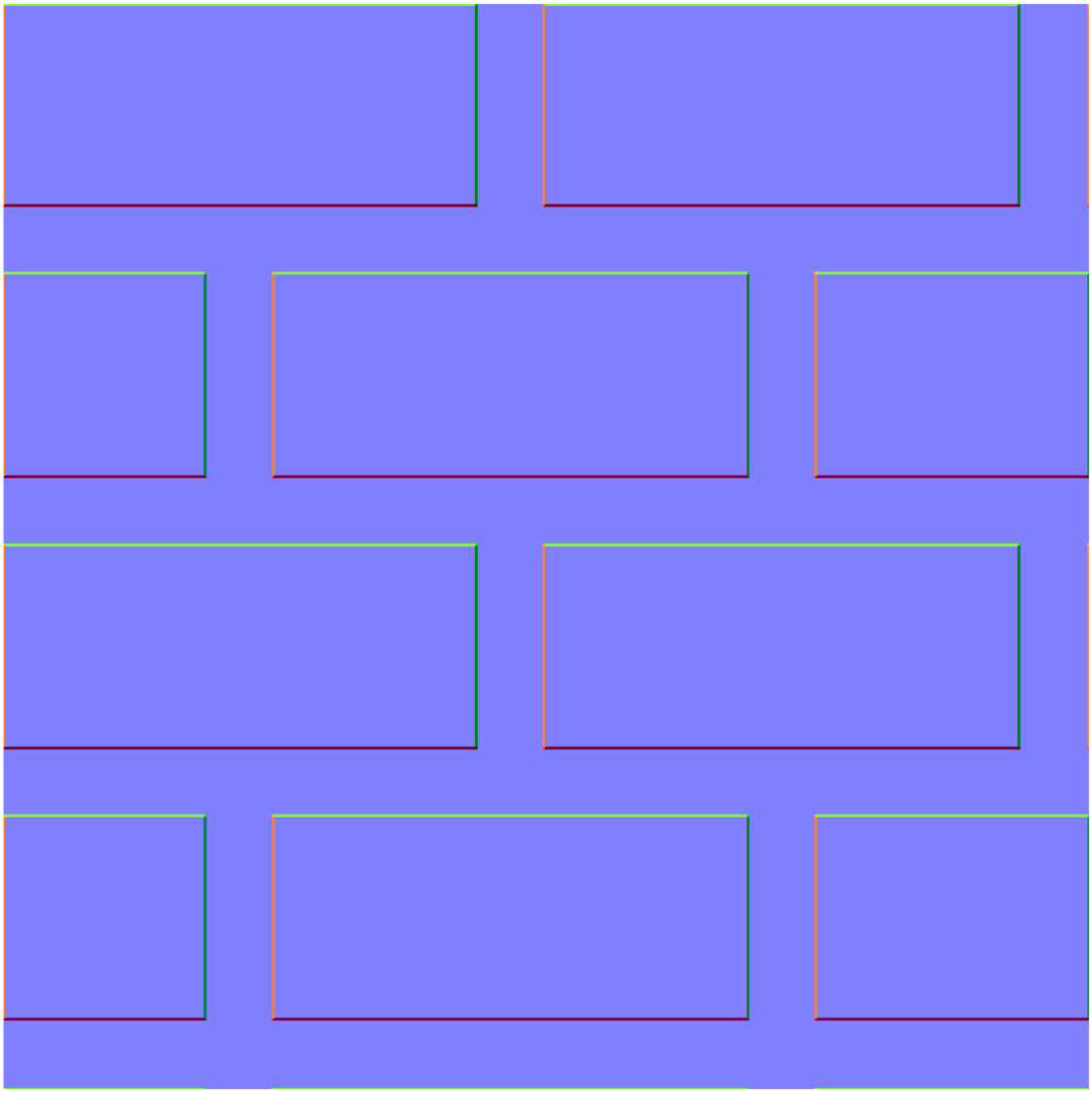
You will see that since GroundMaterial utilises the same shader as Sprite-Lit-Default material that comes with URP package, **there's no change in how the ground tiles look**.

Normal Mapping

When creating GroundMaterial, you might've seen a few properties that you can set, namely Normal Map, Mask, and Diffuse. We will briefly touch about Normal Map here.



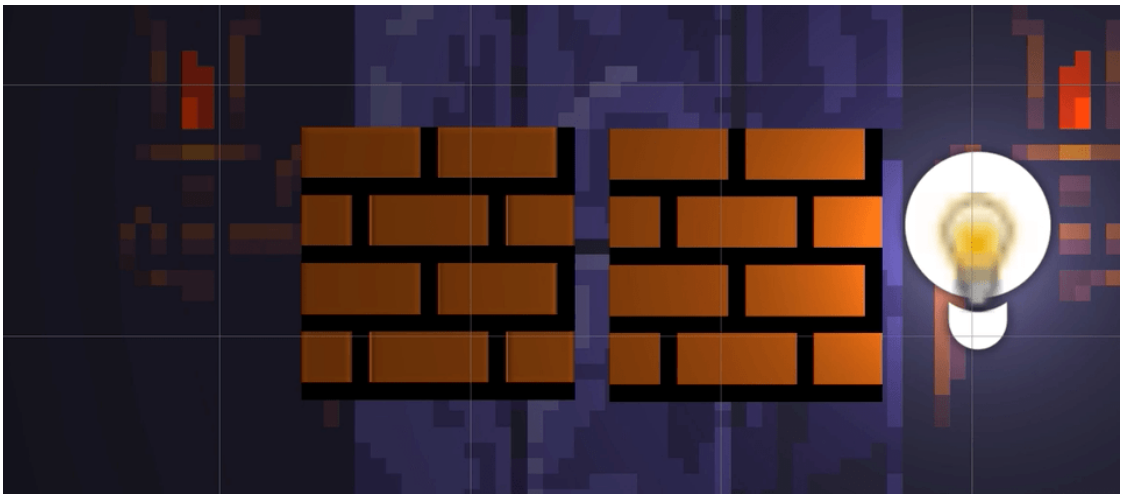
In Computer Graphics, normal mapping is a texture mapping technique used for faking the lighting of **bumps and dents**. A normal map contains normal vector values at each point of the texture, encoded into the RGB color, so you will commonly see them in blue-ish hue as such:



The blue hue comes from the fact that the majority of the surface is flat, and therefore the normal vector is {0,0,1} (full Z-direction, where Z-direction with respect to the local frame of the surface refers to the perpendicular vector (the surface itself is considered to be on X-Y plane *local* to itself). Mapping this into RGB: {0,0,1} results in the color blue.

Normal Mapping Example

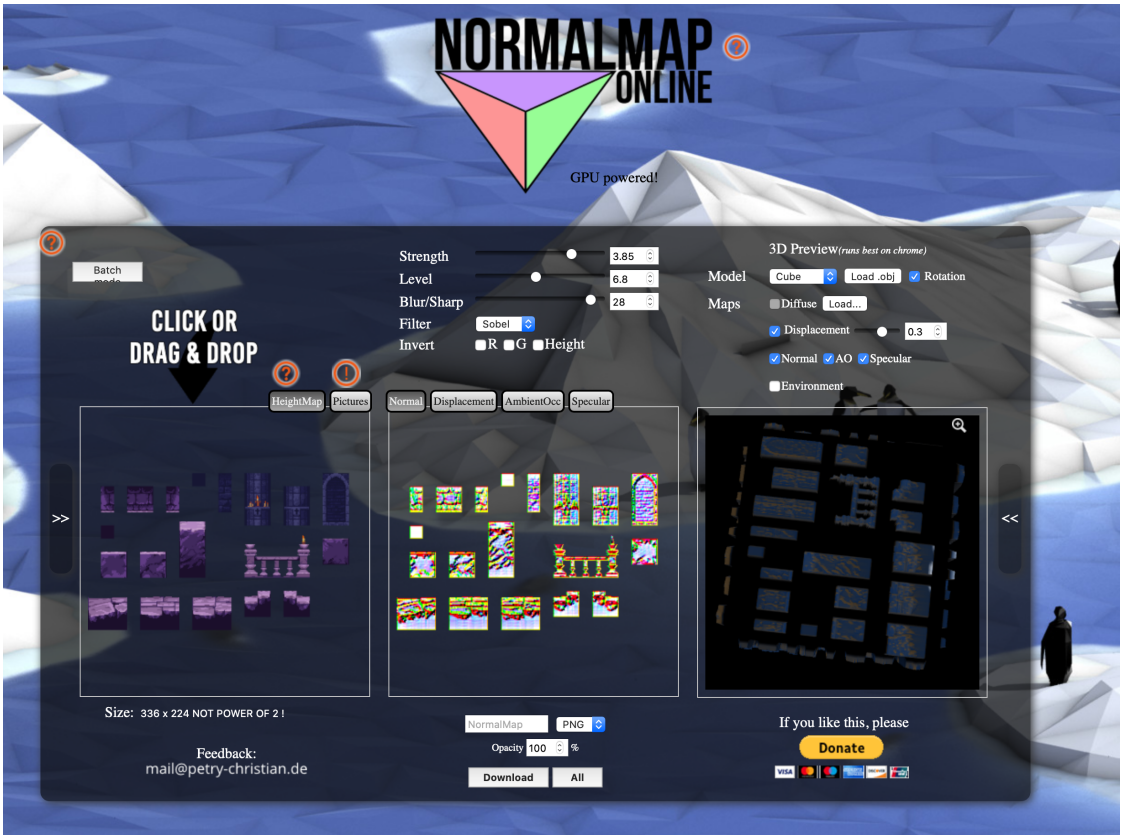
Take for example this Mario brick + normal map applied (left) vs regular Mario brick (right). Two kinds of normal maps were applied: the one that matches the brick surface (the normal map above), and another is a rough rock normal map (so it's more obvious).



Creating Normal Maps

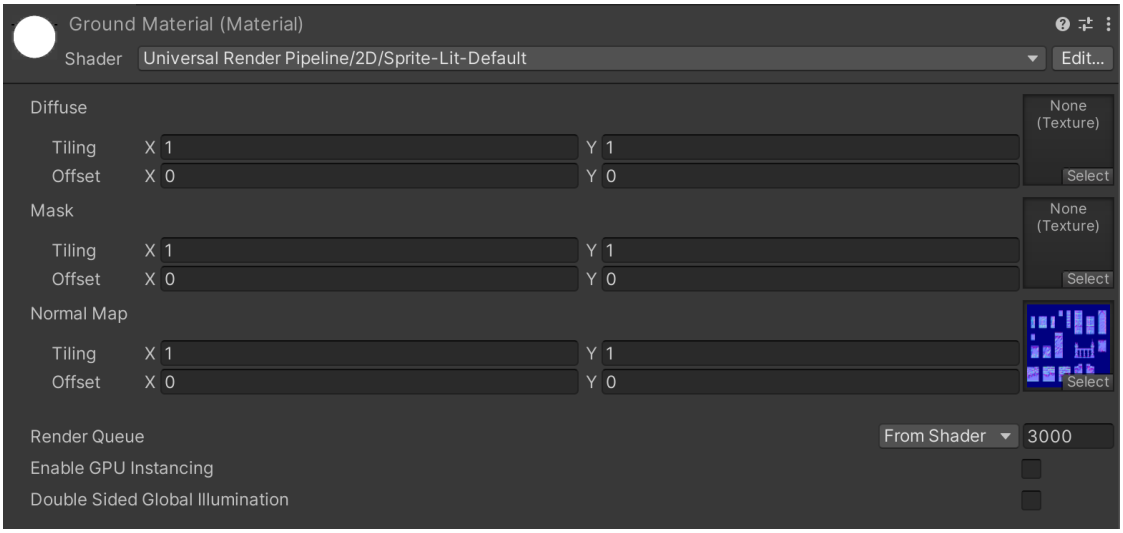
Now let's apply normal map to our floor. Our floor looks flat currently, and we need to generate normal maps from the floor's tileset (can be found under GothicVania Church folder » Environment).

