Unity Take Home

Tuesday, October 10, 2023 2:34 AM

The following is week 7 in the Speaking Shaders lesson for Parsons Design and Technology.

Let's dig into Unity and its associated shaders. This is going to be a fairly straightforward process (though long) document to get you all set up in the (formerly?) popular game engine Unity. Why Unity? It's a place where your shader knowledge can find some immediate usage in real-time 3D rendering. Just a fair warning a lot of this is just learning about the system. Please bear with it!

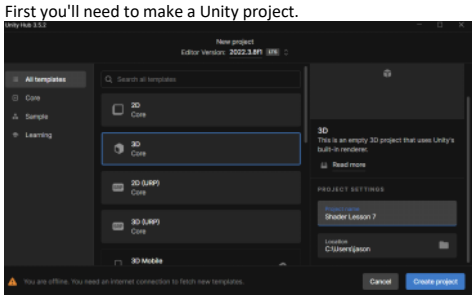
A quick overview of what all we'll be covering in this self-led lesson (and it is *a lot*):

1. Setting a Scene
    a. Unity
    b. Model Fetching
    c. Unity Importing
    d. Model Fixing, A Blender detour
    e. Unity Importing II
    f. Adjusting Materials
    g. Finalizing the setup
2. Unity Rendering
    a. Graphics Engines
    b. Material matters
    c. Writing your first Unity Surface Shader
    d. Shader Lab
    e. Assigning a Shader
    f. Making a vertex animated shader
3. PBR Theory
    a. Homework

Installing Unity

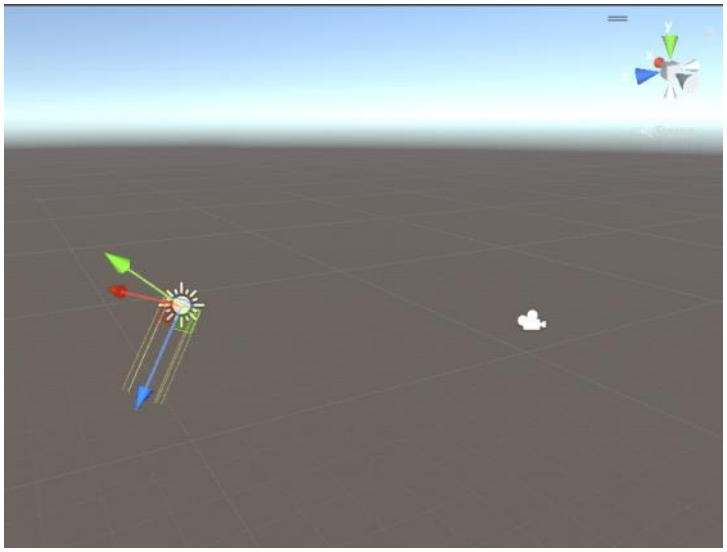
Go to https://unity.com/ and get Unity Hub and get the latest LTS build of Unity. It doesn't really matter which version you use, but I use the 2022.3.8f1 LTS. Variations between the same major versions don't matter too much (so any 2022.3.x version is equivalent to this). Note that you will need to make an account and get a personal license for this. I am somewhat counting on you being resourceful enough to understand how to navigate the Unity editor.

First you'll need to make a Unity project.



Call it Shader Lesson 7 and set it to be 3D Core. For our purposes in this demo we don't need the URP/HDRP features. Ask me to soap box about them later. And you can learn them later too!

This will give you a basic empty Unity project. There's a file browser, an object hierarchy, a scene view, a game view, and an inspector. This is the basics of the Unity project where you will do your object manipulation for scene setting. There is a gizmo for controlling the camera and this is the thing that will be responsible for showing you what is going on in "game."



Basic Anatomy of a Unity Scene

Unity is an ECS based workflow. That means there are Entities, Components, and Systems. For us, there are mostly just Entities and Components. In Unity you will be working with Objects, which have a transform component that describes their physical location, rotation, and scale. Everything in Unity that is an object has a transform component (even if it isn't relevant.

Getting a Model

For a scene, we need assets! I'll be using these two assets from these two URLs for this example.
For this example walkthrough we are going to use these two models.
Nintendo 64 - Super Mario 64 - The Models Resource (models-resource.com)
https://www.models-resource.com/nintendo_64/supermario64/model/503/

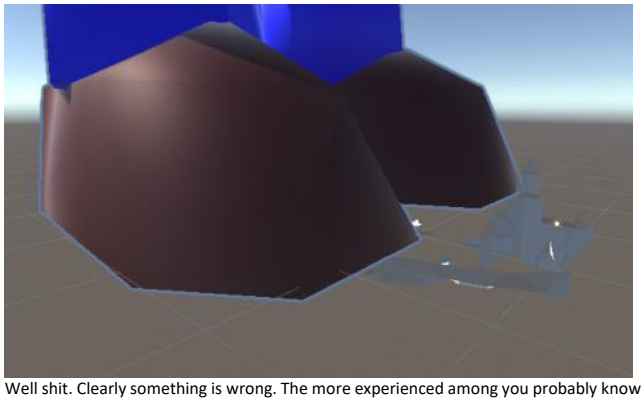
One of Mario
One of the castle exterior

With these two models, we'll be attempting to recreate a scene from this game, Super Mario 64. This scene is particularly important in real-time 3D graphics! It's one of the first fully interactive 3D environments that people could navigate on home consoles!

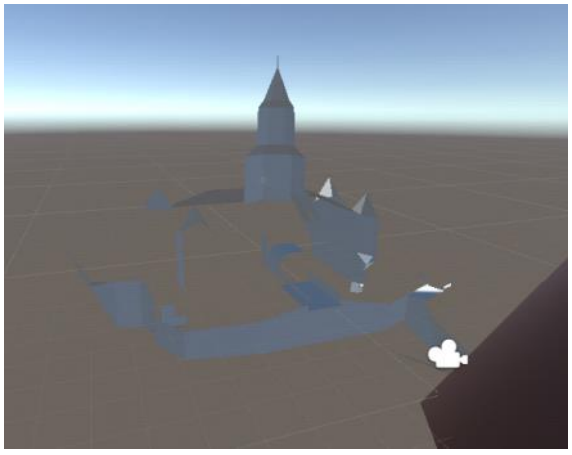
We're going to do our best to recreate this shot:

Importing into Unity

Supposedly getting models into Unity is easy so let's go ahead and import our models into Unity now and see what we got. To import the models, simply drag and drop the .fbx of mario and the .obj+.mtl of the castle exterior into the Unity file explorer in the editor. This will make copies of the files for unity to use. And then drag the objects into the scene.



Well shit. Clearly something is wrong. The more experienced among you probably know what's immediately wrong: flipped normals (the missing faces), colors (textures being lit strangely), scale is wrong (size), and probably more that aren't immediately obvious. What are we to do?



Recalling for a second that our goal in this part of the class is recreation of things we see in different contexts. Those can be games, films, and more. That means we're going to be confront with assets of varying quality when we go hunting. And we need to be prepared to fix things.

For this we will need to get both models in a state where we can import them. And this is where we can briefly talk about what we need for the model import process. Mario here comes as an .fbx which is probably the most useful format to have things imported in. The castle comes in one of the worst format you could ever want because it imports the object and its materials in two parts. We need to address the castle first and then we can address Mario's face problem.

Model Fixing, A Blender Detour

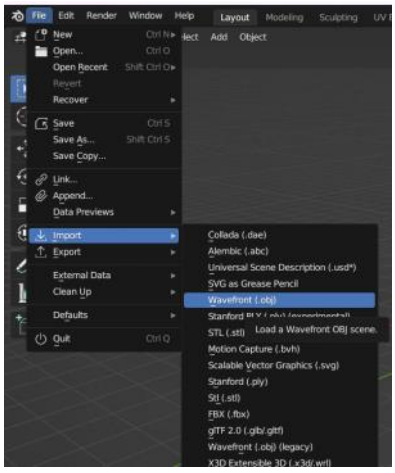
So Blender is going to be a necessary tool in your kit to fix things because real time engines like Unreal and Unity are not equipped for modifying 3D model data out of the box. Time to get Blender now too. This is a very functional tool for you to have in your tool box should you choose to pursue prerendered graphics in your own time. Blender has its own shading process, but most importantly to us, it lets us repackage and prepare our CG models for use.

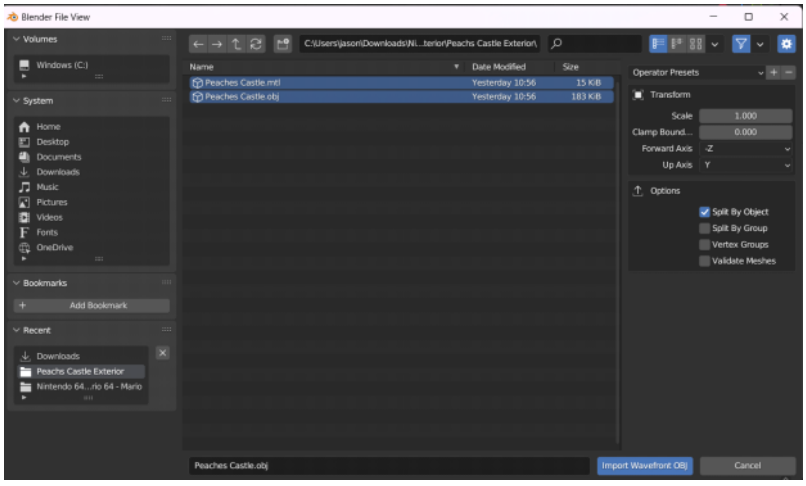
Install Blender from here
Blender 3.6 [blender.org - Home of the Blender project - Free and Open 3D Creation Software]

You don't need anything super special. But we'll start with the castle (the mario file is mostly correct but we need to update its settings).

