HLSL-Base

Shader code is written using the High Level Shading Language (HLSL) in Unity .

HLSLPROGRAM & HLSLINCLUDE

Inside each ShaderLab Pass , we define blocks for HLSL code using HLSLPROGRAM and ENDHLSL tags . Each of these blocks must include a Vertex and Fragment shader . We use the `#pragma vertex/fragment` to set which function is going to be used .

For built-in pipeline shaders "vert" and "frag" are the most common names , but they can be anythings. For URP , it tends to use functions like "UnlitPassVertex" and "UnlitPassFragment" which is a bit more descriptive of what the shader pass is doing .

Inside the SubShader we can also use HLSLINCLUDE to include the code **in every Pass inside that SubShader** . This is very useful for writing shaders in URP as every pass needs to use the same `UnityPerMaterial CBUFFER` to have compatibility with the SRP Batcher and this helps us reuse the same code for every pass instead of needing to define it separately . We could alternatively use a separate include file instead too .

```
SubShader
{
    Tags {"RenderPipeline"="UniversalPipeline" "Queue"="Geometry"}

    HLSLINCLUDE
    ...
    ENDHLSL

    Pass
    {
        Name "Forward"
        // LightMode tag . Using default here as the shader is Unlit
        // Cull, ZWrite, ZTest, Blend, etc

        HLSLPROGRAM
        #pragma vertex UnlitPassVertex
        #pragma fragment UnlitPassFragment
        ...
        ENDHLSL
    }
}
```

We'll discuss the contents of these code block later . For now , we need to go over some basics of HLSL which is important to know to be able to understand the later sections .

Variables

In HLSL , we have a few different variable types , the most common consisting of Scalars , Vectors and Materials . There's also special objects for Textures/Samplers . Arrays and Buffers also exist for passing more data into the shader .

Scalar

The scalar types include :

- bool - true or false .
- float - 32 bit floating point number . Generally used for world space positions , texture coordinates , or scalar computations involving complex functions such as trigonometry or power/exponentiation .
- half - 16 bit floating point number . Generally used for short vectors , directions , object space positions , colours .
- double - 64 bit floating point number . Cannot be used as inputs/outputs , see note here .
- real - Used in URP/HDRP when a function can support either half or float . It defaults to half (assuming they are supported on the platform) , unless the shader specifies "#define PREFER_HALF 0" , then it will use float precision . Many of the common math functions in the ShaderLibrary functions use this type .
- int - 32 bit signed integer .
- uint - 32 bit unsigned integer (except GLES2 , where this isn't supported , and is defined as an int instead) .

Also of notes :

- fixed - 11 (ish) bit fixed point number with -2 to 2 range . Generally used for LDR colours . Is something from the older CG syntax , though all platforms seem to just convert it to half now even in CGPROGRAM . HLSL does not support this but I felt it was important to mention as you'll likely see the "fixed" type used in shaders written for the Built-in RP , use half instead !

Vector

A vector is created by appending a component size (integer from 1 to 4) to one of these scalar data types . Some examples include :

- float4 - (A float vector containing 4 floats)
- half3 - (A half vector , 3 components)
- int2 , etc
- Technically float1 would also be a one dimensional vector , but as far as I'm aware it's equivalent to float .

In order to get one of the components of a vector , we can use .x , .y , .z , or .w (or .r , .g , .b , .a instead , which makes more sense when working with colours) . We can also use .xy to obtain a vector2 and .xyz to obtain a vector3 from a higher dimensional vector .