Let's use an online tool to generate the normal map. This online tool: https://cpetry.github.io/NormalMap-Online/ is absolutely awesome (otherwise you can use Photoshop to generate normal map from a given texture too). Load the tileset image and you should be able to view and download a normal map for it as such:



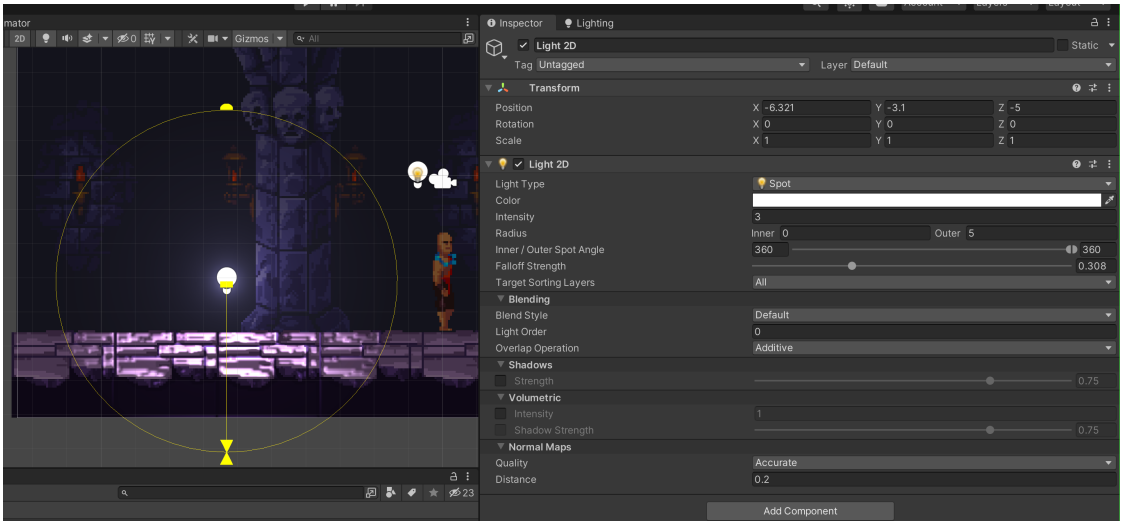
Store the tileset normal map as an asset inside your Unity Project, and **set its Texture Type to be "Normal map"**. There's two ways to apply this normal map to our floor. We can apply it directly on the tileset Texture as a secondary texture, OR create a new material and specifically load this normal map into the shader. The latter is simpler, so we will use that.

Open the GroundMaterial you have created earlier, and **load** the tileset normal map under the **Normal Map property**:



Load `GroundMaterial` as the Sprite Renderer's Material property on `tileset_15` GameObject.

To view the Normal Map, we need a light source. Create a new **SpotLight2D** with **Normal Maps** property enabled, and set its distance into something small as shown:



Move the light around and you should be able to see the bumpy floor with normal map applied on it.

Shaders

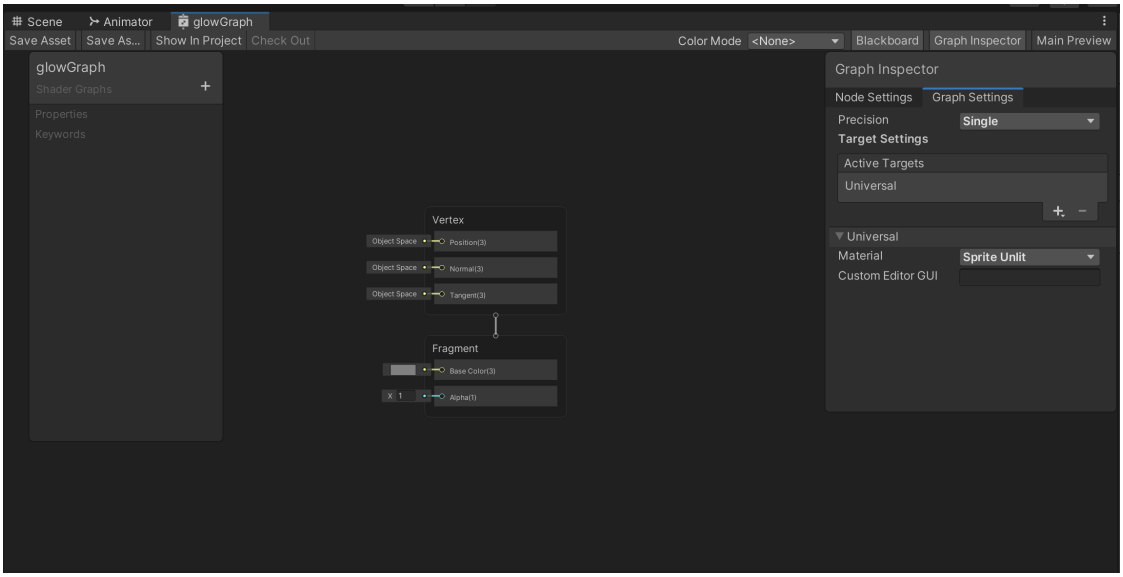
Creating your own shader is no easy feat. It is one of the hardest thing to do in fact, but ShaderGraph in URP made it manageable. In this section, we are going to make **custom shaders** (and post-processing effect) so that we can convert our character from looking like this:

To this:

Creating a Basic Shader Graph

Under Assets, create a folder called Shader. Inside it, right click Create » Shader » Universal Render Pipeline » Sprite Unlit Shader Graph.

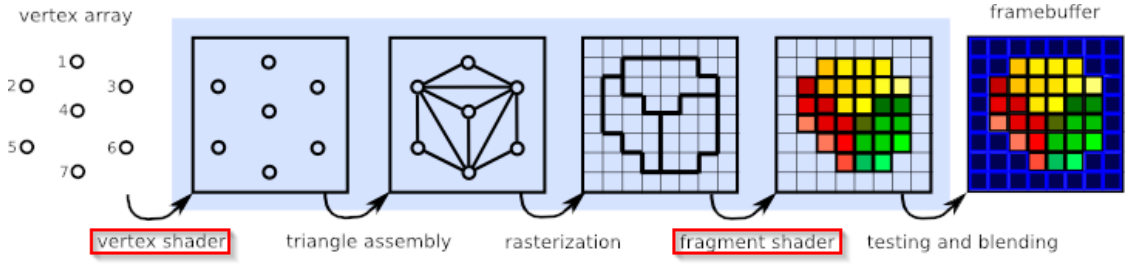
Name it: `glowGraph` . Double click the graph and you should be presented with the following graph editor UI:



The way we encode logic to the Shader is via visual scripting. Right now we only have two **nodes**, called **Vertex** and **Fragment**. The Shader that we are about to create will dictate how we should color a particular *fragment* (think of it as a single pixel) of our gameObject in our Scene (if any).

Vertex Shader and Fragment Shader

The figure below illustrates a simplistic summary of graphics pipeline:



In graphics pipeline, there are *two* types of shaders that are commonly customisable (well, actually *three*. The other one being geometry shader but this is not 50.017 so we are going to spare you from that), called v**ertex shader** and **fragment shader**.

Vertex Shader: applied very early in the graphics pipeline, and its job is to **transform** each **vertex's** 3D position in *virtual space* (world space) to the **2D coordinate** at which it appears on the screen (as well as a **depth** value for the Z-buffer; so that the GPU knows what ought to be rendered in front of what). **Vertex**

shaders can manipulate properties such as position, color and texture coordinates, but *cannot* create new **vertices**. A vertex is normally composed of *several* **attributes** (positions, normals, texture coordinates etc.).

Fragment Shader: applied later in the graphics pipeline, and it defines **RGBA** (red, green, blue, alpha) colors for each **pixel** being processed. It can be programmed to operate on a per pixel basis and takes into account *lighting* and *normal* mapping.

You can learn all about Shaders glory using various 2D and 3D rendering API, for example using OpenGL here, or using Vulkan here, or using Metal here. Writing shaders from scratch is one of the hardest thing to do in life, so we gotta be really grateful with the existence of visual scripting tools like URP which allow us to write shaders without having to be bothered with shading language syntaxes.

> But this doesn't mean that you can forget your linear algebra and programming knowledge, or basic logic.

Let's say we want to shade a simple solid color on the object's **fragment**, e.g: **red** color everywhere, not affected by lighting or anything. In GLSL (that's OpenGL's shading language), we need to specify the vertex shader as follows:

```
#version 330 core
layout (location = 0) in  vec3 aPos; // the position of the vertex is at position 0
out  vec4 vertexColor; // specify a color output to the fragment shader

void main()
{
        gl_Position = vec4(aPos, 1.0); // see how we directly give a vec3 to vec4's
        vertexColor = vec4(1.0, 0.0, 0.0, 1.0); // set the output variable to a pure
}
```

and the fragment shader as follows:

```
#version 330 core
out  vec4 FragColor;
in  vec4 vertexColor; // the input variable from the vertex shader (same name and sa
void main() {
        FragColor = vertexColor; // set fragment color to be equivalent to vertex co
}
```

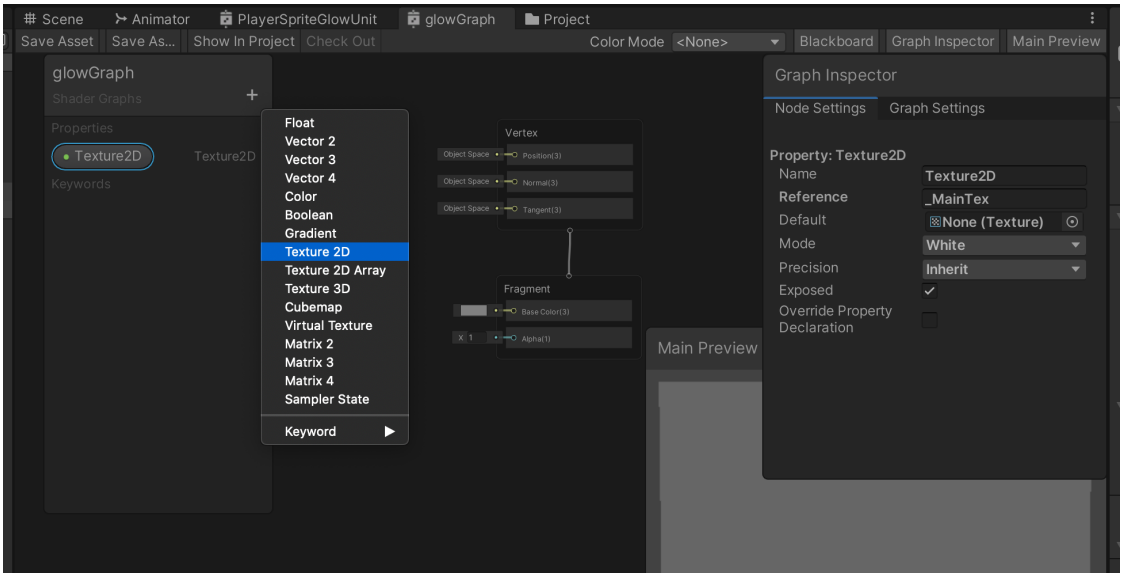
GLSL seems daunting at first, like a nightmarish distant cousin of C programming language. We can create the same logic (that is to shade every pixel where the object in question is in red in this example) using shader graph easily without caring about which part should be the vertex shader or fragment shader.

Using ShaderGraph in URP

Open glowGraph (double click on it). We will create a **material** later using this custom shader glowGraph, and load it as the **Material property** of a gameObject's **Sprite Renderer** so we can observe the output.

Texture2D property with _MainTex Reference

Firstly, to use it with Sprite Renderer, we need to create a **Texture2D property** (whatever you create on the left hand side, the "Blackboard" panel will be the Shader's property) as shown, naming its **reference** (*Reference* field, **not** *Name* field in the Graph Inspector) as `_MainTex` **exactly** as it is required by SpriteRenderer2D in Unity:



> The reason we need `_MainTex` property is because a Sprite Renderer *by default* uses the texture supplied in the Sprite property but uses the Shader and other properties from the Material property. We can use information of the texture (color) in the Sprite property (the `_MainTex`) later on for computations in the Shader to perform more complex computation for the color of each fragment; although in this case we are completely ignoring the color of `_MainTex` since we overwrite everything to be red.

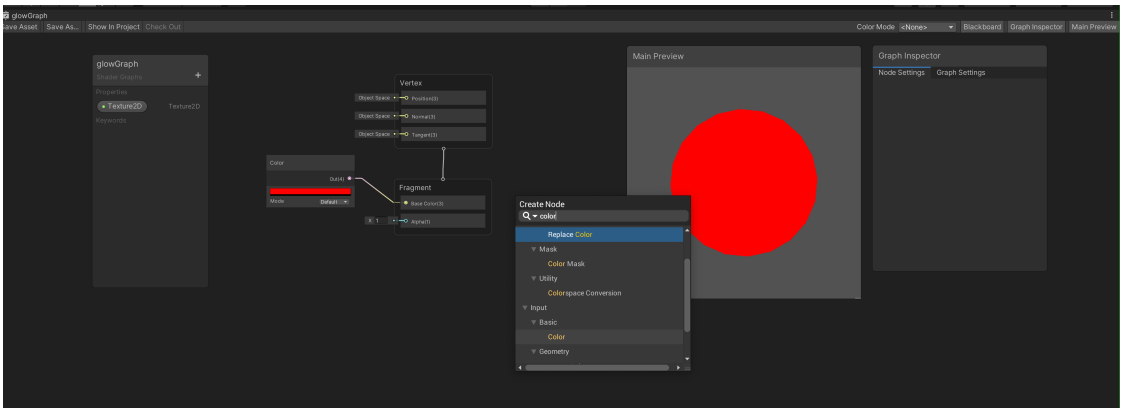
Adding Nodes

We can create nodes to form a shader graph, where each node represents a *logic*. You can right click anywhere at the empty space in the glowGraph window to create new nodes. There's various nodes for all sorts of logic: vector manipulation (dot product, cross product, etc), time stepper, noise generator, sin graphs, etc.