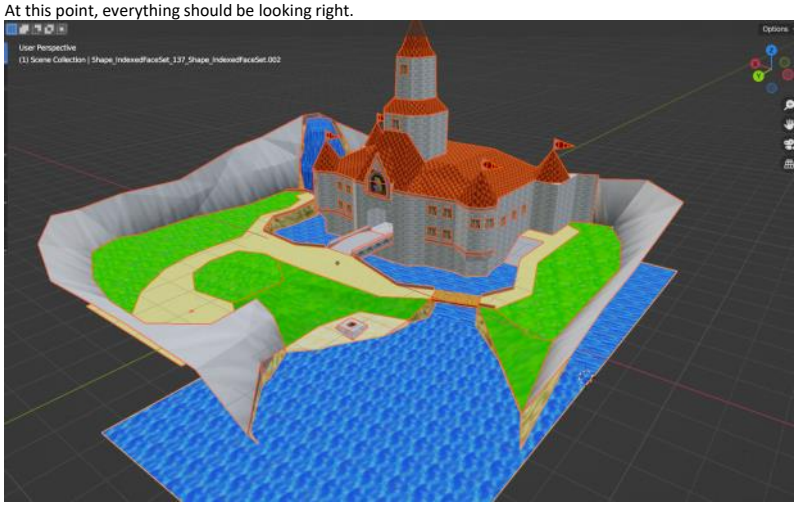
First, open a General scene and delete the default objects. The cube, the camera, and the light. We don't need them (this is Blender cultural tradition)

Then import the .mtl and the .obj into the blender project.

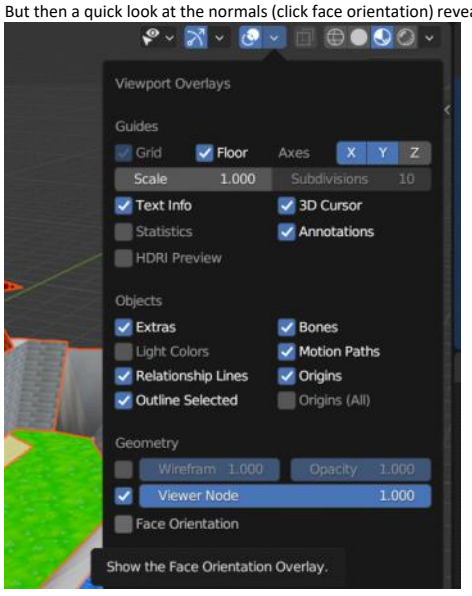


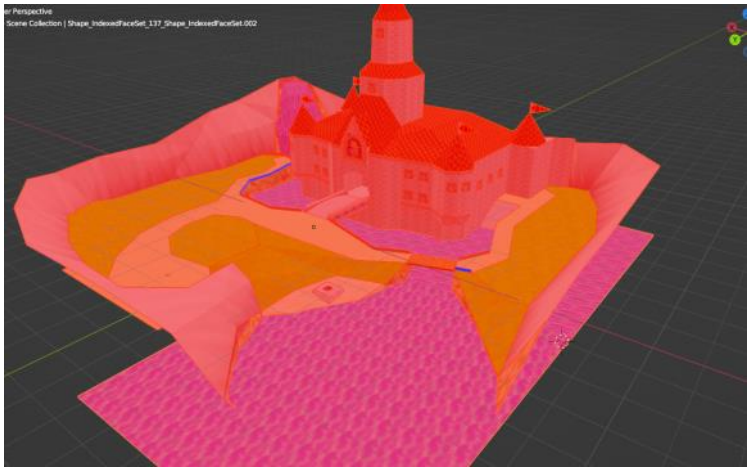


At this point, everything should be looking right.



But then a quick look at the normals (click face orientation) reveals the whole thing is flipped incorrectly.

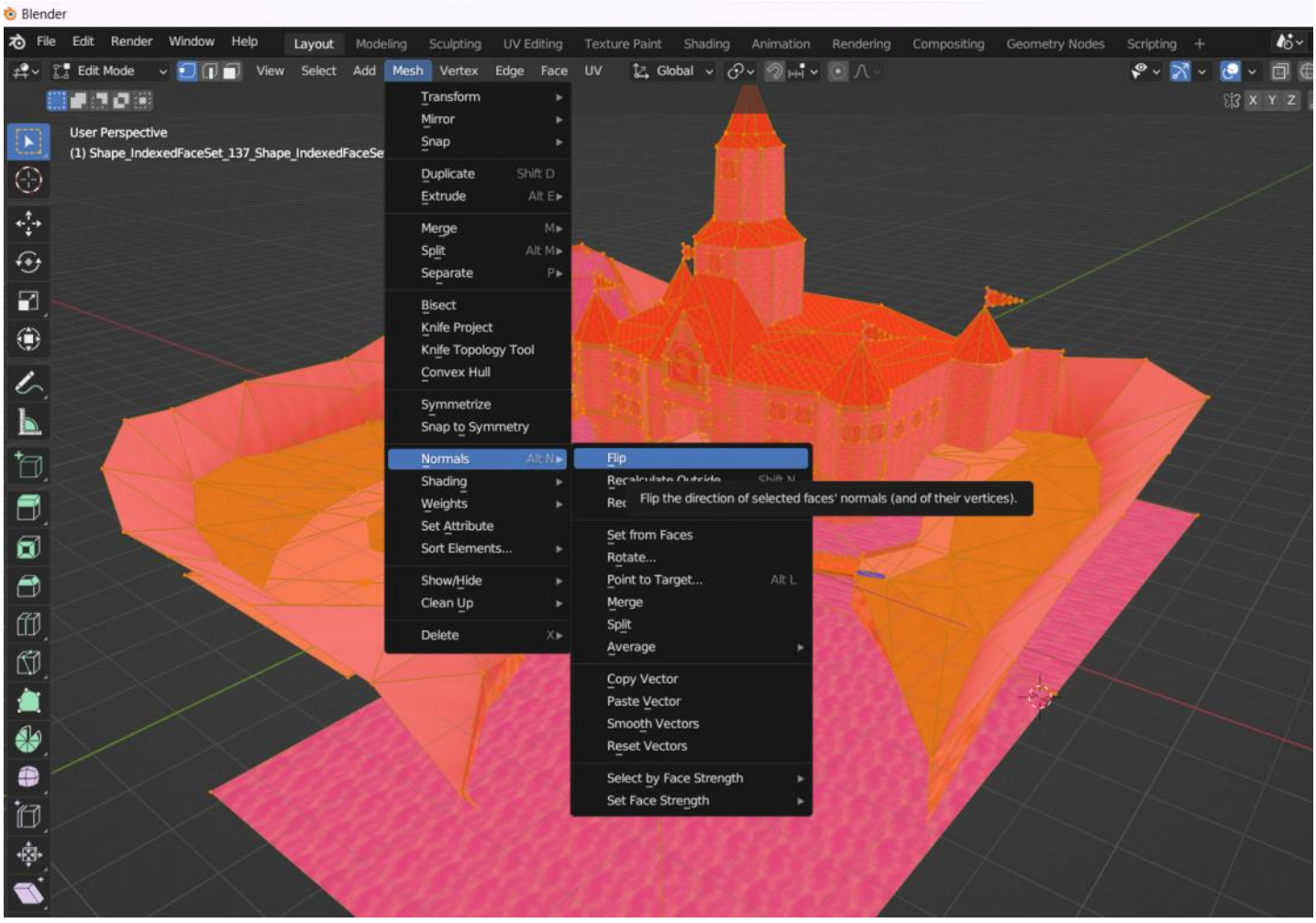




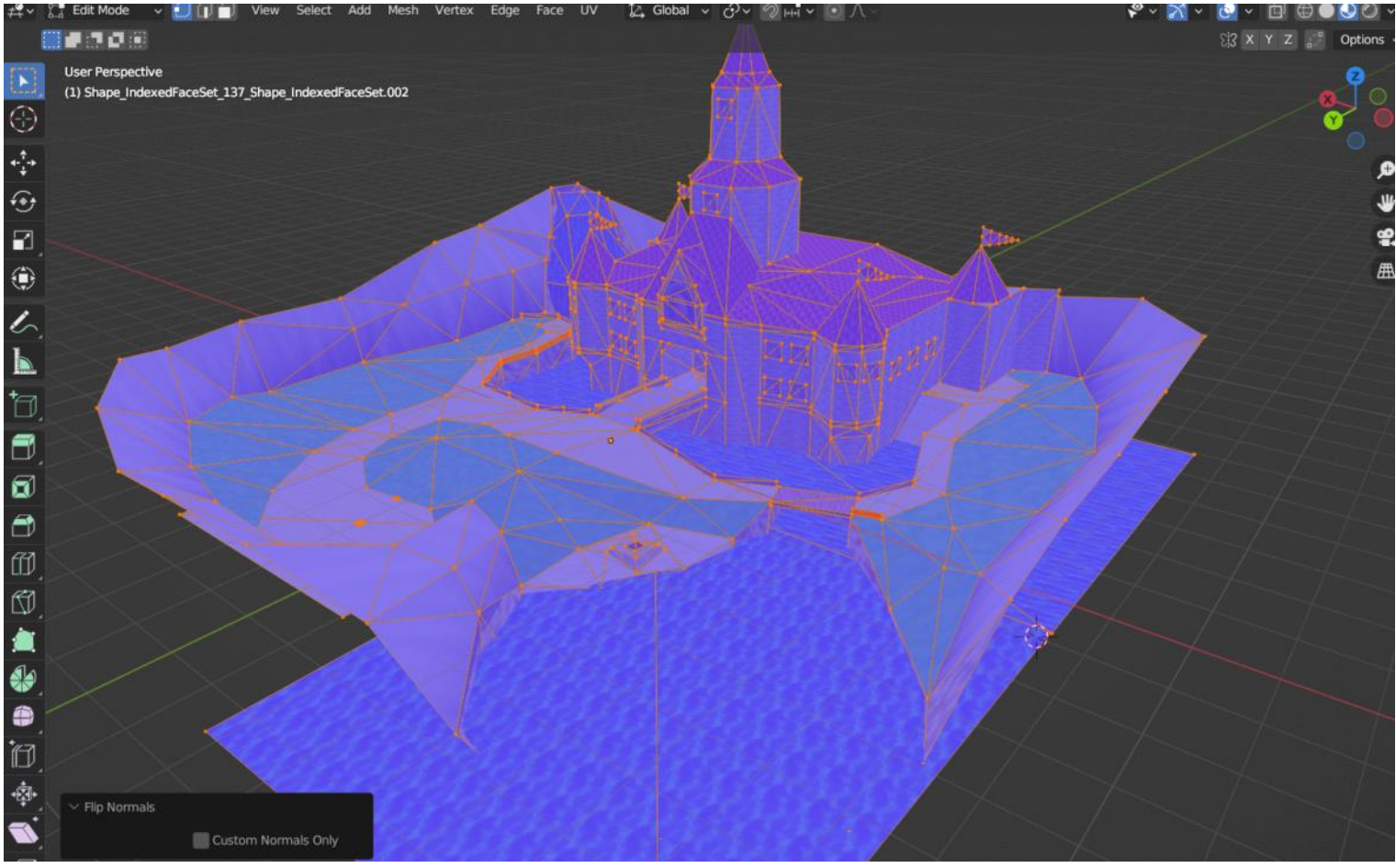
To fix this, we will select *every* object with A, go to edit mode with Tab, select all the vertices with A again, and then go to Flip Normals in the Mesh options. (Blender is a complicated but necessary program)