We can even take this further and return a vector with components rearranged , which is referred to as swizzling . Here is a few examples :

```
float3 vector = float3(1, 2, 3);

float3 a = vector.xyz; // or .rgb,  a = (1, 2, 3)
float3 b = vector.zyx; // or .bgr,  b = (3, 2, 1)
float3 c = vector.xxx; // or .rrr,  c = (1, 1, 1)
float2 d = vector.zy;  // or .bg,   d = (3, 2)
float4 e = vector.xxzz; // or .rrbb, e = (1, 1, 3, 3)
float f = vector.y; // or .g,  f = 2

// Note that mixing xyzw/rgba is not allowed .
```

Matrix

A matrix is created by appending two sizes (integers between 1 and 4) to the scalar , separated by an "x" . The first integer is the number of **rows** , while the second is the number of **columns** in the matrix . For example :

- `float4x4` - 4 rows , 4 columns
- `int4x3` - 4 rows , 3 columns
- `half2*1` - 2 rows , 1 column
- `float1x4` - 1 row , 4 columns

Matrix are used for transforming between different spaces . If you aren't very familiar with them , I'd recommend looking at this tutorial by CatlikeCoding .

Unity has built-in transformation matrices which are used for transforming between common spaces , such as :

- `UNITY_MATRIX_M` (or `unity_ObjectToWorld`) - **Model** Matrix , Converts from Object space to World space .
- `UNITY_MATRIX_V` - **View** Matrix , Converts from world space to View space
- `UNITY_MATRIX_P` - **Projection** Matrix , Converts from View space to Clip space
- `UNITY_MATRIX_VP` - **View Projection** Matrix , Converts from World space to Clip space .

Also inverse versions :

- `UNITY_MATRIX_I_M` (or `unity_WorldToObject`) - **Inverse Model** Matrix , Converts from World space to Object space
- `UNITY_MATRIX_I_V` - **Inverse View** Matrix , Converts from View space to World space
- `UNITY_MATRIX_I_P` - **Inverse Projection** Matrix , Converts from Clip space to View space
- `UNITY_MATRIX_I_VP` - **Inverse View Projection** Matrix , Converts from Clip space to World space

While you can use these matrices to convert between spaces via matrix multiplication (e.g. `mul(matrix, float4(position.xyz, 1))`) , there is also helper function in the SRP Core ShaderLibrary SpaceTransforms.hlsl .

Something to be aware of is when dealing with matrix multiplication , the order is important . Usually the matrix will be in the first input and the vector in the second . A Vector in the second input is treated like a Matrix consisting of up to 4 rows (depending on the size of the vector) , and a single column . A Vector in the first input is instead treated as a Matrix consisting of 1 row and up to 4 columns .

Each component in the matrix can also be accessed using either of the following : The zero-based row-column position :

- ._m00, ._m01, ._m02, ._m03
- ._m10, ._m11, ._m12, ._m13
- ._m20, ._m21, ._m22, ._m23
- ._m30, ._m31, ._m32, ._m33

The one-based row-column position:

- ._m11, ._m12, ._m13, ._m14
- ._m21, ._m22, ._m23, ._m24
- ._m31, ._m32, ._m33, ._m34
- ._m41, ._m42, ._m43, ._m44

The zero-based array access notation:

- [0][0], [0][1], [0][2], [0][3]
- [1][0], [1][1], [1][2], [1][3]

- [2][0], [2][1], [2][2], [2][3]
- [3][0], [3][1], [3][2], [3][3]

With the first two options , you can also use swizzling . e.g. `._m00_m11` or `._11_22` .

Of note , `._m03_m13_m23` corresponds to the translation part of each matrix . So `UNITY_MATRIX_M._m03_m13_m23` gives you the World space postion of the origin of the GameObject , (assuming there is no static/dynamic batching involved for reasons explained in my [Intro to Shaders post](#) .

Texture Objects

Textures store a colour for each **texel** - basically the same as a pixel , but they are known as texels (short for texture elements) when referring to textures and they also aren't limited to just two demensions .

The fragment shader stage runs on a per-fragment/pixel basis , where we can access the colour of a texel with a given coordinate . Textures can have different sizes (widths/heights/depth) , but the coordinate used to sample the texture is normalised to a 0-1 range . These are known as Texture Coordinates or UVs . (where U corresponds to the horizontal axis of the texture , while V is the vertical . Sometimes you'll see UVW where W is the third dimension / depth slice of the texture) .