To shade a simple red color on each pixel representing the gameObject, do the following:

1. Create a **Color node**. Right click anywhere at the graph window » Create Node, type "Color", then select Basic » Color.
2. Change the Color node's value to red color (click on the color bar and adjust)

3. Link the output of the node to the input Base Color of the Fragment node as shown (click and drag):



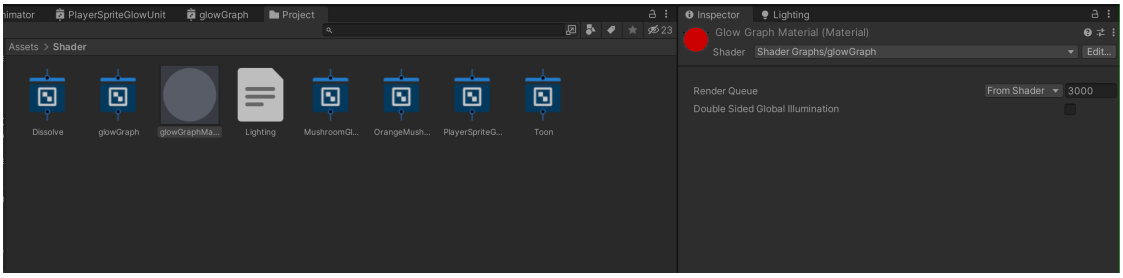
Click on the **Main Preview toggle** on the top right hand corner of the graph editor to see what the output should look like, and then press **Save Asset** on the top left hand corner.

As said before, we need to create a **Material** (name it glowMaterial for example) using this glowGraph shader and then we need to apply it on a gameObject so that we can see the effect of our Shader. In your project folder, highlight `glowGraph` and right click on it, then select Material. This automatically creates a material utilising your custom shader glowGraph.

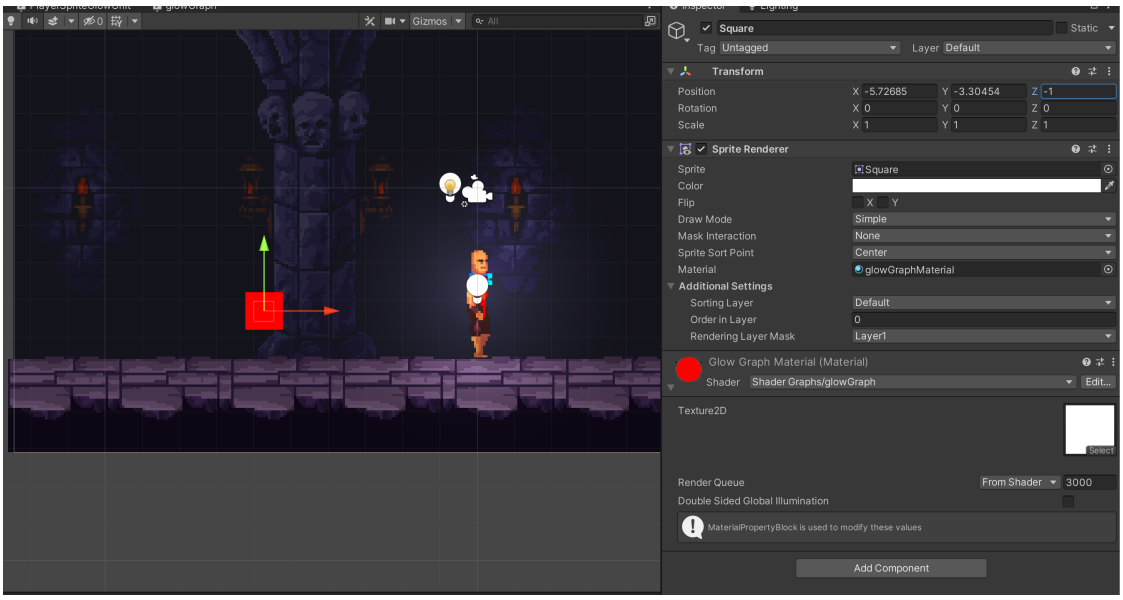
> Alternatively, you can create a generic Material and change the shader in the inspector to be Shader Graphs/glowGraph.

Rename this material into glowGraphMaterial as shown in the screenshot below:

> Ignore the other shaders. We haven't reached there yet.



Now create any 2D sprite » Square gameObject and place it on the scene. Load glowGraphMaterial onto the Material property of its Sprite Renderer, and you should see a bright square box, unaffected by lighting:



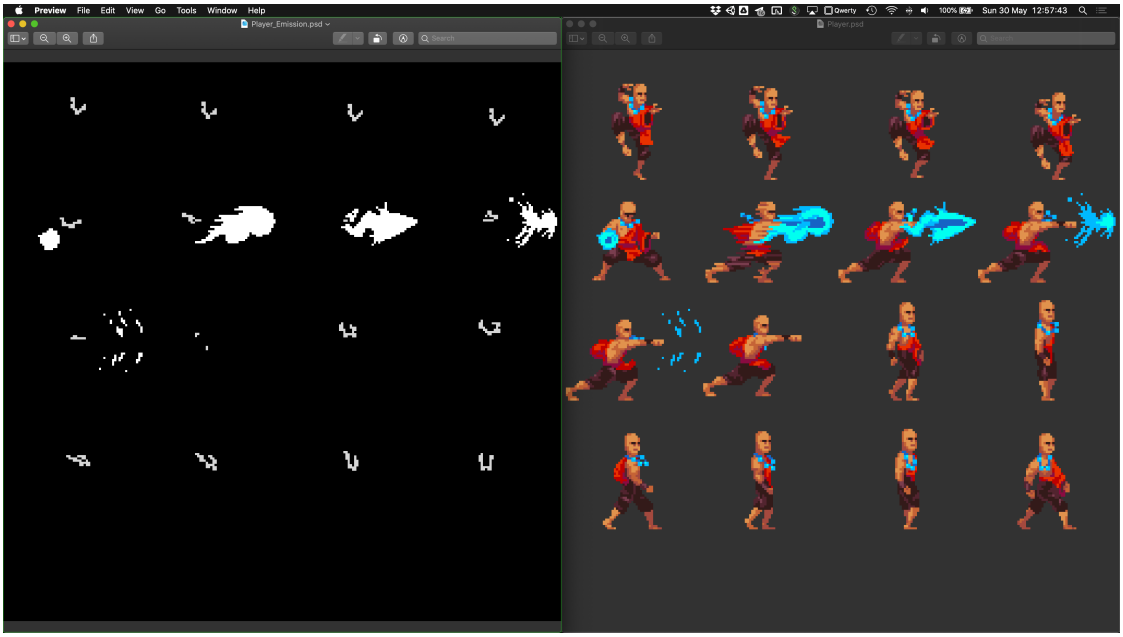
> If you don't see it, make sure its position property in Transform has negative Z value, so that it is "in front" of the background which Z position is set at 0. Alternatively, you can also change the Sorting Layer property of the sprite renderer.

Creating an *Actual* Shader Graph

Since our shader graph name is `glowGraph`, let's modify it further so that it lives up to its name. In order to allow our character to ***glow***, we need to:

1. Brighten all blue parts (the necklace and the "water" looking skill effect) on the character sprite.
2. Add post-processing Bloom filter (later, not part of our shader).

We will do (1) in this `glowGraph` shader. In order to brighten all the blue parts of the Player sprite, we need to create a matching sprite to the Player sprite where we have exclusively the traces of the emission, as shown:



To create the sprites on the left, you can easily use photoshop and trace each blue pixels, replacing it with white. What's important is that the two sprites are of the **same dimension**.

> For now, both sprites are given to you, simply search the name Player and Player_Emission in your Assets folder.

The idea is that we want to:

1. **Load** both as textures to the Shader
2. **Multiply** the emission map with some color, e.g: teal
3. **Add** the output of (2) to the original Player Sprite, so we have this:

ShaderGraph Properties

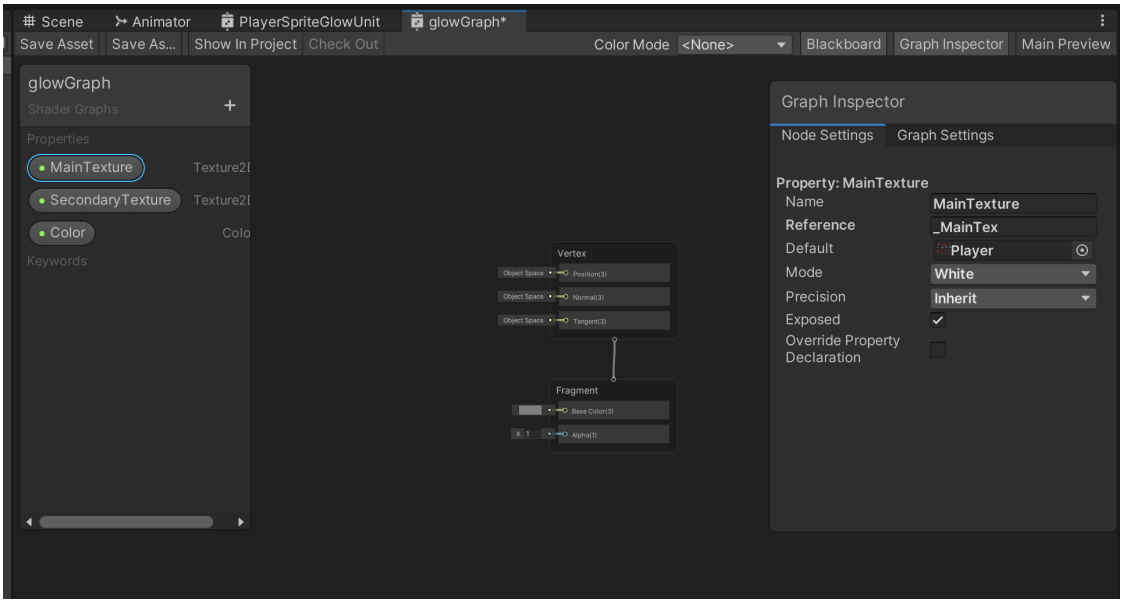
Open glowGraph and add **two** more properties (on top of the MainTex you have created earlier with reference _MainTex):

- Another Texture2D named SecondaryTexture
- Color

Click on **MainTexture** property, and click on the Graph Inspector toggle to view its Node Settings. Make sure you enter the Player sprite set as its Default property.

Click on the **SecondaryTexture** property, and load Player_Emission sprite as its Default property. Set its reference to be anything you want, for example: _SecondaryTexture.

Finally, click on **Color** property and set the Default property color value to be something teal. Here's a screenshot to clarify what we mean by changing the Node Setting:

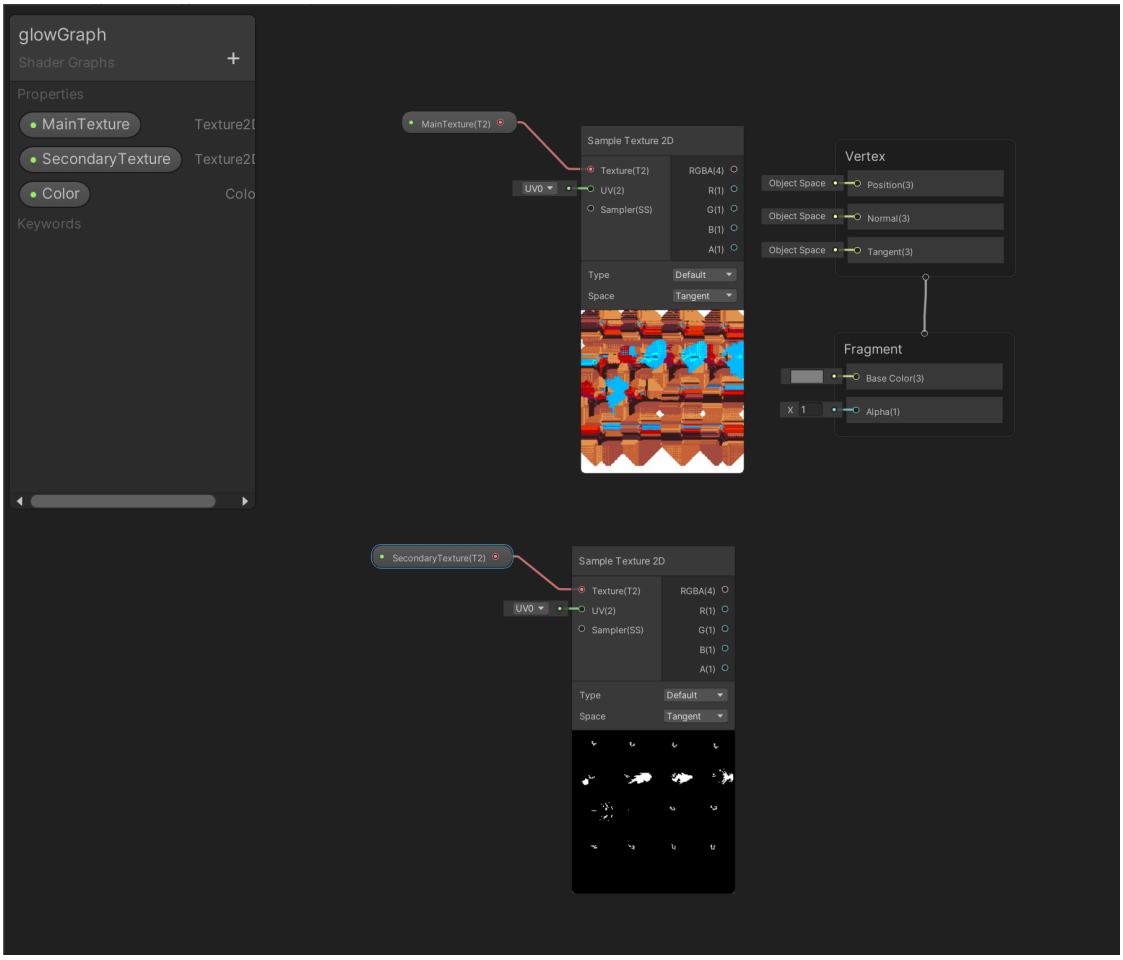


To use these three properties as part of the Shader's computation, we need to create Shader nodes.