Again, that sequence is A > Tab > A, then in the menus, Mesh > Normals > Flip. This process is selecting all the vertices of all the objects in the scene an then flipping the normals (red being inside to blue's outside).

See the below image, if done correctly, your entire model should be selected and you should see all the dots be orange.

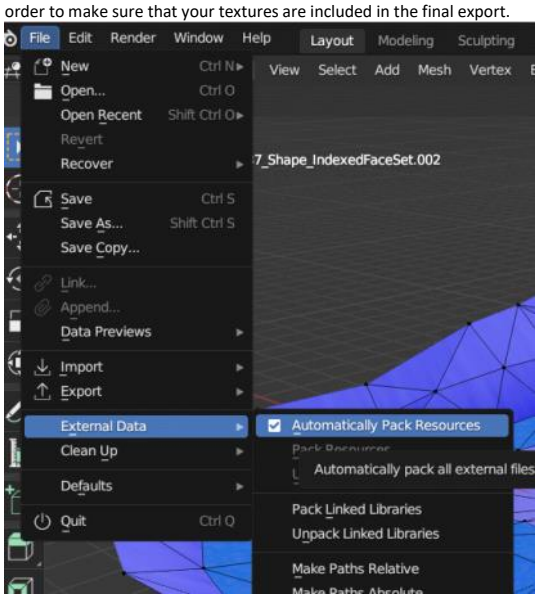


And the result should be blue with face orientation on

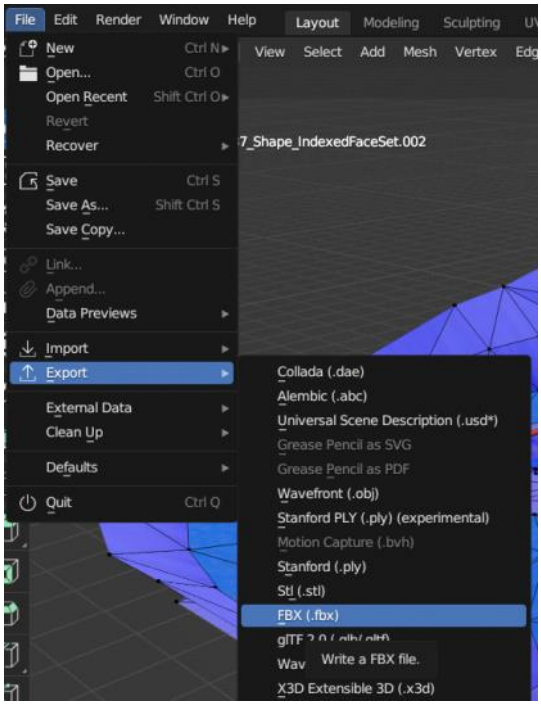


Ok! Now it's time to export this model. And for the record, 99% of the time if you have an imported asset and there are just suddenly random faces missing, it is because of normals being flipped weirdly. You can adjust them easily with Blender. There is also a situation where your shaders can ignore this problem, but we'll solve this problem by flipping the normals since it's clearly the main issue.

To export our corrected model, we will need to make sure things are packed in.
To do this, again in Blender make sure to pack resources. Don't skip this step! It's important to do in order to make sure that your textures are included in the final export.



And then you can (finally) export the actual models as .fbx for you to use.

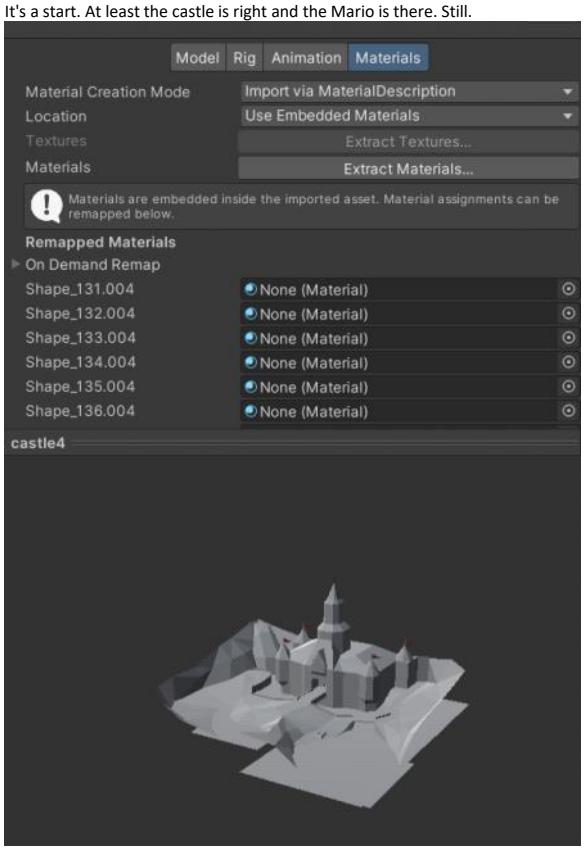


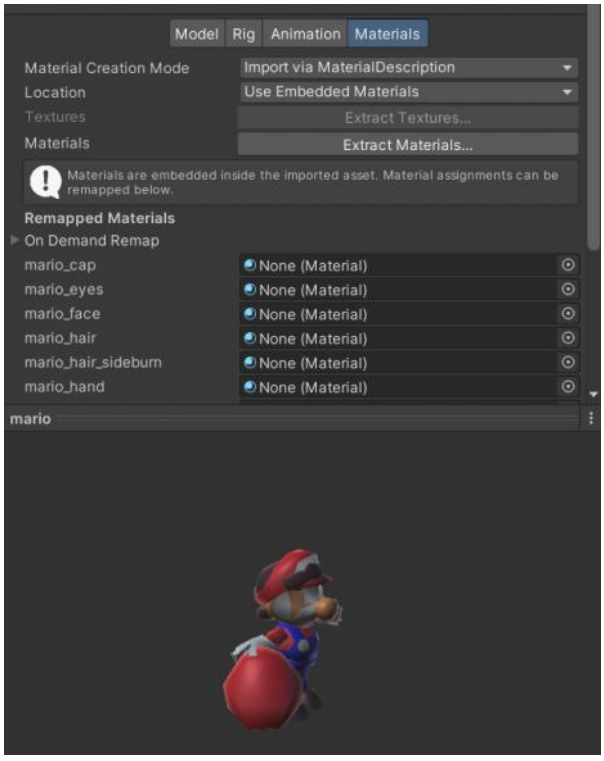
A call out for you: Unity and Blender actually play exceptionally nicely together. You can actually import a whole .blend file into Unity and use the .blend as an asset. Whether or not this is actually helpful to you is up to your workflow. But it does enable you to modify the scene in Blender and have Unity receive the updates immediately.

Unity Importing II

Ok, now we have a mario.fbx and a castle.fbx that we can use. Let's bring these into Unity and see what we got.

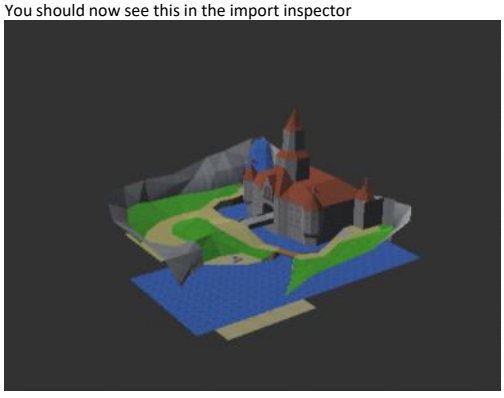
It's a start. At least the castle is right and the Mario is there. Still.



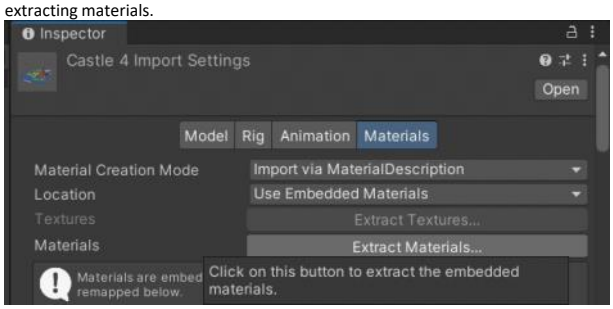


Let's now address the Castle problem, which is the more straight forward of the two. After all that work to fix it and everything, it still is a grey mess! And that is because it can't find the texture data. Unity is actually pretty good about sourcing this, so all you have to do is drag all the texture data (which is unhelpfully named as alphanumeric strings) into the project.