The most common texture is a 2D one , which can be defined in URP using the following macros in the global scope (outside any functions) :

```
TEXTURE2D(textureName);
SAMPLE(sampler_textureName);
```

For each texture object we also define a [SampleState](#) which contains the wrap and filter modes from the texture's import settings . Alternatively , we can define an inline sampler , e.g. `SAMPLER(sampler_linear_repeat)` .

Filter Modes

- **Point** (or Nearest-Point) : The colour is taken from the nearest texel . The result is blocky/pixellated , but that if you're sampling pixel art you'll likely want to use this .
- **Linear / Bilinear** : The colour is taken as a weighted average of close texels , based on the distance to them .
- **Trilinear** : The same as Linear/Bilinear , but it is also blends between mipmap levels .

Wrap Modes

- **Repeat** : UV values outside of 0-1 will cause the texture to tile/repeat .
- **Clamp** : UV values outside of 0-1 are clamped , causing the edges of the texture to stretch out .
- **Mirror** : The texture tiles/repeats while also mirroring at each integer boundary .
- **Mirror Once** : The texture is mirrored once , then clamps UV values lower than -1 and higher than 2 .

Later in the fragment shader we use another macro to sample the Texture2D with a uv coordinate that would also be passed through from the vertex shader :

```
float4 color = SAMPLE_TEXTURE2D(textureName, sampler_textureName, uv);
// Note, this can only be used in fragment as it calculates the mipmap level used .
// If you need to sample a texture in the vertex shader, use the LOD version
// to specify a mipmap (e.g. 0 for full resolution) :
float4 color = SAMPLE_TEXTURE2D_LOD(textureName, sampler_textureName, uv, 0);
```

Some other texture types include : Texture2DArray , Texture3D , TextureCube (known as a Cubemap outside of the shader) & TextureCubeArray , each using the following macros :

```
// Texture2DArray
TEXTURE2D_ARRAY(textureName);
SAMPLER(sampler_textureName);
// ...
float4 color = SAMPLE_TEXTURE2D_ARRAY(textureName, sampler_textureName, uv, index);
float4 color = SAMPLE_TEXTURE2D_ARRAY_LOD(textureName, sampler_textureName, uv, lod);

// Texture3D
TEXTURE3D(textureName);
SAMPLER(sampler_textureName);
// ...
float4 color = SAMPLE_TEXTURE3D(textureName, sampler_textureName, uvw);
float4 color = SAMPLE_TEXTURE3D_LOD(textureName, sampler_textureName, uvw, lod);
// uses 3D uv coord (commonly referred to as uvw)

// TextureCube
TEXTURECUBE(textureName);
SAMPLER(sampler_textureName);
// ...
float4 color = SAMPLE_TEXTURECUBE(textureName, sampler_textureName, dir);
float4 color = SAMPLE_TEXTURECUBE_LOD(textureName, sampler_textureName, dir, lod);
// uses 3D uv coord (named dir here, as it is typically a direction)

// TextureCubeArray
TEXTURECUBE_ARRAY(textureName);
SAMPLER(sampler_textureName);
// ...
float4 color = SAMPLE_TEXTURECUBE_ARRAY(textureName, sampler_textureName, dir, index);
float4 color = SAMPLE_TEXTURECUBE_ARRAY_LOD(textureName, sampler_textureName, dir, lod);
```

Array

Arrays can also be defined , and looped through using a for loop . For example :

```
float4 _VectorArray[10];
float _FloatArray[10];

void ArrayExample(out float Out)
{
    float add = 0;
    [unroll]
    for(int i = 0; i < 10; i++)
    {
        add += _FloatArray[i];
    }
    Out = add;
}
```

If the size of the loop is fixed (i.e. not based on a variable) and the loop does not exit early , it can be more performant to "unroll" the loop , which is like copy-pasting the same code multiple times with the index changed .

It's technically also possible to have other types of arrays , however Unity can only set Vector(float4) and Float array from a C# script .