Sample Texture 2D

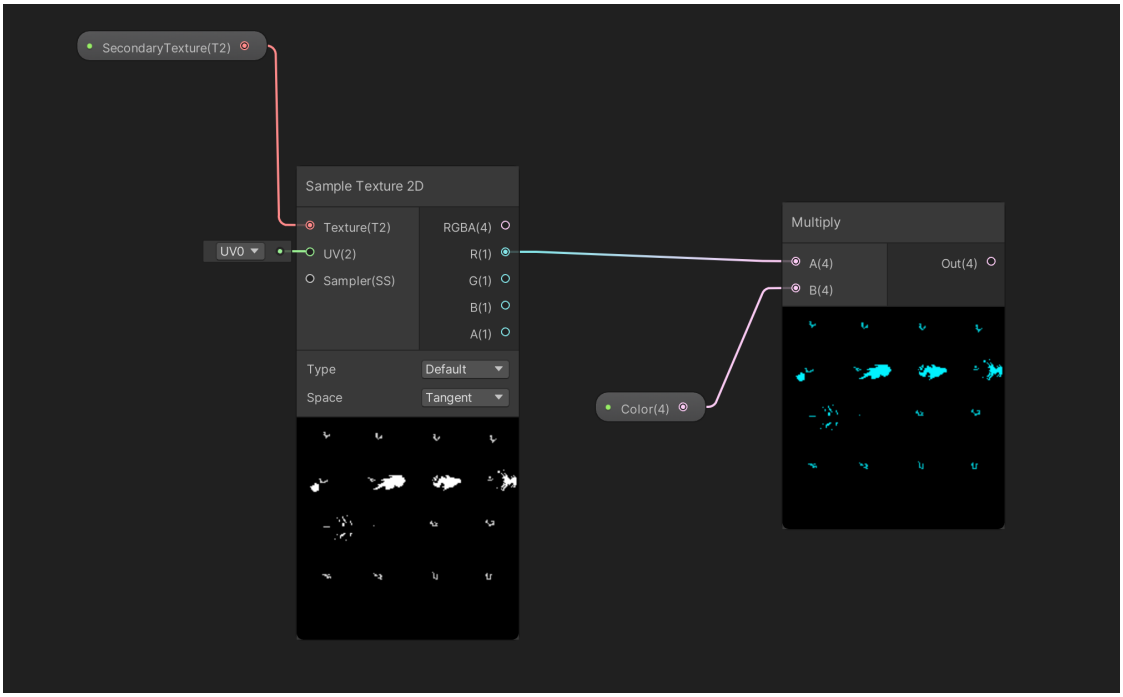
Right click and create **two** Sample Texture 2D nodes. This node as the name suggests, samples textures from either properties you set before: MainTexture or SecondaryTexture.

Drag MainTexture and SecondaryTexture from the Blackboard graph pane on the left, and connect each to the Texture(T2) input of each Sample Texture 2D nodes, as shown:



Multiply Node

Then create a Multiply node, and drag the Color property into the editor. Connect them as shown here:

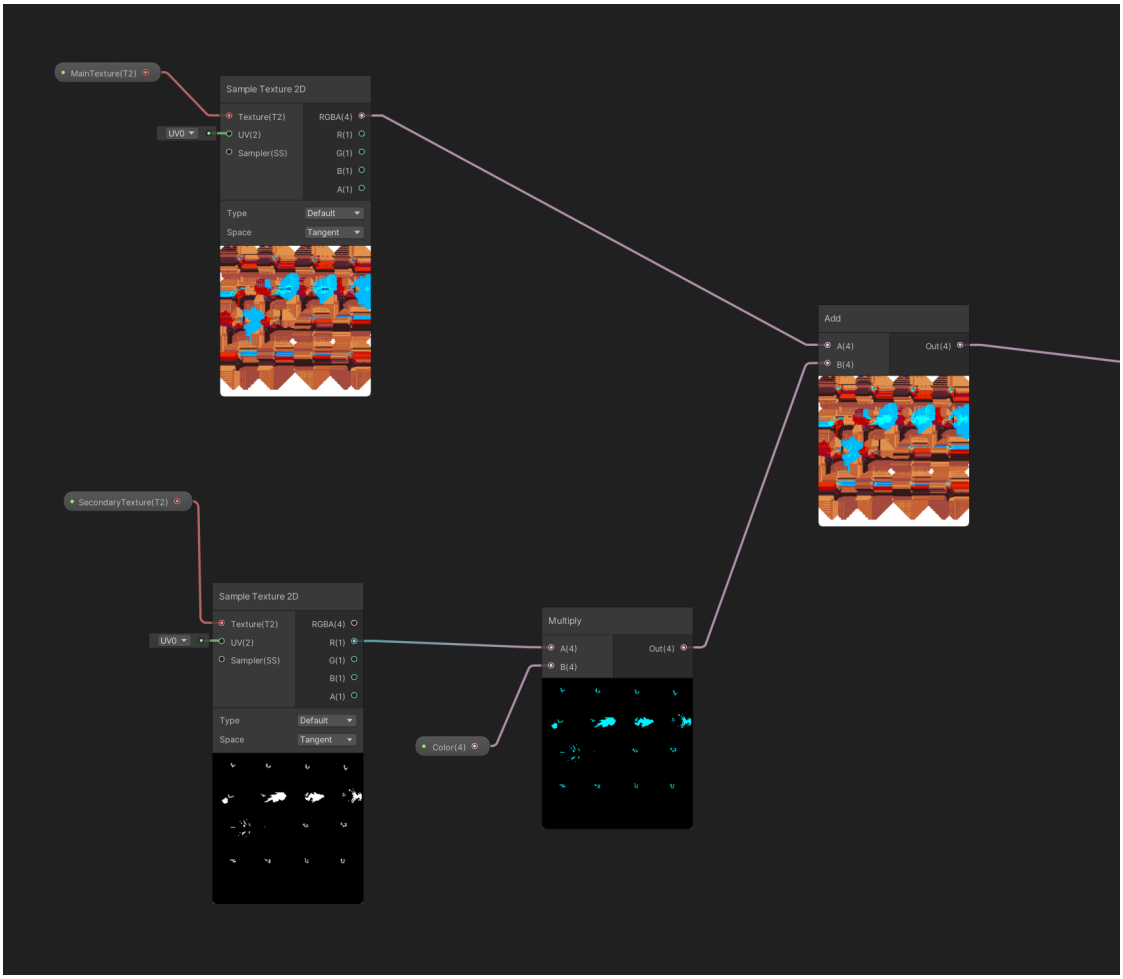


It should be pretty obvious to deduce what's happening: we are doing element-wise vector multiplication. Since the color of each point in the secondary texture is white and everything else is black (zero color value), multiplying it with another color results in that color nicely overtaking all white points in the texture.

> It doesn't matter if you connect R, G, B, A, or RGBA output to the multiply A-input port since the emission texture is purely black and white.

Add Node

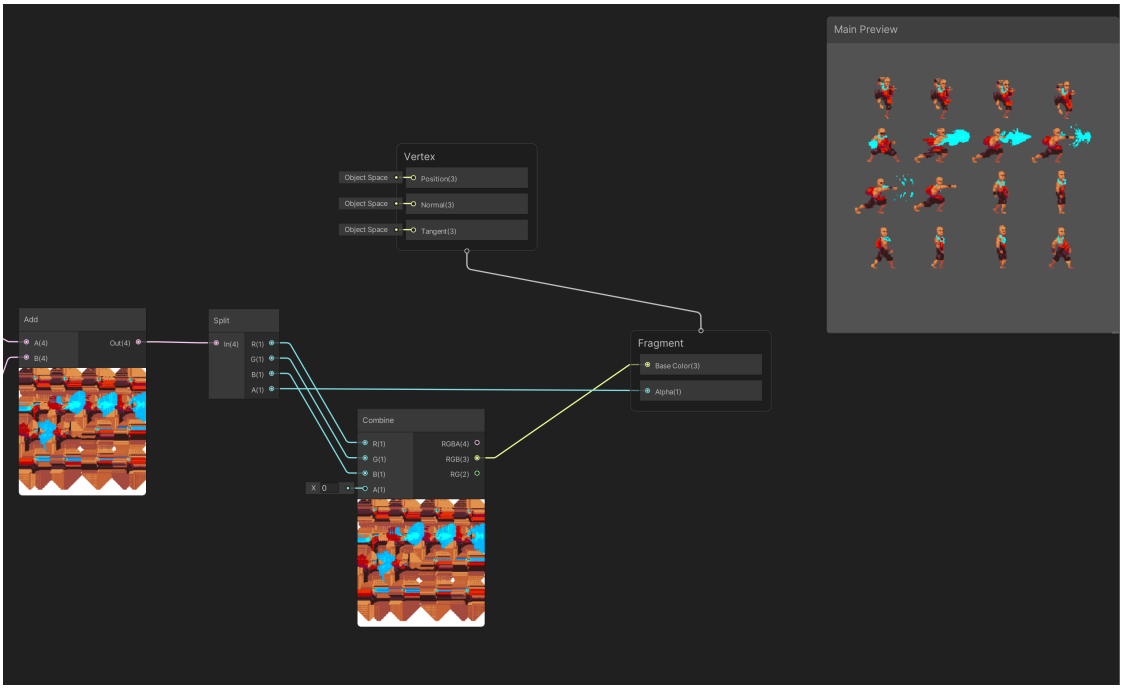
What's left now is to add the RGBA output of the main texture to the RGBA output of the previous step. Create an Add node and connect them accordingly.



Splitting and Combining Channel

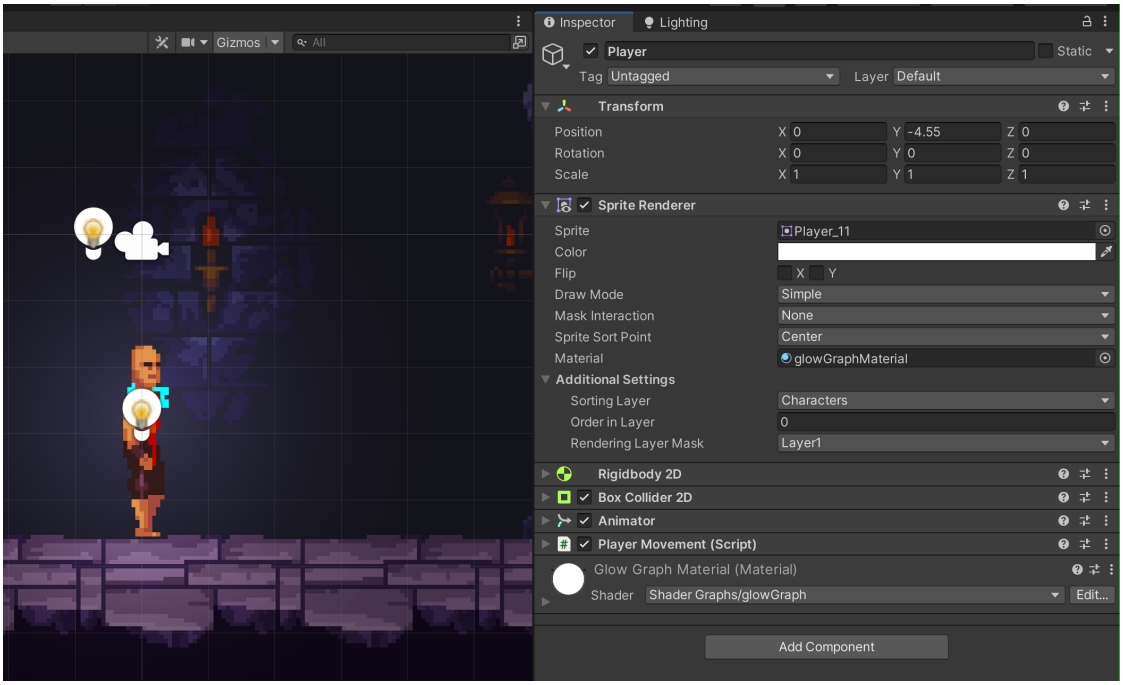
We cannot connect the output of Add node directly to Base Color input of Fragment node because of the mismatch in the dimension. Base Color in Fragment only accepts RGB, and the Alpha is treated as a separate input port.