To fix the issue with the castle not having the textures assigned perform the following steps:

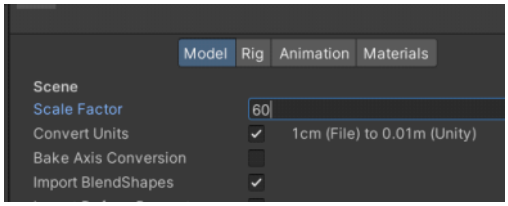
1. Delete the *.fbx in the Unity Project* if you already have it in the Unity project.
2. Put the textures of the original folder into a unity folder marked 'Castle Textures'
3. Re-import the .fbx of the castle.
4. You should now see this in the import inspector
5. 

Alright and then we'll do one more step for ourselves which will come in handy later and that is extracting materials.



In the inspector window on the model in the file browser, you can select this materials tab and click to extract the materials in a folder of your choice. I recommend making a folder marked "Castle Materials". Then click apply at the bottom.

As a final final step, change the model scale to about 60 again apply the settings at the bottom.



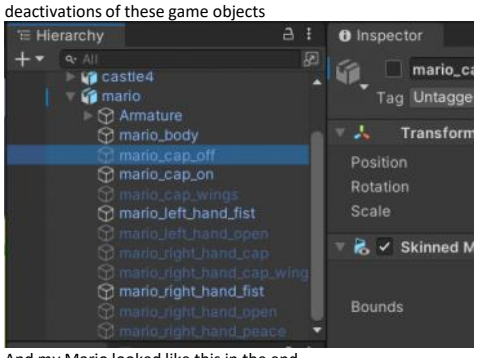
You can now drag the castle model into the scene. Let's do Mario now.

Do the following steps for Mario

1. Set his scale factor to .1
2. Extract the materials to a folder named 'Mario Materials'
3. Drag mario into the scene and place him wherever you like. I chose near the bridge

You should have something like this.

Next you need to clear out some of his unnecessary game objects. For me I did the following deactivations of these game objects



And my Mario looked like this in the end

Still not quite right but getting there.

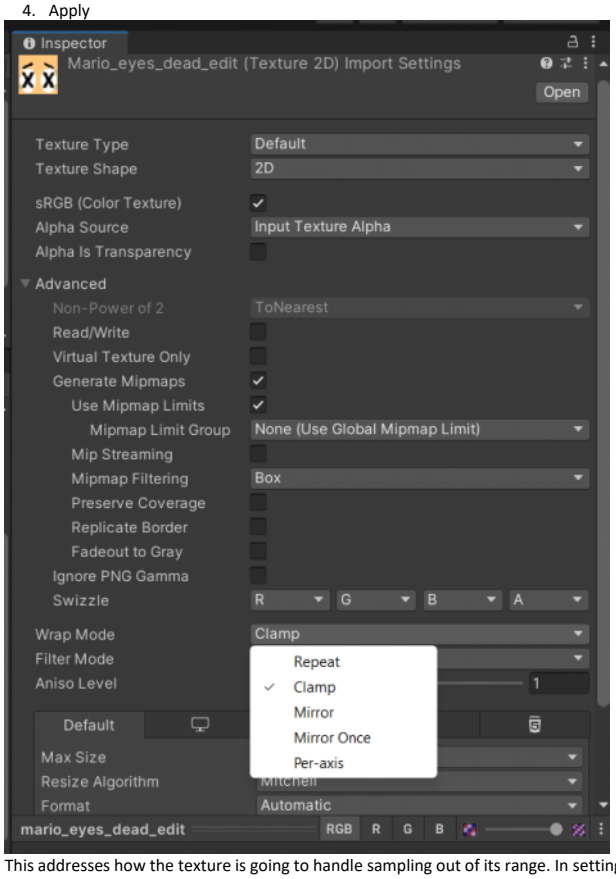
Adjusting Materials

We need to give him textures! Fortunately, his pack came with everything necessary, go ahead and import those files from the zip folder into the project. Again into a mario textures folder

The game we are going to play now is to assign the textures into the right location. First we need to do a quick update to our textures and that is set them to "clamp" rather than "repeat".

1. Select all the textures you just imported in the Mario Textures folder

2. Navigate to the Repeat Mode in the Inspector
3. Change it from Repeat to Clamp
4. Apply

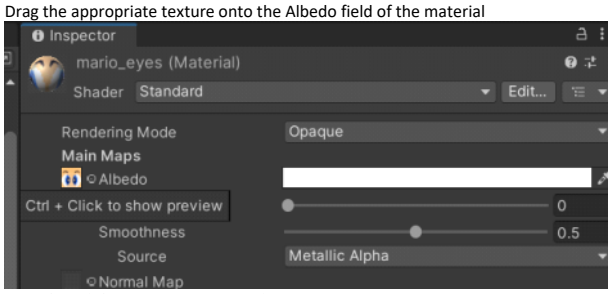


This addresses how the texture is going to handle sampling out of its range. In setting it to clamp, it will simply repeat the edge value endlessly rather than loop back.

We are ready now to start fixing these materials. We need to figure out which of the materials are showing these blank colors. And to do that we could go through all the materials or click on the part of the model that is showing blank. Let's simply go through the materials that are broken for simplicity

1. Mario_eyes
2. Mario_face
3. Mario_hair_sideburn
4. Mario_m_logo
5. Mario_mustache
6. Mario_yellow_button

Each of these is missing its appropriate texture. Do the following

1. Select the material in the project window
2. Drag the appropriate texture onto the Albedo field of the material
3. 

The Materials and their textures you need to assign

Mario_eyes	Mario_eyes_center_edit
Mario_face	skin
Mario_hair_sideburn	Mario_sideburn_edit
Mario_m_logo	Maroi_logo_edit
Mario_mustache	Mario_mustache_edit
Mario_yellow_button	Mario_overalls_button_edit

Aaannnnddddd tada!!

Wait a second what is wrong with his face? It turns out his default face material is set to have an Albedo color of vec3(.99,.54,.19) or the approximation of his current value of RGB(253,136,49).

This Albedo color is a multiplication applied to the texture color.

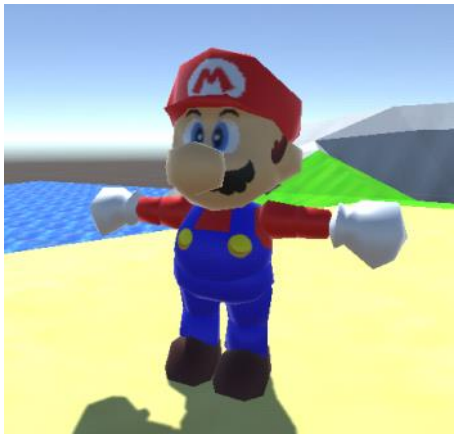
In GLSL terms that you're familiar, this tinting is like the following

Vec3 textureColor = texture2D(face_tex,uv).rgb;
Vec3 albedo = vec3(.99,.54,.19);

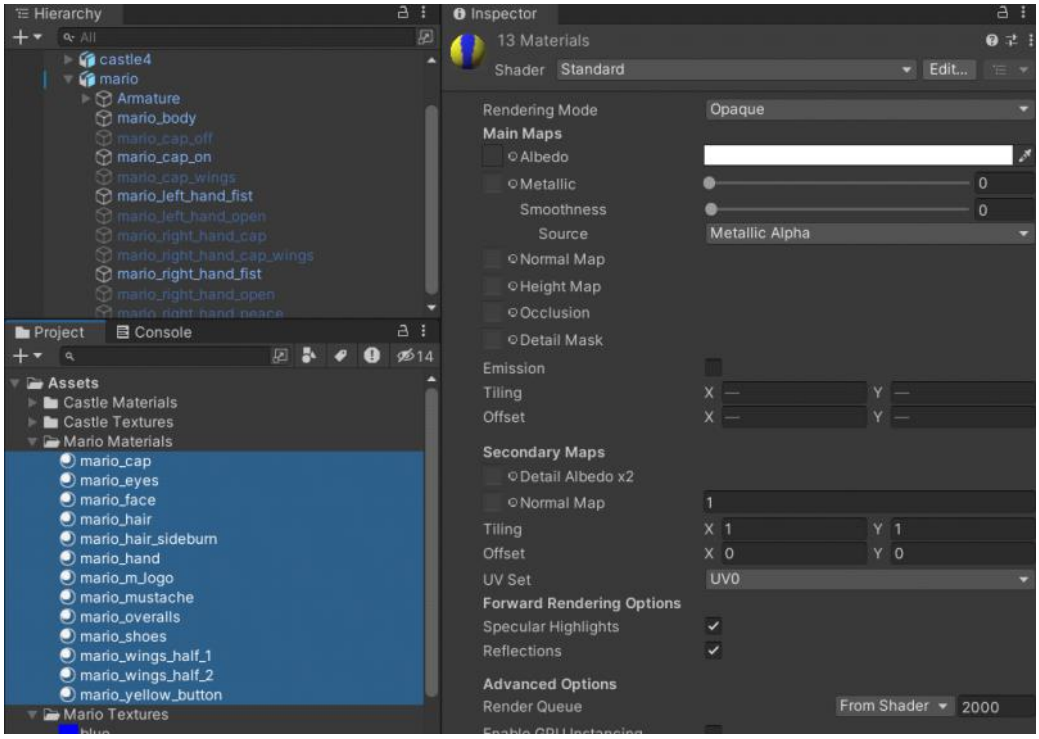
Vec3 result = textureColor * albedo;

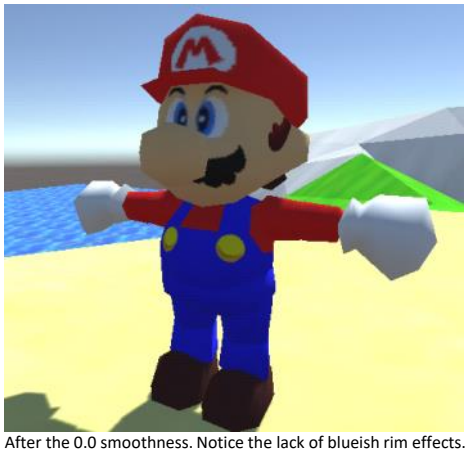
However all we want is the actual color of the texture. Can you figure out what the albedo color is supposed to be then? Take a second to think about this. And try a few colors until you get it right.