I also recommend to always set them **globally** , using `Shader.SetGlobalVectorArray` and/or `Shader.SetGlobalFloatArray` rather than using the `material.SetVector/FloatArray` versions . The reason for this is arrays cannot be properly included in the UnityPerMaterial CBUFFER (as it requires it to also be defined in the ShaderLab Properties , and arrays aren't supported there) . If the objects are batched using the SRP Batcher , multiple materials trying to use different arrays leads to glitchy behaviour where the values will change for all objects depending on what is being rendered on screen . By setting them globally , there can only ever be one array used which avoids this .

Note that these SetXArray methods are also limited to a maximum array size of 1023 . If you need larger you might need to try alternative solutions instead , e.g. Compute Buffers (StructuredBuffer) , assuming they are supported on the target platform .

Buffer

An alternative to arrays , is using [Compute Buffers] , which in HLSL is referred to as a **StructuredBuffer** (which is read-only . Alternatively there's **RWStructuredBuffer** for reading&writing but is only supported in pixel/fragment and compute shaders) .

You'd also need at lease `#pragma target 4.5` to use these . Not all platforms will support compute buffers too (and some might not support StructuredBuffer in vertex shaders) . You can use `SystemInfo.supportsComputeShaders` in C# at runtime to check if the platform supports them .

```
struct Example
{
    float3 A;
    float B;
};

StructuredBuffer<Example> _BufferExample;

void GetBufferValue(float Index, out float3 Out)
{
    Out = _BufferExample[Index].A;
}
```

And using this C# for setting it , as a test :

```csharp
using UnityEngine;

[ExecuteAlways]
public class BufferTest : MonoBehaviour
{
    private ComputeBuffer buffer;

    private struct Test
    {
        public Vector3 A;
        public float B;
    }

    private void OnEnable()
    {
        Test test = new Test
        {
            A = new Vector3(0f, 0.5f, 0.5f),
            B = 0.1f,
        };
        Test test2 = new Test
        {
            A = new Vector3(0.5f, 0.5f, 0f),
            B = 0.1f,
        };

        Test[] data = new Test[]{test, test2};

        buffer = new ComputeBuffer(data.Length, sizeof(float) * 4);
        buffer.SetData(data);

        GetComponent<MeshRenderer>().shaderMaterial.SetBuffer("_BufferExample", buffer);
    }

    private void OnDisable()
    {
        buffer.Dispose();
    }
}
```

I'm not super familiar with StructuredBuffers so sorry if this section is a bit lacking . I'm sure there are resources online that can explain it better!

Functions

Declaring functions in HLSL is fairly similar to C# , however it is important to note that you can only call a function if it's already been declared . You cannot call a function before declaring it so the order of functions and `#include` files matters !

```
float3 example(float3 a, float3 b)
{
    return a * b;
}
```

Here float3 is the return type , "example" is the function name and inside the brackets are the parameters passed into the function . In the case of no return type , `void` is used . You can also specify output parameters using `out` before the parameter type , or `input` if you want it to be an input that you can edit and pass back out . (There's also in but we don't need to write it) .

```
void example(float3 a, float3 b, out float3 Out)
{
    Out = a * b;
}
/* This might be more useful for passing multiple outputs , though they could also be packed int
```

You may also see `inline` before the function return type . This is the default and only modifier a function can actually have , so it's not important to specify it . It means that the compiler will generate a copy of the function for each call . This is done to reduce the overhead of calling the function .