Hence, we need to split the RGBA Channels, and Combine the back the RGB while keeping the Alpha channel separate. Create a Split and a Combine Node, and connect them as such. You should see in the Main Preview (right click » quad to see it on a plane instead of a circle) that the Player sprite now has a nice teal color at the emission parts:



Saving Asset and Loading the Material to Player

To test the shader, click Save Asset as usual in the graph editor, and load the material utilising this glowGraph shader (e.g: glowMaterial you created earlier) as the Material property of the SpriteRenderer component in the Player gameObject. You should see something like this, where the player's necklace is now brighter.



Postprocessing

One final thing to do is to enable *postprocessing*. Postprocessing effects are full-screen, and it can really change how the game looks very quickly. See this manual for more information.

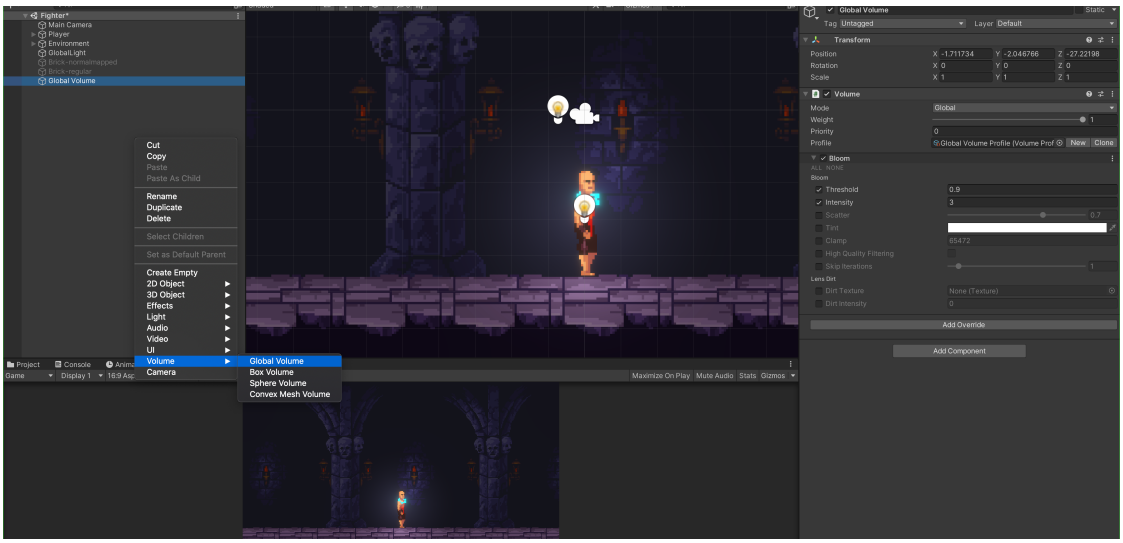
Enabling postprocessing

In order to “see” the postprocessing effect, you need to enable it in your Main Camera.

- Click on the Main Camera
- On the inspector, ensure that the Renderer is set as 2D Renderer,
- And then tick the **Post Processing** property

Adding Volume and Override

We need to tell Unity the context of which postprocessing should be applied. To do this, right click in the Hierarchy and create Volume » Global Volume. Over at the inspector side, create new Profile. Then, Add Override » Post Processing » Bloom. Enable the bloom filter and set the Threshold and Intensity as shown:



Bloom Filter

Bloom gives the illusion of an extremely bright light and is a great way to enhance add visual ambiance to your Scene. The two properties we set here are:

- Intensity: the strength of Bloom filter
- Threshold: Filters out pixels under this level of brightness

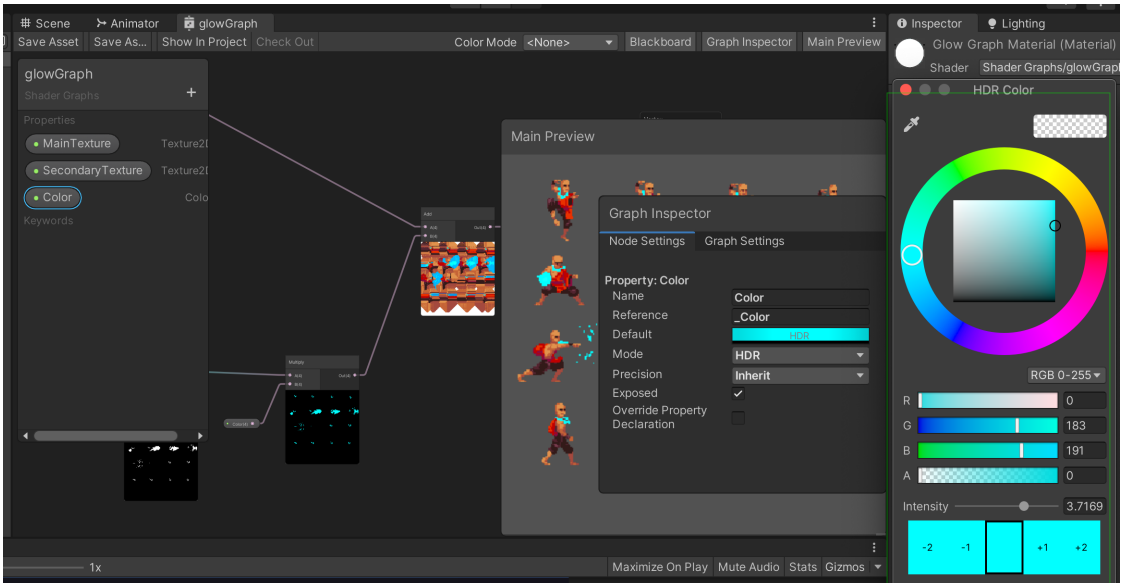
> If we did not apply the glowGraph shader to the player and instead to use regular Sprite-Unlit-Shader, then the necklace or emission effect **will not be bright enough** for the Bloom Filter to pick up. Now save your scene and observe the nice effect in all its glory. Feel free to adjust the Light gameobject on the player: intensity or color to indicate different “aura”.

Other Post-Processing Effects

There’s a ton of other effects which we will not cover here, but worth exploring if it is necessary for your project. Another commonly used effect is the Depth of Field (adding Bokeh), Motion Blur, Screen Space Reflection, etc. You can view the full list here. Remember to not overdo, but only invest your time in features that *matters* to your game.

HDR

To add even more punch with the colors, we can enable HDR. Head to glowGraph shader and click on the Color properties. Under Node Settings, change its Mode into HDR. You can now adjust the intensity of the color manually from the **Material’s inspector.**



The result really packs a punch, with more “glow” effect. You can increase the Bloom filter’s intensity to your liking.

Checkoff Task: Applying ‘Glow’ on Mario’s Shirt

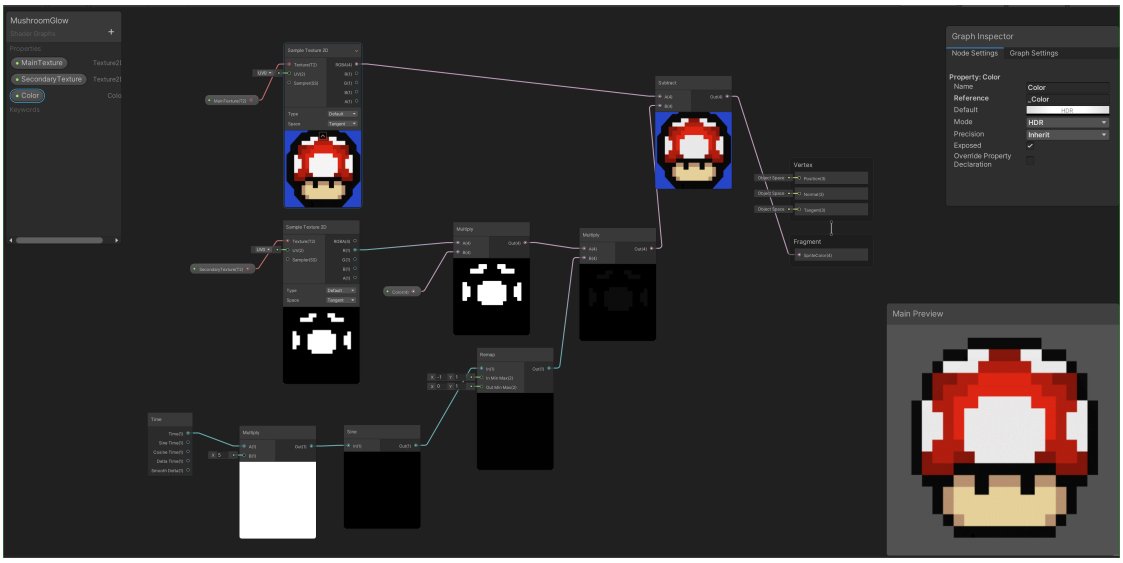
Your task for checkoff now is to apply glow on Mario's shirt. For example, the following is done by adding some point light on Mario, then creating another texture only on the poses where Mario is running (use Photoshop or any image editor), and applying a Bloom filter. Ignore the dust particle effect. It is not part of the checkoff, but simply just a nice visual effect to add.

Other Simple Shader Samples: Blinking, Glow Outline

You can also use the shader to dynamically change the texture of the object at runtime, such as creating this blinking mushroom:

Note: download the mushroom assets yourself. There's a lot of free ones that can be found in the internet.

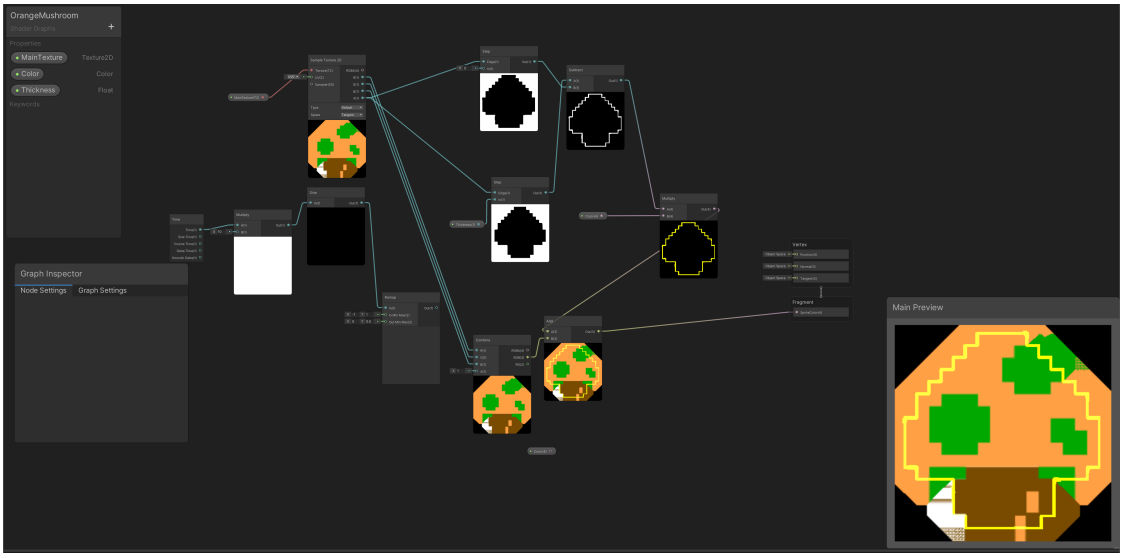
The shader graph used to render the Blinking mushroom is:



You need a secondary texture that contains only its white pixels. This can be done fairly quickly in Photoshop or the likes.