After figuring out the right color, you will get this!



Looking very Mario esque. But something is still a little off compared to the original and it has to do with the lighting model. He's a little… shiny. To remove this, we will select all of his materials and set the smoothness to 0.0.





After the 0.0 smoothness. Notice the lack of blueish rim effects.

Finalizing our Setup

And because the original Nintendo 64 did not have any smoothness, you should do the same for the castle materials as well if they are not already 0 smoothness.

And after all this, at last we have our basic scene!!! Move the camera around and take a snap shot.

It's just like Nintendo 64 (ignore the skybox). Congratulations! You've successfully imported and adjusted models and textures. And you did it without touching a shader once!

Wait a minute… what about the shaders you're asking me? You say this is all well and good but I've largely only walked you through the model import and troubleshooting process. Where are the shaders you ask?

PART 2 UNITY RENDERING

Graphics Rendering

The obvious fact now is that we've done all this work and we still haven't talked about shaders. It turns out that this is all the stuff surrounding shaders that we haven't touched yet. Shaders need surfaces to work on, and so there's no way to just ignore that requirement. Speaking shaders includes all the other stuff too. Luckily, everything prior to this point can be considered a kind of boilerplate. It's repeatable and you're likely to repeat it when working with other assets. The custom work of making your own shaders is something that comes in *after* the work of importing and adjusting assets. It's however, extremely important to get these other things out of the way precisely so that you can work unimpeded on your shader work because they're all connected. Your 3D models impact how your shaders look, and so too does the compositor (Blender vs Unity) that you use.

Without further ado, let's talk about the Unity shading model.

A Graphics Engine
Unity's default renderer is a forward rendering shading model. And to render an image, the graphics engine will do the following steps.

1. Take all the geometry of a particular camera perspective
2. Collect any rendering data, such as time or lighting data
3. Run the vertex shaders to manipulate the vertex as needed
4. Transform the geometry into a flat plane using matrix multiplication
5. Raster these into pixels on the flat surface
6. Run the shaders associated with those pixels in the fragment step
7. Take the result of that fragment shader step and throw it onto your screen

This is the expanded version (but still a simplified version of Graphics pipeline - UWP applications | Microsoft Learn) of what we saw in the Nvidia-Mythbusters demo. And thus far, we've been exclusively looking at that step 6. To fully manipulate and set up a scene in Unity, we need to start before step 1 (setting up geometry for the scene in question), which is why the shading hasn't been discussed until now. Perhaps luckily again, steps 1, 2, 4, 5, and 7 are all automatic unless you start making your own pipeline. That leaves us to care about 3 and 6, the actual fragment steps.

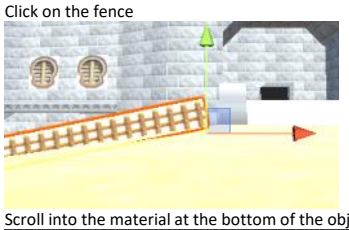
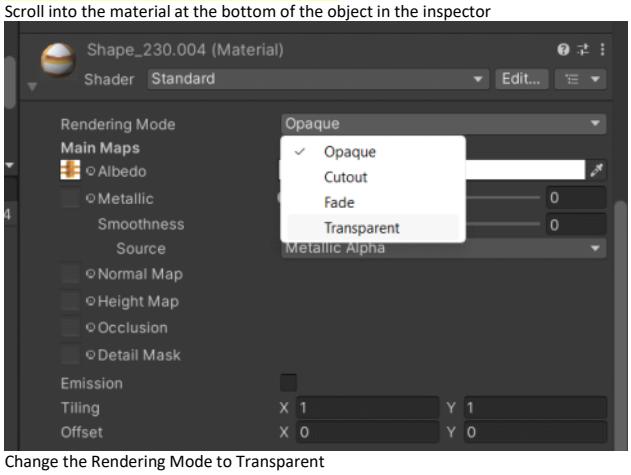
The eagle eyed have probably noticed some strange artifacts in our scene. Peach's glass window and the fences have this strange white texture look that we need to fix. We'll work our way down into the shaders from the top down, and slowly adjust features in our scene.

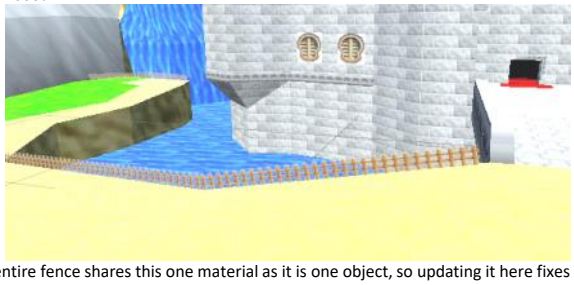
Material Matters

To fix these issues, we need to modify the container of the shader, known as a Material. In most rendering platforms, the material is a piece of data that contains references to a shader file that is used by the rendering engine for efficiency. This middle man exists for efficiency: multiple objects can refer to

a single material (and thus shader program) when we pass it through to the shader cores. This reduces the overall memory load on the system. When you modify one material, it modifies all the uses of it and when you modify the shader, it changes all the materials. You're all probably pretty familiar with that already.

The issue we're seeing is caused by the way the shader is rendering the texture. It's currently set to Opaque rendering. We'll set it to transparent.

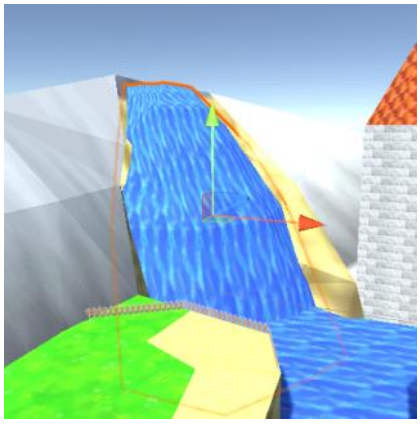
1. Click on the fence
2. 
3. Scroll into the material at the bottom of the object in the inspector
4. 

5. Change the Rendering Mode to Transparent
6. Tada!
7. 

The entire fence shares this one material as it is one object, so updating it here fixes it entirely.

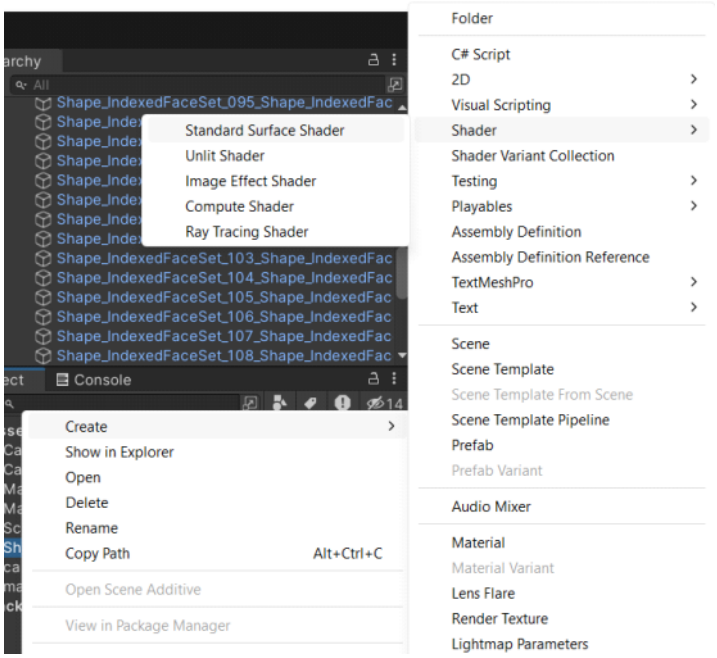
Next you should do this for Princess Peach and the Shutter objects on the sides of the castle.