You may also see functions like :

```
#define EXAMPLE(x, y) ((x) * (y))
```

This is referred to as a macro . Macros are handled before compiling the shader and they get replaced with the definition , with any parameters substituted . For example :

```
float f = EXAMPLE(3, 5);
float3 a = float3(1, 1, 1);
float3 f2 = EXAMPLE(a, float3(0, 1, 0));

// just before compiling this becomes :
float f = ((3) * (5));
float a = float(1, 1, 1);
float3 f2 = ((a) * (float3(0, 1, 0)));

// An important note , is that the macro has () around x and y .
// This is because we could do :
float b = EXAMPLE(1 + 2, 3 + 4);
// which becomes :
float b = ((1 + 2) * (3 + 4));// 3 * 7, so 21
// If those () wasn't included , it would instead be :
float b = (1 + 2 * 3 + 4)
// which equals 11 due to * taking precedence over +
```

Another macro example is :

```
#define TRANSFORM_TEX(tex, name) (tex.xy * name##_ST.xy + name##_ST.zw)

// Usage :
OUT.uv = TRANFORM_TEX(IN.uv, _MainTex)

// which becomes :
OUT.uv = (IN.uv.xy * _MainTex_ST.xy + _MainTex_ST.zw);
```

The ## operator is a special case where macros can be useful . It allows us to concatenate the name and `_ST` parts , resulting in `_MainTex_ST` for this input . If the ## part was left out , it would just produce `name_ST` , resulting in an error since that hasn't be defined . (Of course , _MainTex_ST still also needs to be defined , but that's the intended behaviour . Appending `_ST` to the texture name is how Unity handles the tiling and offset values for a texture) .

UnityPerMaterial CBUFFER

Moving onto actually creating the shader code , we should first specify the **UnityPerMaterial CBUFFER** inside a **HLSLINCLUDE** block inside the SubShader . This ensures the same CBUFFER is used for all passes , which is important for the shader to be compatible with the SRP Batcher .

The CBUFFER must include all of the **exposed** properties (same as in the Shaderlab Properties block) , except textures , though you still need to include the texture tiling & offset values (e.g. _ExampleTexture_ST , where S refers to scale and T refers to translate) and TexelSize (e.g. _ExampleTexture_TexelSize) if they are used .

It cannot include other variables that aren't exposed .

```
HLSLINCLUDE
    #include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Core.hlsl"

    CBUFFER_START(UnityPerMaterial)
    float4 _ExampleTexture_ST;
    float4 _ExampleTexture_TexelSize;// x = 1/width, y = 1/height, z = width, w = height
    float4 _ExampleColor;
    float _ExampleRange;
    float _ExampleFloat;
    float4 _ExampleVector;
    // etc.
    CBUFFER_END
ENDHLSL
```

Note : While variables don't have to be exposed to set them via the C# `material.SetColor / SetFloat / SetVector / etc` , if multiple material instances have different values , this can produce glitchy behaviour as the SRP Batcher will still batch them together when on screen . If you have variables that aren't exposed - **always** set them using `Shader.SetGlobalX` functions , so that they remain constant for all material instances . If they need to be different per material , you should expose them via the Shaderlab Properties block and add them to the CBUFFER instead .

In the above code we are also including **Core.hlsl** from the **URP ShaderLibrary** using the #include as shown above . This is basically the URP-equivalent of the built-in pipeline UnityCG.cginc . Core.hlsl (and other ShaderLibrary files it automatically includes) contain a bunch of useful functions and macros , including the `CBUFFER_START` and `CBUFFER_END` macros themselves , which is replaced with "cbuffer name {" and "};" on platforms that support them , (I think all except GLES2 , which makes sense as the SRP Batcher isn't supported for that platform too) .

Structs

Before we define the vertex or fragment shader functions we need to define some **structs** which are used to pass data in and out of them . In built-in it is common to create two named "appdata" and "v2f" (short for vertex to fragment") while URP shaders tend to use "Attributes" and "Varyings" instead . These are just names and usually aren't too important though , name them "VertexInput" and "FragmentInput" if you want .

The URP ShaderLibrary also uses some structs to help organise data needed for certain functions - such as InputData and SurfaceData which are used in lighting/shading calculations , I'll be going through those in the Lighting section .