To create a "glow" outline, you can use the step function, although it might not always be obvious. You can enhance the glow using Bloom filter later on. The

shader graph for the glowy orange mushroom is as such:



Note that this method does not always work for all kinds of texture. The gif above showed that the outline of the orange mushroom isn't that clear. The reason lies in how the **step** function operates:

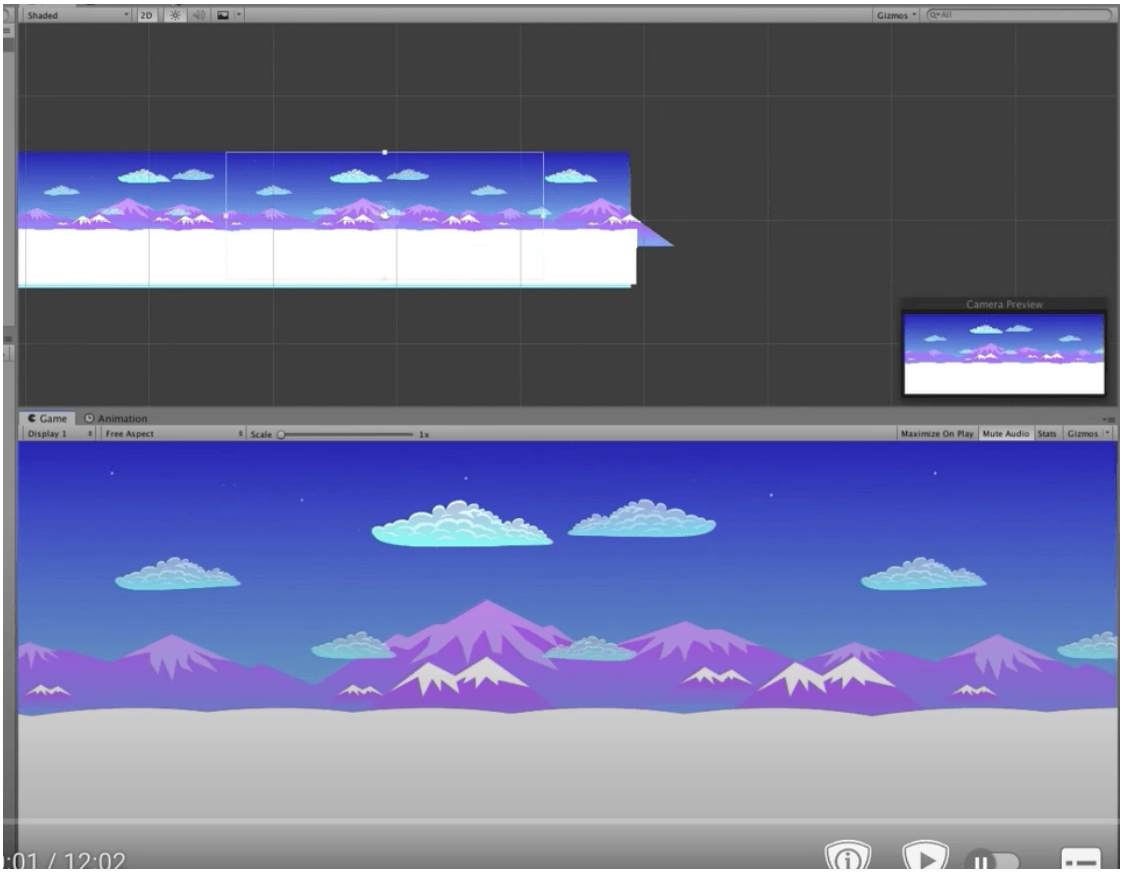
- If you read the documentation, shader **step** function returns 1 **if the `Edge` Input is greater than `In` Input**.
- The `Edge` input is receives by the texture's alpha channel, which has a value between 0 to 1.
- This works because most images are usually "blurry" on the side. You might need to edit the sprite image first and blur the edges in order to give a more discernible outline using the step function.
- Then ignore the alpha using the Combine node so that the final output does not have that blurry edge from the original texture:

Parallax Background

Another nice effect that you can add to your game is parallax scrolling. Parallax scrolling is a technique in computer graphics where background images move past the camera **more slowly** than foreground images, creating an **illusion** of **depth** in a 2D scene of distance.

Here's a preview of such effect. The foreground trees move with Mario but the other trees move *slower*, hence creating the illusion that the background trees are *further* away than the foreground trees.

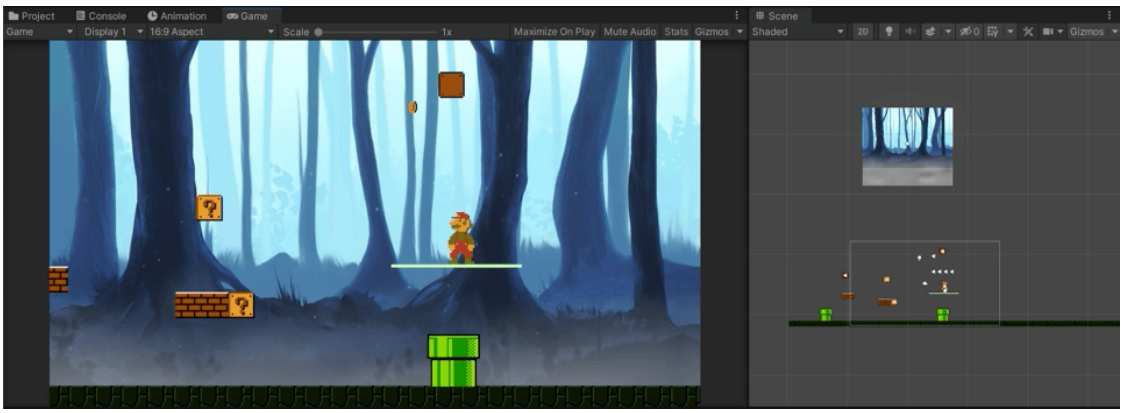
There's a lot of methods that can do this, such as creating different gameObject for each sprite, and then follow Mario at different speed, spawning new ones once the end of the sprite is about to be reached, like this online tutorial:



However, we are going to take this opportunity to learn more about Cameras:

- Setting **Layers** and **Culling Mask**
- Setting Texture **offset**
- Using **secondary cameras** and **combining** the output with the Main Camera

Our scene will look something like this. The parallax scrolling effect is achieved without having to spawn new GameObjects.



Let's get started!

Getting Parallax Background Asset

The first step is to get vector images that support parallax scrolling. These backgrounds are mainly composed of different layers. You can download some samples for free online such as from here. The specific one that was used for this demo can be found here.

Import them as sprites to your project.

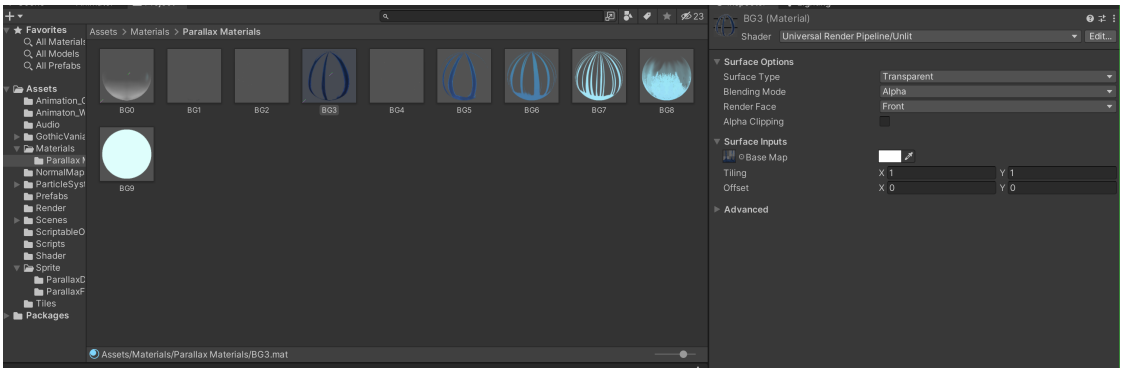
Creating Materials and Layers for Each Background Object

Then create a Material for each sprite, with the following setting:

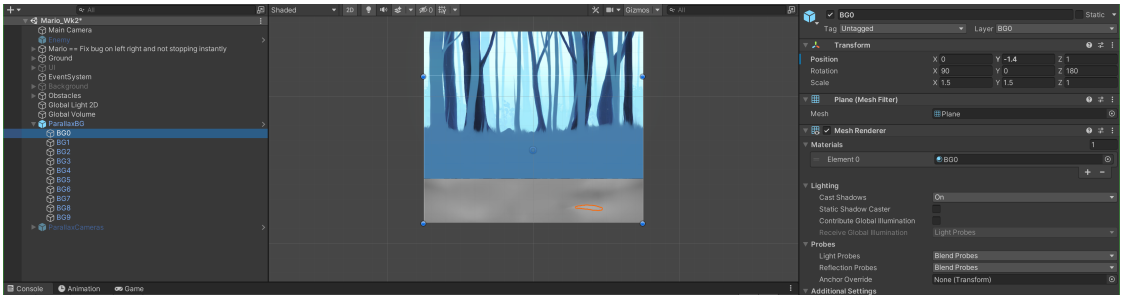
- Shader: Universal Render Pipeline/Unlit (only for this example where we don't need it to be affected by light)

- Surface Type: Transparent
- Blending Mode: Alpha
- Base Map Surface Input: each parallax background Sprite.

An example is as shown:



Then create a new GameObject and name it ParallaxBG. Create as many children 3D » Plane gameObjects as the background sprite layers. Load each of the corresponding Material you just created above to each GameObject Mesh Renderer's Material property:



Don't worry if the order is messed up for now, we will fix that later with the cameras. Also, create a Layer for each gameObject, and match them accordingly. In the example above, BG0 GameObject is loaded with BG0 material, and has a **Layer** called BG0 as well.

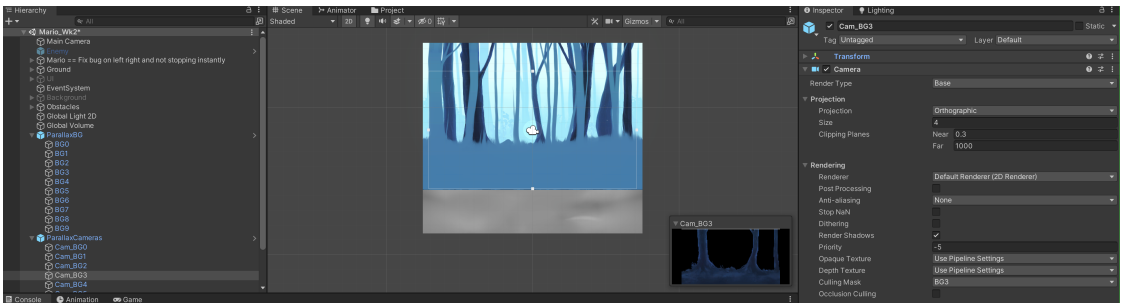
Creating Secondary Cameras

Create a new Empty GameObject and name it ParallaxCameras. Under it, create as many Cameras as the number of sprites making up your Parallax Background. The idea is to allow each Camera to only "see" each corresponding background item,

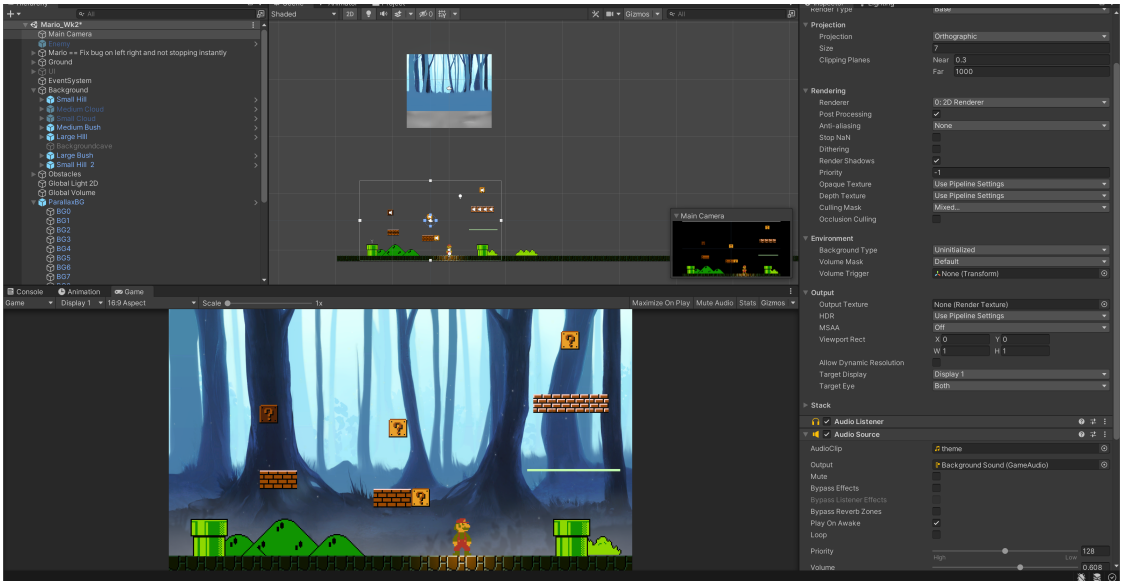
and then combine all their views together with the MainCamera to make up the final output.