Alright so that's models and materials both discussed, we are now finally ready to get into the shaders. And to do so with a practical example, we will build an effect for this water feature in the left-rear of the castle



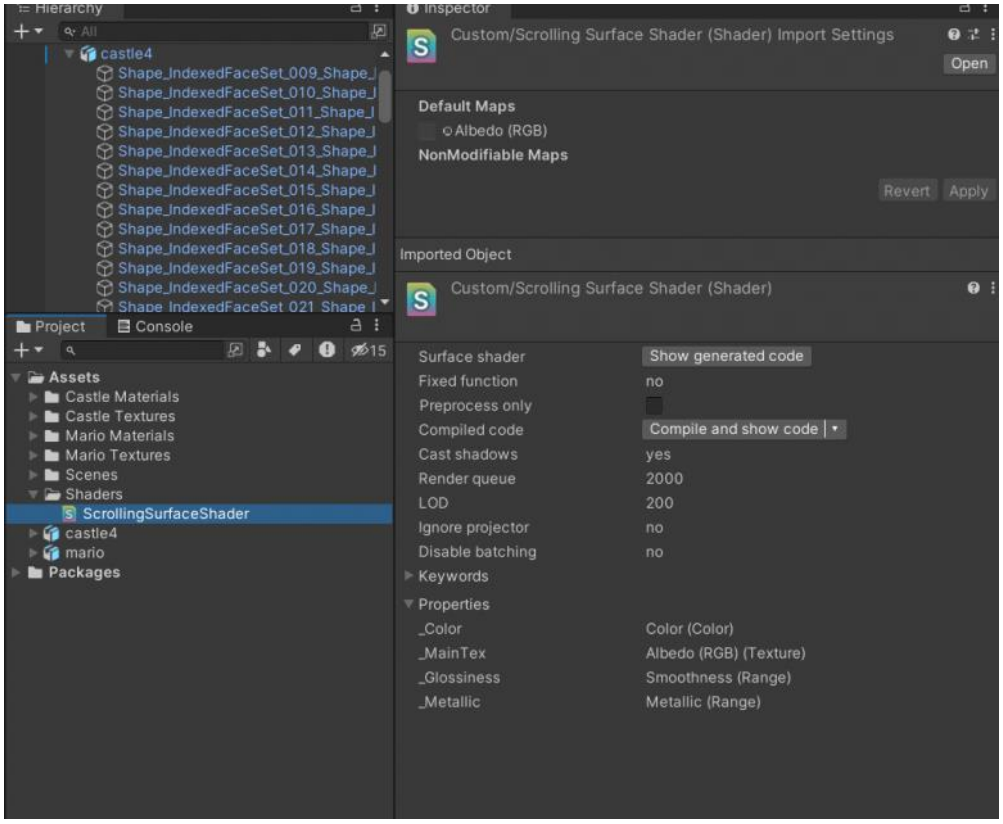
Writing your first Unity Surface Shader

What's in a Unity Shader you ask? We'll open one now. In your project window in Unity, right click and find Create > Shader > Surface Shader



Go ahead and add this file to a folder marked Shaders. Name it ScrollingSurfaceShader

Let's open this up and dive in ***finally***



The top should say Shader "Custom/ScrollingSurfaceShader" if not, go ahead and change it here. This name will be important later

Shader Lab

ScrollingSurfaceShader.shader

```
Shader "Custom/ScrollingSurfaceShader"
{
    Properties
    {
        _Color ("Color", Color) = (1,1,1,1)
        _MainTex ("Albedo (RGB)", 2D) = "white" {}
        _Glossiness ("Smoothness", Range(0,1)) = 0.5
        _Metallic ("Metallic", Range(0,1)) = 0.0
    }
    SubShader
    {
        Tags { "RenderType"="Opaque" }
        LOD 200

        CGPROGRAM
        // Physically based Standard lighting model, and enable shadows on all light types
        #pragma surface surf Standard fullforwardshadows

        // Use shader model 3.0 target, to get nicer looking lighting
        #pragma target 3.0

        sampler2D _MainTex;

        struct Input
        {
            float2 uv_MainTex;
        };

        half _Glossiness;
        half _Metallic;
        fixed4 _Color;
```

Welcome to Unity's ShaderLab format. ShaderLab is Unity's method of creating a shared shader development API-esque layer for you to build your own shaders. It is fairly well documented here Unity - Manual: ShaderLab (unity3d.com) But I'll walk you through the main components (which you can also read about here Introduction to Shader in Unity (hackingwithunity.com)). Now there is a lot of stuff, but

you can thankfully ignore most of it given the context of what we've discussed before. And then you can gradually add more detail to your understanding as we go along.

First of all, the shader is broken up into multiple parts. Properties & SubShaders. The properties declare all the variables that you'll need to modify for the material. The subshader defines multiple passes for the shader to create a composite effect (multi-pass shaders). Compared to our work in p5js, the properties are all those uniform declarations while subshaders contain ***both*** the .vert and .frag files that we used to use. This .shader file we have contains all of this now. We'll just use the one subshader for this one. Subshaders are where you'll see the familiar stuff from other shaders

Let's first just add the effect we want: a scroll for the water!

Navigate down to the surf function. Surf is short for surface. This function describes how the surface of the material should be rendered.

```
void surf (Input IN, inout SurfaceOutputStandard o)
{
    // Albedo comes from a texture tinted by color
    fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
    o.Albedo = c.rgb;
    // Metallic and smoothness come from slider variables
    o.Metallic = _Metallic;
    o.Smoothness = _Glossiness;
    o.Alpha = c.a;
}
```

You can probably guess what we have to do here. Tex2D is the HLSL version of the GLSL texture2D function we used before. This tex2D function is the same. So let's make it scroll by adding a float2 to the uv.

Input IN is the struct from above, this is automatically filled by the surface shader in Unity. We want to scroll by the y direction, so let's do the intuitive thing and add time to the float2 in the y dimension.

All you have to do to change this value is this IN.uv_MainTex + float2(0., _Time.y)

In Shaderlab, you will have access to a few global parameters. _Time is one of them. It's conveniently broken up into 4 values, Time.x is time/20, Time.y is time, Time.z is time * 2 and Time.w is time * 3. So we can use _Time.y;

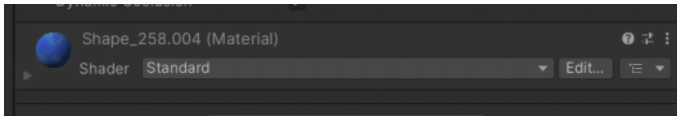
Save this change

```
void surf (Input IN, inout SurfaceOutputStandard o)
{
    // Albedo comes from a texture tinted by color
    fixed4 c = tex2D (_MainTex, IN.uv_MainTex + float2(0.,_Time.y)) * _Color;
    o.Albedo = c.rgb;
    // Metallic and smoothness come from slider variables
    o.Metallic = _Metallic;
    o.Smoothness = _Glossiness;
    o.Alpha = c.a;
}
```

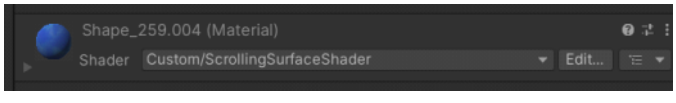
Shader is changed but now we need to actually update the material!

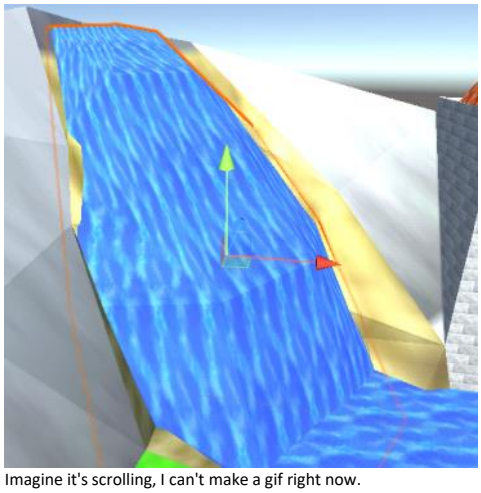
Assigning a Shader

Update the material by clicking the object and navigating to the material settings.



And here you'll change the Shader dropdown from Standard to Custom > ScrollingSurfaceShader. This process will automatically update the properties that match between the two shaders so the texture will be maintained. And you'll see the scrolling effect if you hit play!





Imagine it's scrolling, I can't make a gif right now.

But it's scrolling the wrong direction. This is because of *another* modeling quirk. The UVs on this model are set to go horizontally. So we need to update the scrolling offset vector to horizontal (in the x direction).

```
// Albedo comes from a texture tinted by color
fixed4 c = tex2D (_MainTex, IN.uv_MainTex + float2(_Time.y,0.)) * _Color;
o.Albedo = c.rgb;
```

Save this and notice your effect will automatically update even if you haven't left playmode. This is because shaders are immediately compiled and set to the graphics pipe. Updating them completely separately from the interactive code (a unity quirk many of you are probably familiar with).

Woohoo! Now it scrolls correctly. Go ahead and modify the rest of the water to use the same shader.

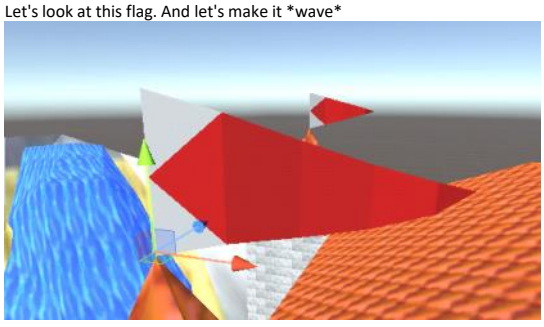
Again please imagine that it is scrolling. You have now made a custom shader effect.

That's the really Simple version. Questions about "what is a surf" "where is the fragment" "what about lighting" "how do shadows work" are all for further documentation and research.

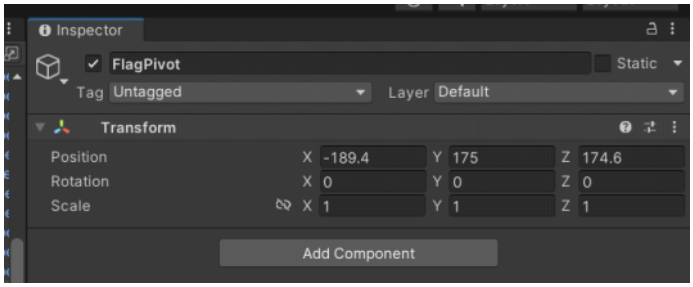
Making a vertex animated shader

What we'll do next is to build a more complex shader that will teach us how to set a variable from the C# side, then update the value in the shader, and we'll do it with a vertex effect.

Let's look at this flag. And let's make it *wave*



First we'll create an empty game object and place it here. Make sure it has no parent.



This is the rough approximation of the location for the "flag pole".

And now what we'll do is some C# coding.