Since this is a fairly simple Unlit shader our Attributes and Varyings won't be all that complicated :

Attributes (VertexInput)

```
struct Attributes
{
    float4 positionOS : POSITION;
    float2 uv : TEXCOORD0;
    float4 color : COLOR;
};
// Don't forget the semi-colon at the end of the struct here,
// or you'll get "Unexpected Token" errors!
```

The **Attributes** struct will be the input to the vertex shader . It allows us to obtain the per-vertex data from the mesh , using the strings (most of which are all in caps) which are known as semantics . This includes :

- *POSITION* : Vertex position
- *COLOR* : Vertex colour
- *TEXCOORD0-7* : UVs (aka texture coordinates) . A mesh has 8 different UV channels accessed with a value from 0 to 7 . Note that in C# , Mesh.uv corresponds to *TEXCOORD0* . Mesh.uv1 does not exist , the next channel is uv2 which corresponds to *TEXCOORD1* and so on up to Mesh.uv8 and *TEXCOORD7* .
- *NORMAL* : Vertex Normals (used for lighting calculations . This is unlit currently so isn't needed)
- *TANGENT* : Vertex Tangents (used to define "tangent space" , important for normal maps and parallax effects)

Varyings(FragmentInput)

```
struct Varyings
{
    float4 positionCS : SV_POSITION;
    float2 uv : TEXCOORD0;
    float4 color : COLOR;
};
```

The **Varyings** struct will be the input to the fragment shader , and the output of the vertex shader (assuming there's no geometry shader in-between , which might need another struct , but we aren't going though that in this post) .

Unlike the previous struct , we use *SV_POSITION* instead of *POSITION* , which stores the **clip space position** from the vertex shader output . It's important to convert the geometry to fragments/pixels on the screen at the correct location .

We also use the *COLOR* and/or *TEXCOORD0-7* semantics but unlike before don't have to correspond to the mesh vertex colors / uvs at all . Instead they are used to interpolate data across the triangle . *NORMAL/TANGENT* is typically not used in the Varyings struct , and although I have seen them still work (along with completely custom semantics) , I assume the compiler is converting them but that behaviour might not supported on all platforms so I'd stick to TEXCOORD to be safe .

Depending on the compile target (e.g. *#pragma target 3.0*) there can also be additional TEXCOORD interpolators that can be used . Shader Model 2 supports up to 8 of these interpolators (and two COLOR semantics , probably COLOR/COLOR0 and COLOR1 though I've never used that) . Shader Model 3 combined these so supports up to 10 (so just TEXCOORD0-9 , or COLOR & TEXCOORD0-8) . More interpolators might also be available in other targets but also might not be supported on all platforms . See the [Shader Compile Targets docs page] for more info.

FragmentOutput

The fragment shader can also provide an output struct . However it's usually not needed as it typically only uses a single output semantic , `SV_Target` , which is used to write the fragment/pixel colour to the current render target . In this case we can just define it with the function like :

```
half4 UnlitPassFragment(Varyings input) : SV_Target
{
    // calculate color
    // ...

    return color;
}
```

It is possible for a shader to output to more than one render target though , known as **Multi Render Target** (MRT) . This is used by the Deferred Rendering path , e.g. [see UnityGBuffer.hlsl] (which isn't fully supported in URP yet) .

If not using the deferred path , using MRT would require setup on the C# side , such as using `Graphics.SetRenderTarget` with a `RenderBuffer[]` array , or `CommandBuffer.SetRenderTarget` with a `RenderTargetIdentifier[]` array . MRT is not supported on all platforms however (e.g. GLES2)

In the shader we would define the MRT output like so :

```
struct FragOut
{
    half4 color : SV_Target0; // aka SV_Target
    half4 color2 : SV_Target1; // another render target
};

FragOut UnlitPassFragment(Varyings input)
{
    // calculate color and color2
    // ...

    FragOut output;
    output.color = color;
    output.color2 = color2;
    return output;
}
```

It is also possible to change the value used for depth , using the `SV_Depth` semantic (or `SV_DepthGreaterEqual/SV_DepthLessEqual`) as explained in my Depth article.