Setting Camera Culling Mask and Priority

For each Camera GameObject, set its **Culling Mask** to correspond to each Layer so that it can only "see" the right gameObject. Then, set the **Priority** property of the Camera as well. **The object that's supposed to be rendered "in front" should have higher priority value**. We start with -2 for the foreground, -3 for the one right behind it, and so on, with -11 as the tenth layer. Here's a reference screenshot:



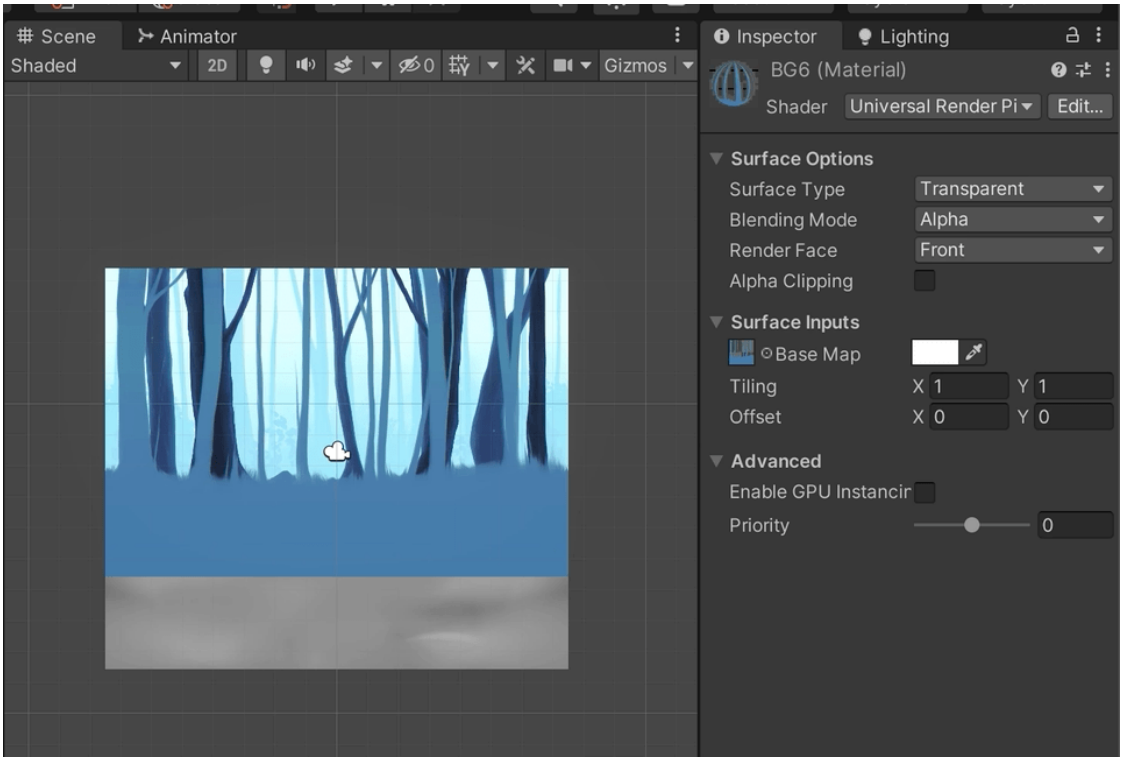
Now click on the Main Camera and ensure that the Culling Mask does NOT include any of the layers for your background gameObjects so the Main Camera *"does not see them"*. Also, set its Background Type to be "Uninitialized". Ensure that you disable any Background "wallpaper". The Main Camera's preview should be dark as shown:



Note that the **Priority** property of the main camera is -1, which results in its output rendered **above** the output of the other cameras with **lower priority**. Your scene preview should look somewhat like the above now.

Scrolling the Background

Instead of physically moving the Background's transform, we will scroll the background by programmatically change each background object's Material **x offset** at runtime at different speed. Here's the effect from doing so:



Create a new script and name it ParallaxScroller.cs, and declare the following variables and `Start` method:

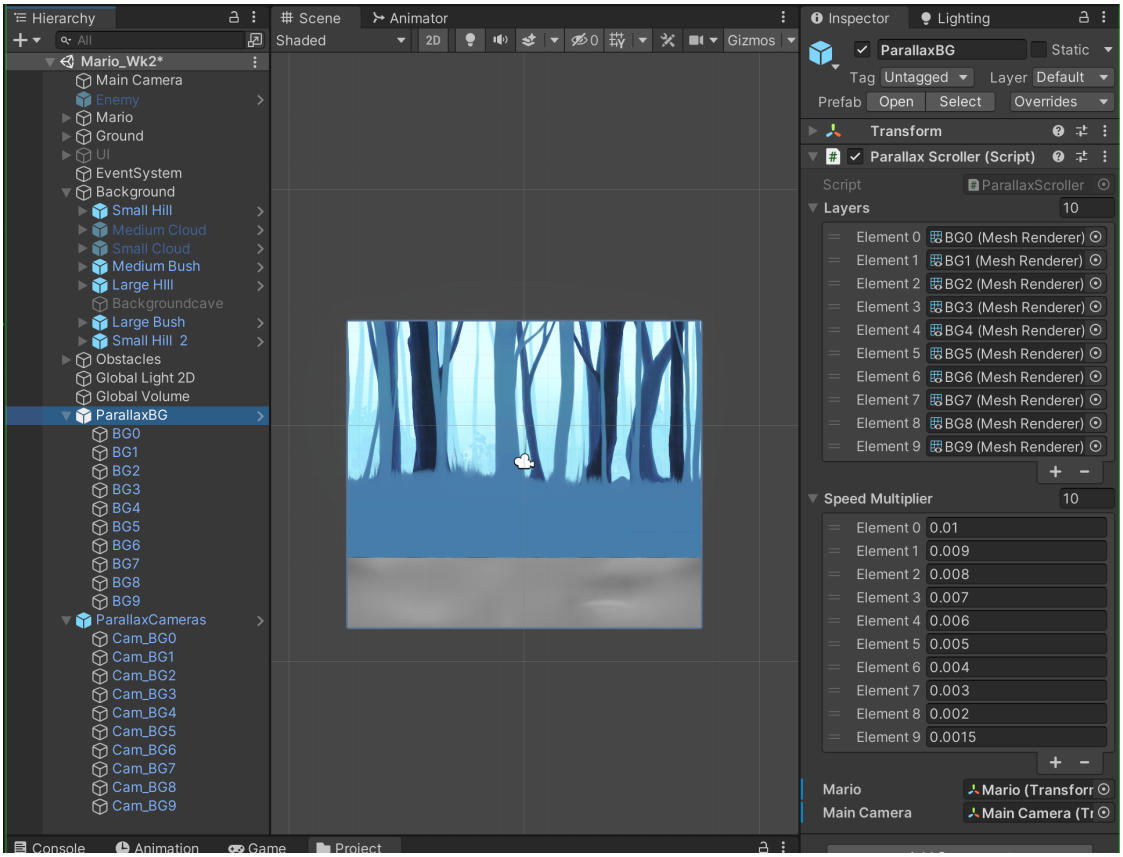
```
public  class ParallaxScroller : MonoBehaviour

{
        public  Renderer[] layers;
        public float[] speedMultiplier;
        private float previousXPositionMario;
        private float previousXPositionCamera;
        public Transform mario;
        public Transform mainCamera;
        private float[] offset;

        void  Start()
        {
                offset = new  float[layers.Length];
                for(int i = 0; i<  layers.Length; i++){
                        offset[i] = 0.0f;
                }
                previousXPositionMario = mario.transform.position.x;
                previousXPositionCamera = mainCamera.transform.position.x;
        }
}
```

The offset array keeps track of each background object's material x-offset value. Initially, they're all set as zero. We then compute how much mario has moved during `Update` and keep track of the MainCamera's location. This because sometimes at the **edges** of the map, the Main Camera doesn't move anymore and we do not want our Parallax Camera to move in this case. It will look weird.

Implement the `Update` function as such:

```
void Update()
{
        // if camera has moved
        if (Mathf.Abs(previousXPositionCamera - mainCamera.transform.position.x) >
                for(int i = 0; i< layers.Length; i++){
                        if (offset[i] > 1.0f || offset[i] < -1.0f)
                                offset[i] = 0.0f; //reset offset
                        float newOffset = mario.transform.position.x - previousXP
                        offset[i] = offset[i] + newOffset * speedMultiplier[i];
                        layers[i].material.mainTextureOffset = new Vector2(offset
                }
        }
        //update previous pos
        previousXPositionMario = mario.transform.position.x;
        previousXPositionCamera = mainCamera.transform.position.x;
        }
}
```

Then load the script to the **parent** gameobject of all your background objects, and **set the parameters accordingly** in the **Inspector** depending on the number of parallax background layers you have. Enter any speed multiplier that you want for each layer, just ensure that they're in decreasing order so the background objects scroll a little slower than the others in front of it.



Test run and we are done. You should have a nice parallax scrolling effect viewable from the Game window.

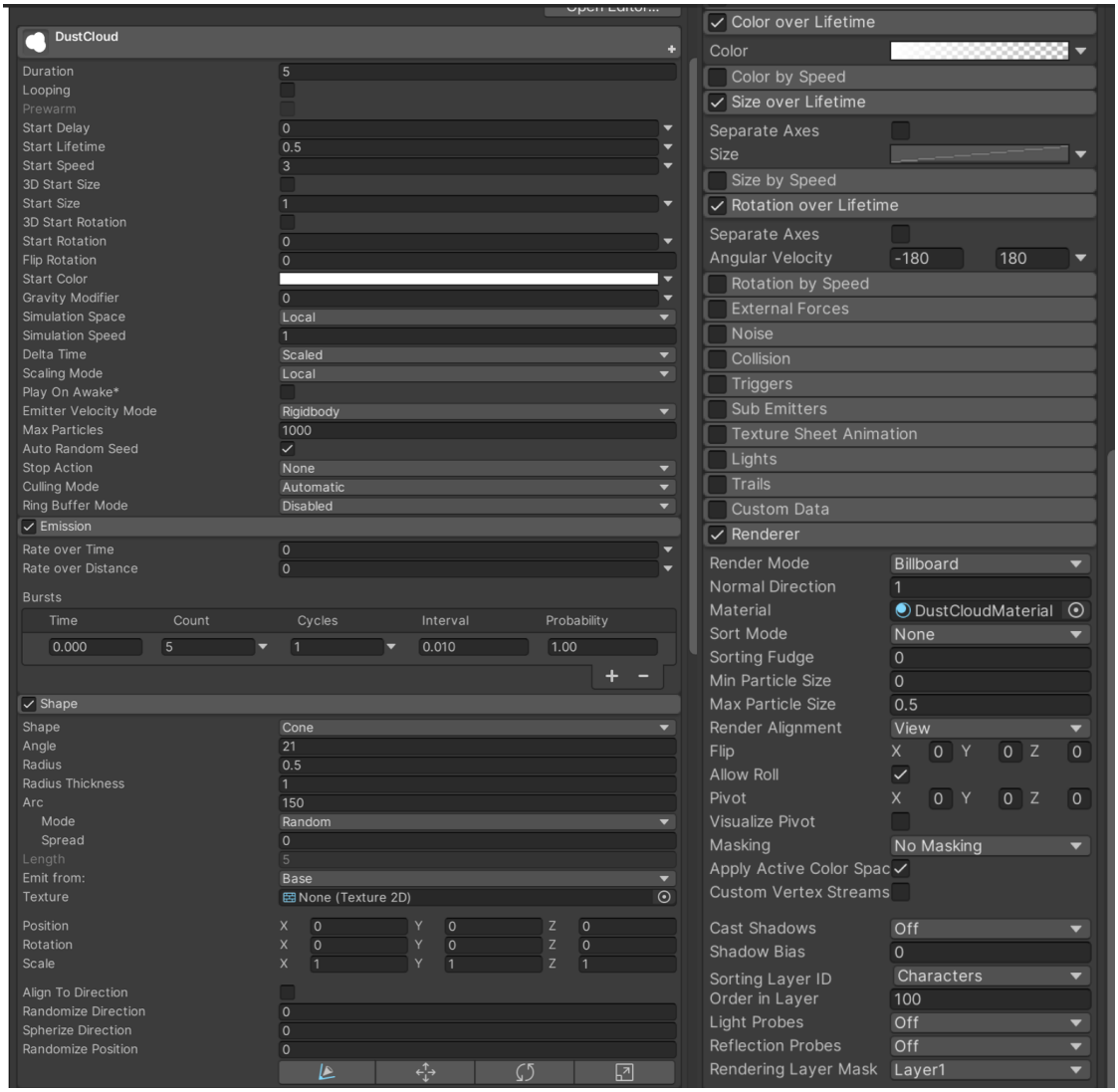
Particle System 2D

The final thing to learn about VFX in this lab is the **particle system**. Using Particle System component, you can simulate moving liquids, smoke, clouds, flames, magic spells, and a whole slew of other effects effectively. You can view their official tutorial here, and in this Lab we are bringing you up to speed with two sample particles: landing jump “dust” and fire. Here’s a demo (the fire glow is enhanced by the *bloom filter*):

Creating Dust Particle

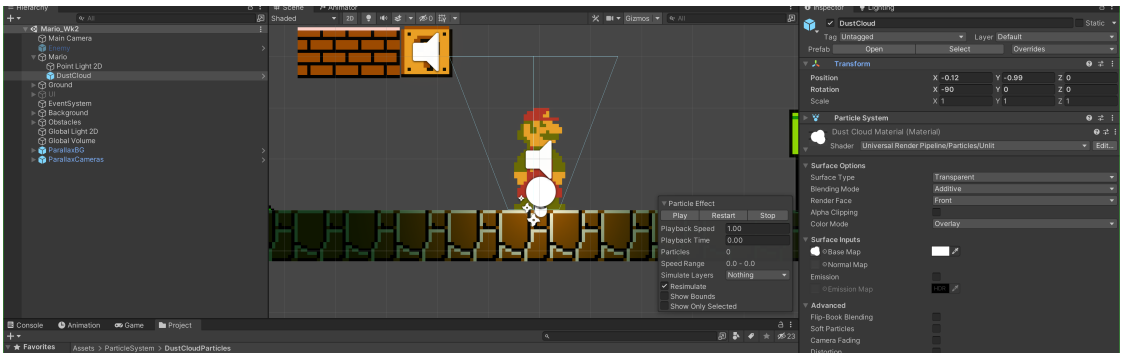
Create a new gameObject and attach a ParticleSystem component to it. Editing particle systems can be mainly done via the Inspector, by adjusting its properties accordingly. We need to first think how we want our particles to behave, for example, here's a few questions we can ask ourselves: How many particles can be emitted at a time? Do they have the same speed? Are they affected by gravity?