Add this script component to the FlagPivot

```
1 asset usage
public class MaterialSetter : MonoBehaviour
{
    [SerializeField] private List<Material> materialsToSet;  Serializable

    Event function
    void Start()
    {
        foreach (Material mat in materialsToSet)
        {
            mat.SetVector( name: "_FlagPivot", (Vector4) transform.position);
        }
    }
}
```

This doesn't work as is, we'll need to create the shader that does.

1. Create a new surface shader "WavingSurfaceShader"
2. Add this new Vector property to the shader
3.
```
Properties
{
    _FlagPivot ("Flag Pivot", Vector) = (0,0,0,0)
    _Color ("Color", Color) = (1,1,1,1)
    _MainTex ("Albedo (RGB)", 2D) = "white" {}
    _Glossiness ("Smoothness", Range(0,1)) = 0.5
    _Metallic ("Metallic", Range(0,1)) = 0.0
}
```
3. Add the new property to the subshader
4.
```
float4 _FlagPivot;
half _Glossiness;
half _Metallic;
fixed4 _Color;
```
5. Add a new pragma step `#pragma vertex vert` this declares that this shader also has a vertex function
    a. These are special lines of code the define the behavior of the program to the compiler
    b.
```
CGPROGRAM
// Physically based Standard lighting model, and enable shadows on all light types
#pragma surface surf Standard fullforwardshadows
#pragma vertex vert
```
6. Create this vertex function

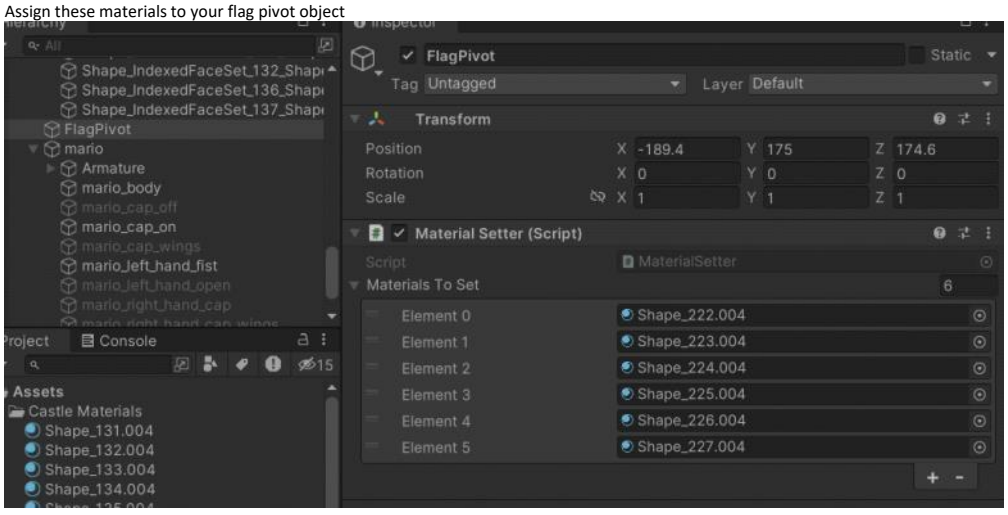
a.

```
UNITY_INSTANCING_BUFFER_START(Props)
    // put more per-instance properties here
UNITY_INSTANCING_BUFFER_END(Props)

void vert(inout appdata_full v)
{
    //World position of the vertex
    float4 world = mul(unity_ObjectToWorld, v.vertex);

    //distance from the flag pivot in pure x value
    float dist = abs(_FlagPivot.x - world.x);

    //change amplitude based on distance from pivot
    float amplitude = 0.11 * dist;

    //offset vertex in the z direction based on the sine wave
    world.z += amplitude * sin(_Time.y + world.x);

    //Set the vertex in v to the new position
    v.vertex = mul(unity_WorldToObject, world);
}

void surf (Input IN, inout SurfaceOutputStandard o)
{
    // Albedo comes from a texture tinted by color
```

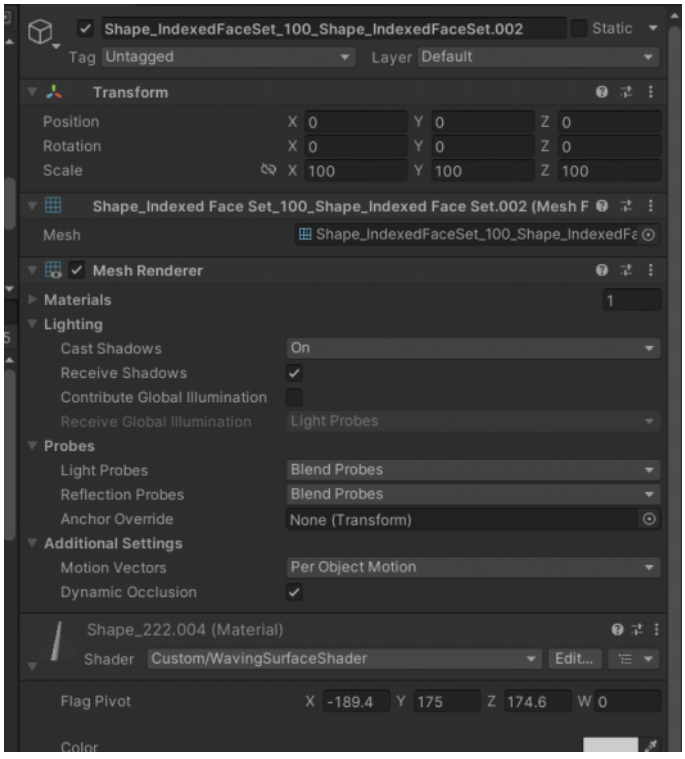
b. This is a vertex function so the purpose of this function is to modify the vertex data.
c. Inout declares that the appdata_full variable v will be sent into this function and sent back out to the fragment step
d. Appdata_full is a special data structure that Unity uses has a full set of internal data.

Next up we'll setup the FlagPivot game object

Assign these materials to your flag pivot object

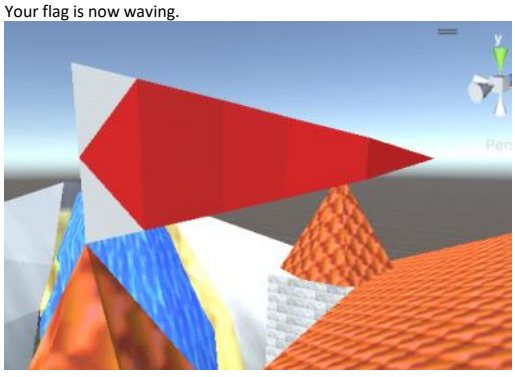


And then in the objects that make up the flag, set the shaders for these materials to the new shader that you just made "WavingSurfaceShader"



And then hit play!

Your flag is now waving.



Alright now we've gone and manipulated the flag in shader!

If you found this whole process painstaking, it's because it is. Welcome to tech art. You're initiated now. I promise it gets easier.

Part 3 PBR

What exactly are we making here? What is a surface shader? What happened to the fragment or vertex shaders before? Pragma?

Unity's standard shading model is based on a Physically Based Rendering (PBR) shading model. That means there are surfaces and those surfaces are lit by the lights in a scene in some way. That is the very broad idea here. This model (as in mathematical, not art asset) is the guiding principle behind how the engine chooses to show a surface to you.

Recall that process we discussed way back for the engine to render an image. This PBR model that Unity uses also has this surface step (surf) that determines a set of surface properties. Those properties are then hit with a lighting equation of some kind to then create the final color. This is why we had to set our Smoothness property from .5 to 0.0. This adjustment told the lighting equations that the surface should absorb all the light and thus show a solid, non reflecting color.

This is important because Light, as we all know, is fake. If your time with shaders has shown you anything so far, it' sthat color is just something you manipulate with math. This is fundamental to how we simulated life in real time graphics. A graphics pipeline will typically have a specific lighting model (again, as in mathematical) that it supports. This model specifies how light should be calculated and how it is reflected (a pun, but sometimes literally) in the final result. These describe how light should bounce off of the surface and be colored by it. You will find multiple kinds of these models out in the wild; they are typically named for the mathematician who describe or published them. They're typically not named for how they work or what effect they have.

Just to name a few:

- Lambertian
- Phong
- Blinn-Phong
- Minnaert
- Cook-Torrance
- Beckmann
- GGX
- Oren-Nayar
- "Cell"

These are all kinds of bidirectional reflectance distribution functions, or BRDFs. Combined with their sibling, the bidirectional transmittance distribution function (BTDF), we get the bidirectional scattering distribution function (BSDF). These classes of algorithms are used for lighting simulation.

Our surface function is a way of determining the surface values *prior* to sending it off to the lighting step. This involves a surface output O that contains the necessary values for calculation.

If you want more detail you can read about it Unity - Manual: Standard Shader (unity3d.com) but notably, Unity's shading is a composite of multiple lighting models

> The Standard Shader lighting math implementation uses the Disney model for diffuse component, GGX model for specular, with Smith Joint GGX visibility term and Schlick Fresnel appromixation.

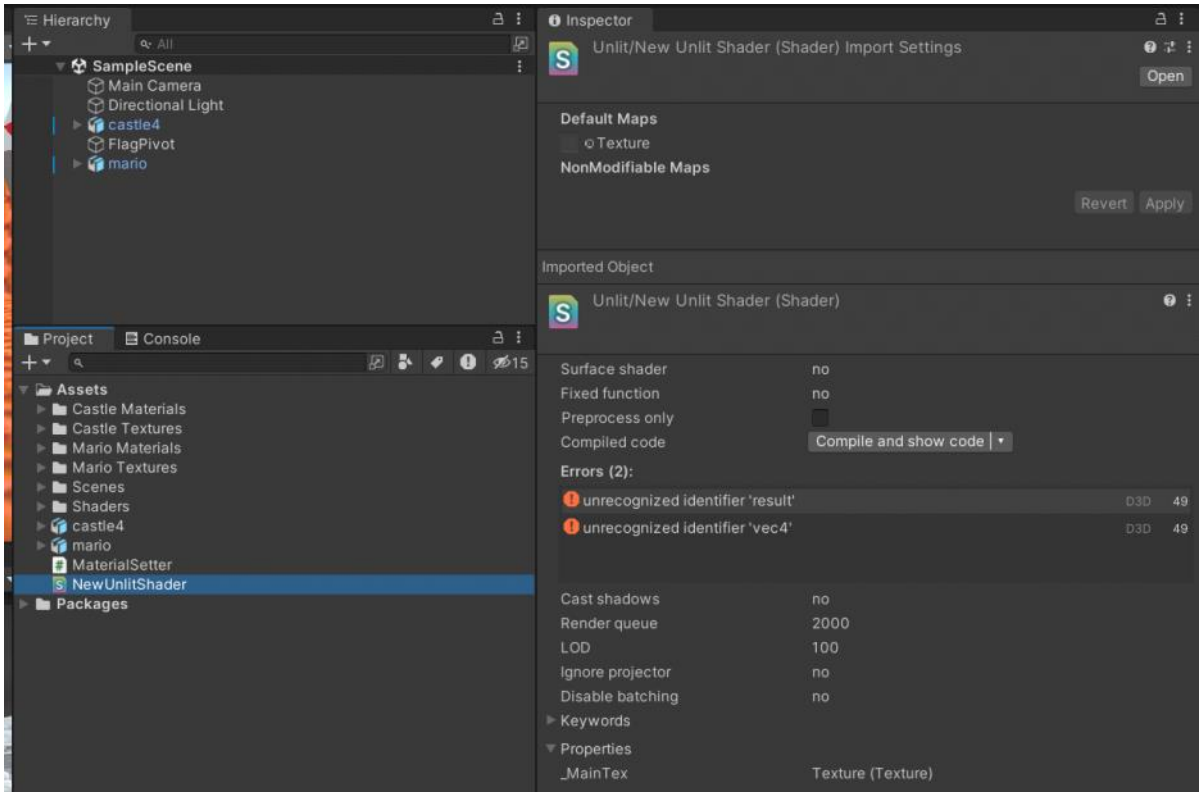
When it comes to making your own shader, know that your shader, if lit by the lighting model in the built-in pipeline, you will have the shader create some kind of surface detail for the lighting model. The is smoothness, metalness, diffuse color, bumps, displacement, and more.
Unity - Manual: Make your own (unity3d.com)

Unity basically requires us to use this format in order for our shaders to be lit in any way. If you don't need that then you you can use the unlit shader.

With what you've seen so far, the unlit shader is far more familiar. Below you will see the pragma declarations for the vert and frag functions. And instead of varyings, you'll see that the vert function returns a struct of v2f which is then input into the frag function as i. Why didn't with this? Because it's actually really nothing that new besides the Unity ShaderLab components (which you can largely ignore if you are writing one of these).

```
Pass
{
    CGPROGRAM
    #pragma vertex vert
    #pragma fragment frag
    // make fog work
    #pragma multi_compile_fog

    #include "UnityCG.cginc"

    struct appdata
    {
        float4 vertex : POSITION;
        float2 uv : TEXCOORD0;
    };

    struct v2f
    {
        float2 uv : TEXCOORD0;
        UNITY_FOG_COORDS(1)
        float4 vertex : SV_POSITION;
    };

    sampler2D _MainTex;
    float4 _MainTex_ST;

    v2f vert (appdata v)
    {
        v2f o;
        o.vertex = UnityObjectToClipPos(v.vertex);
        o.uv = TRANSFORM_TEX(v.uv, _MainTex);
        UNITY_TRANSFER_FOG(o,o.vertex);
        return o;
    }

    fixed4 frag (v2f i) : SV_Target
    {
        // sample the texture
        fixed4 col = tex2D(_MainTex, i.uv);
        // apply fog
        UNITY_APPLY_FOG(i.fogCoord, col);
        return col;
    }
    ENDCG
}
```

Go ahead and copy and paste any of your existing shaders (I would start with a simple one) into this fixed4 frag function.
You will get a bunch of errors. But where do you see them? Here in this project window



Yes it is arguably even *less* convenient than p5js' error logging but at least it's here.

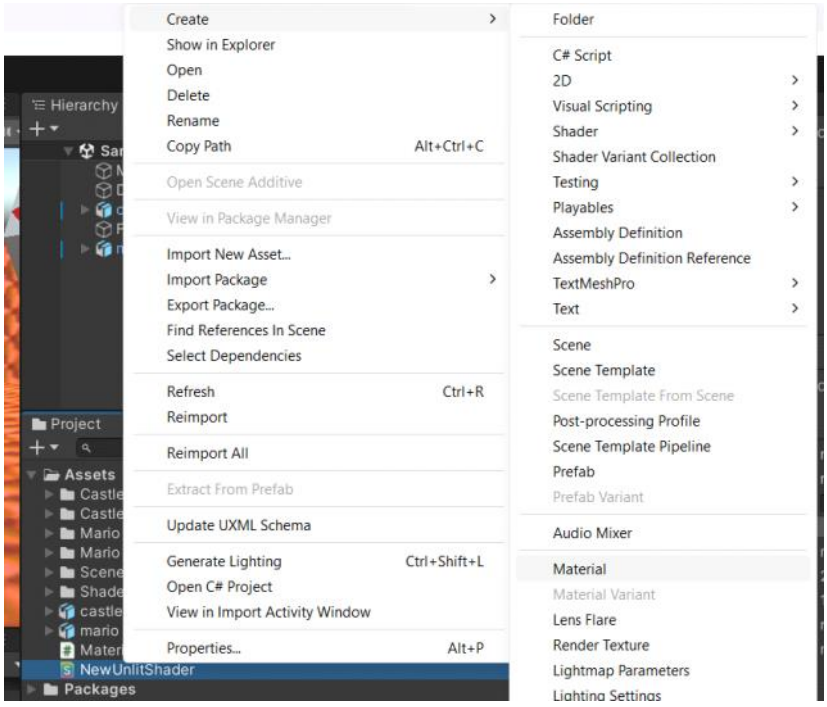
For an example let's look at this fragment function.

```
fixed4 frag (v2f i) : SV_Target
{
    float4 result = float4(i.uv,0.,1.);
    fixed4 col = result;
    return col;
}
ENDCG
```

I've gone ahead and made a simple unlit shader that shows our UV square. Notice the change of types from what you're used to. You will probably need to refer to this document [GLSL-to-HLSL reference - UWP applications | Microsoft Learn]

But for the most part vec types become float + number and that's about it. But the functions are sometimes different too. These are things like fract -> frac too. For these I advise you to just look up the HLSL equivalents (usually the name plus hlsl will be enough).

Once you've made the adjustments to the shader, you still need to make a material. To do that, right click on the shader file > Create > Material will automatically create one for you.



Aaaand tada!

From here it's your world for importing and getting shaders in with more complicated effects.

In the sense of what we've done so far. The textures from last week were really the last step before moving onto this, which now incorporates the full range of 3D modelling.

Good luck to us, we're in the thick of it now.

There's a lot of documentation scattered about ShaderLab: Anatomy of a Shader - Unity Learn so we'll try and get through this as best as we can.

Properties
This is where you will define publicly available and code manipulatable fields for the shader. The documentation for this is here
Unity - Manual: ShaderLab: Properties (unity3d.com)

Homework

For next week, I'd like you to prepare the following

1. Import at least 1 more model from Mario 64 into the scene. Get their textures and normals fixed as needed.
2. Create a new surface shader with some other kind of visual effect on the surface, you do not have to use a vertex function, but a surface function is enough.
    a. Consider translating the texture coordinates or mixing another texture on top (declare a new 2D property to do this)
    b. Try doing a sin wave effect on the vertices of the water to make it have a wave
    c. Try creating an SDF in the surface and have it change the Albedo!
3. Create a new Unlit Shader and set it to one of the surfaces. Bring over one of your shaders from previous lessons! Try a noise function or similar. And apply it as a material to a quad or any other surface

And differently from other weeks, I'd like you all to bring at least 2 questions to class next week for us to

discuss. It can be literally anything that this document has made you think about or you want to know more about.

Extra Notes on ShaderLab

The basic idea of the shaderlab format is to combine the vertex and fragment stages into a single shader file and combine it with a place for where to declare properties.

[VariableNameForSubShader] ("[String Name For Inspector]", [data type]) = [initial value]

```
Properties
{
    _Color ("Color", Color) = (1,1,1,1)
    _MainTex ("Albedo (RGB)", 2D) = "white" {}
    _Glossiness ("Smoothness", Range(0,1)) = 0.5
    _Metallic ("Metallic", Range(0,1)) = 0.0
}
SubShader
```

These are the equivalent of our uniforms from before. And like uniforms, you need to redeclare them in the subshader section.

```
SubShader
{
    Tags { "RenderType"="Opaque" }
    LOD 200

    CGPROGRAM
    // Physically based Standard lighting model, and enable shadows on all light typ
    #pragma surface surf Standard fullforwardshadows

    // Use shader model 3.0 target, to get nicer looking lighting
    #pragma target 3.0

    sampler2D _MainTex;

    struct Input
    {
        float2 uv_MainTex;
    };

    half _Glossiness;
    half _Metallic;
    fixed4 _Color;
```

Note especially how the _MainTex is noted as a 2D in the properties above but is a sampler2D down here. Also note the other change: fixed4 and half. Welcome to HLSL from DirectX [High-level shader language (HLSL) - Win32 apps | Microsoft Learn] (keep this documentation handy)

So we are officially programming in a new shading language. OpenGL and HLSL are very similar where it counts but have some key differences. One of those is data types. [Data Types (HLSL) - Win32 apps | Microsoft Learn]

In this format you need to declare your scalars (ints and floats) separately from their vector forms.