For this dust cloud particle, we definitely only want a few particles to be present at a time, and for it to fade over time quickly. They also must be instantiated only once – in **bursts**. The setting of the particle system component must be pretty much as such:



- **Start Lifetime** is short, set at 0.5. We don't need the dust to linger for too long.
- We also **disable** Play on Awake. We will trigger the particle at runtime.
- **Looping** is also disabled.
- The particle has a certain initial **speed** upwards (depending on the direction of "Shape")
- Emission is done in **bursts**, in counts of 5 every 0.01 second (but since looping is disabled, it's going to happen just once when called to `Play()` via script later on).
- **Shape** is "Cone", as the dust gets bigger when it travel upwards.
- **Color over lifetime** turns transparent, so we can no longer see the dust after a short period of time.
- **Rotation** over lifetime is enabled, so the dust will rotate as it moves upwards and appear more natural.

- Finally, we use a **custom material** (under Renderer).
  - Create a new material called Dust Cloud Material, with Shader set as: Universal Render Pipeline » Particles » Unlit
  - And load the Cloud sprite under Base Map
  - Set other properties as such:

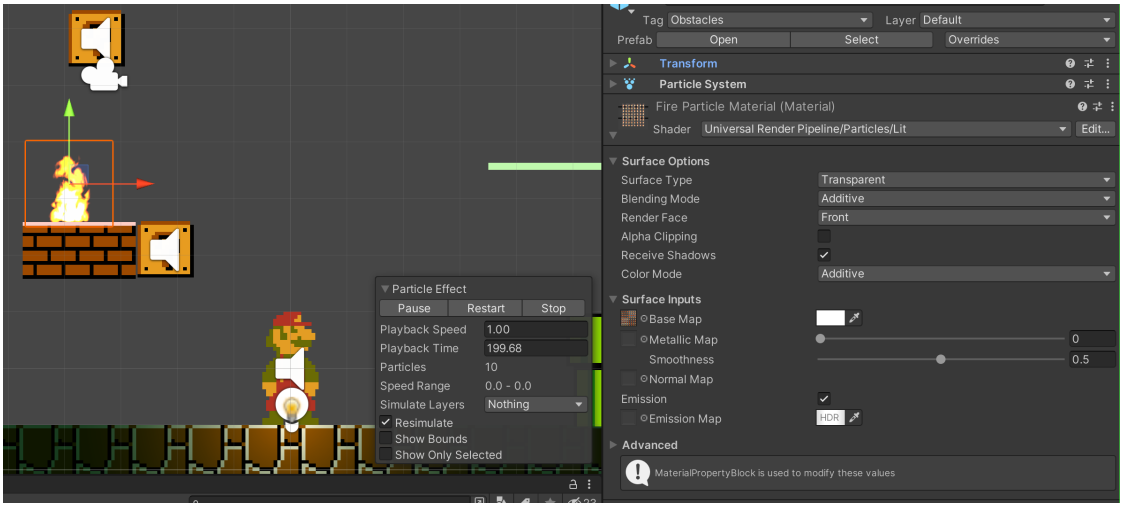


Place the DustCloud gameObject as a Child object under Mario. Now edit PlayerController.cs to trigger the DustCloud each time Mario lands on the ground using `dustCloud.Play();` where `dustCloud` is a reference to the Particle System component. Pretty sure you can do this by yourself easily.

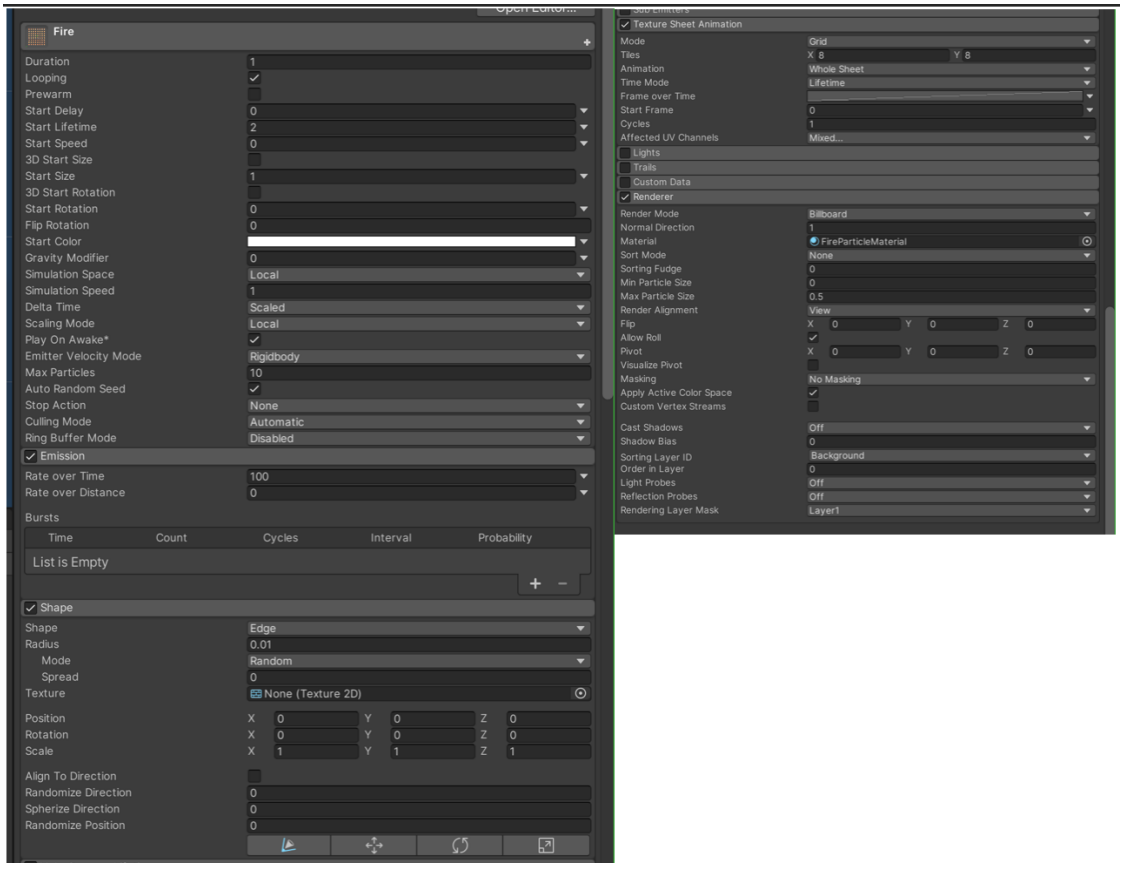
Creating Fire Particle

The Fire particle is created via **Textured Sheet Animation**. A particle's graphic need not be a still image. This module lets you treat the Texture as a **grid of separate sub-images** that can be played back as **frames** of animation. You can use your own sprites for this, or use the provided Cartoonfiretexture. If you're importing your own textures, make sure you **slice** them properly and set the texture's Sprite Mode property into **Multiple**.

Create a new Material called FireParticleMaterial using the Shader UniversalRenderingPipeline » Particle » Lit to make particles appear photorealistic. Set the Materials to have the following properties:



Now create a gameObject called Fire with a ParticleSystem component attached. Set its property as such:



The main learning point here is to set the Texture Sheet Animation property as Grid, and to take up the Whole Sheet (the “texture” refers to the texture provided by the Material). You should observe the flame animating nicely by now. If you want Mario to be “burned” by it, you need to add a Collider component and implement a **Trigger callback** as we have learned before.

“Breakable” Prefabs

Another common feature that we can have in a game is to have “breakable” things, as shown:

There’s a lot of different tips and tricks to do this. You’re free to implement such effect however way you want: using particle system, using animation, or programmatically via the script. In this example, we will go back to the basics and use the latter.

To create this simple ‘breakable brick’, we need **three** gameObjects:

1. A gameObject that contains the **Sprite Renderer** of the breakable brick and a regular **BoxCollider2D**.
2. A **child** gameObject of (1) that contains a single **EdgeCollider2D** to detect collision only at the **bottom** of the brick
3. A single “debris” **prefab**, which is simply a gameObject with RigidBody2D component attached (so gravity can act on it) and a Sprite Renderer. It has a single script `Debris.cs` which tells the Physics engine to propel this single

debris outwards at the point of instantiation, and render itself invisible after some time.

Now create a new script called `BreakBrick.cs`, to be attached in gameObject (1). Open the script and create a few variables:

1. A **boolean** to indicate whether this brick has been ***broken*** or not,
2. A reference to the "debris" **prefab**

In `BreakBrick.cs`, implement `OnTriggerEnter2D` callback such that when the gameObject with this script collides with "Player", it spawns a few Debris prefab and disable all colliders: both edge collider and box collider, as well as the sprite renderer of the breakable brick:

```
void  OnTriggerEnter2D(Collider2D col){
        if (col.gameObject.CompareTag("Player") &&  !broken){
                broken  =  true;
                // assume we have 5 debris per box
                for (int x =  0; x<5; x++){
                        Instantiate(prefab, transform.position, Quaternion.identity)
                }
                gameObject.transform.parent.GetComponent<SpriteRenderer>().enabled
                gameObject.transform.parent.GetComponent<BoxCollider2D>().enabled  =
                GetComponent<EdgeCollider2D>().enabled  =  false;
        }
}
```

You are free to **destroy** this gameObject (1) afterwards.

The Script `Debris.cs` attached to a single debris prefab governs how each debris should behave. From the sample .gif above, each debris spawned will have an initial upward force with a random x-component, initial torque, and then it will gradually reduce in scale while being affected by Gravity.

We need a **Coroutine** for this, because we need to slowly reduce the scale of the object **after each frame**, giving control back to Unity to perform other

computations in the meantime. A sample implementation is as such:

```
private  Rigidbody2D rigidBody;
private  Vector3 scaler;

// Start is called before the first frame update
void  Start()
{
        // we want the object to have a scale of 0 (disappear) after 30 frames.
        scaler  =  transform.localScale  / (float) 30 ;
        rigidBody  =  GetComponent<Rigidbody2D>();
        StartCoroutine("ScaleOut");
}

IEnumerator  ScaleOut(){

        Vector2 direction =  new  Vector2(Random.Range(-1.0f, 1.0f), 1);
        rigidBody.AddForce(direction.normalized  *  10, ForceMode2D.Impulse);
        rigidBody.AddTorque(10, ForceMode2D.Impulse);
        // wait for next frame
        yield  return  null;

        // render for 0.5 second
        for (int step =  0; step  < 30; step++)
        {
                this.transform.localScale  =  this.transform.localScale  -  scaler;
                // wait for next frame
                yield  return  null;
        }

        Destroy(gameObject);

}
```

You can enhance the effect by adding **sound effect** at the moment of impact, or allow the debris to bounce off other things in the scene.

Checkoff

Implement all the following in any way you deem fit, be it following our tutorial above or other creative avenues:

1. Breakable bricks (you are free to define what breakable means, that is if the debris has collisions with the ground, tiny animations, etc).
2. Glowing mario shirt when he is running, or jumping, or doing anything.
3. Use particle system to create any interesting object, you must utilise **textured sheet animation**.
4. Create your own simple graph shader for anything in the scene that changes at runtime (hence you need to use the `time` node).
5. Implement parallax background effect

Here's a checkoff gif sample to follow if you don't want to think too much:

Next

We are almost equipped to recreate World 1-1 of Super Mario Bros, with a few enhanced effects. In the next Lab, we will implement a Game and State Manager, an Object Pooler, and also an Audio Mixer. It will be mostly housekeeping and

some C# stuff that hopefully will help you to arrange your code in a neater way. Now that you have understood Unity basics, it's time to clean up these loose ends.

NEXT POST
Unity for Babies