**Part1. Introduction to Shader in Unity3D**

Type of Shader:

1. Surface shaders
2. Fragment and Vertex shaders

Anatomy of Shader

Shader "MyShader"

{

Properties

{

// The properties of your shaders

// - textures

// - colours

// - parameters

// ...

}

SubShader

{

// The code of your shaders

// - surface shader

// OR

// - vertex and fragment shader

// OR

// - fixed function shader

}

}

You can have multiple SubShader sections.

**Property**

Properties

{

\_MyTexture ("My texture", 2D) = "white" {}

\_MyNormalMap ("My normal map", 2D) = "bump" {} // Grey

\_MyInt ("My integer", Int) = 2

\_MyFloat ("My float", Float) = 1.5

\_MyRange ("My range", Range(0.0, 1.0)) = 0.5

\_MyColor ("My colour", Color) = (1, 0, 0, 1) // (R, G, B, A)

\_MyVector ("My Vector4", Vector) = (0, 0, 0, 0) // (x, y, z, w)

}

The type 2D, indicates the parameters for textures.

Default value of 2D, in string way, {white, black, gray, bump (specially for normal map)}

Vectors and Colors always have four elements (XYZW and RGBA, respectively).

**Data type in Property**

2D, Int, Float, Range (), Color, Vector

The parameter name should contain a \_ before it.

**Sub Shader**

SubShader

{

// Code of the shader

// ...

sampler2D \_MyTexture;

sampler2D \_MyNormalMap;

int \_MyInt;

float \_MyFloat;

float \_MyRange;

half4 \_MyColor;

float4 \_MyVector;

// Code of the shader

// ...

}

**Data type in SubShader relative to Property**

|  |  |
| --- | --- |
| 2D | sampler2D |
| Int | int |
| Float | float |
| Range | float |
| Color | half4 |
| Vector | float4 |

The name of parameter in sub shader should be the same as the one in property.

If you can define a property of type Vector, which is linked to a float2 variable; the extra two values will **be ignored** by Unity.

**Rendering Order**

The body of a shader, typically looks like this:

SubShader

{

Tags

{

"Queue" = "Geometry"

"RenderType" = "Opaque"

}

CGPROGRAM

// Cg / HLSL code of the shader

// ...

ENDCG

}

**Tags** are a way of telling Unity certain properties of the shader we are writing. For instance, **the order** in which it should be rendered (Queue) and **how it should be rendered** (RenderType).

When rendering triangles, the GPU usually sort them **according to their distance from the camera,** so that the further ones are drawn first.

Queue accepts integer positive numbers (the **smaller** it is, the **sooner** is drawn); mnemonic labels can also be used:

* Background (1000): used for backgrounds and skyboxes,
* Geometry (2000): the default label used for most solid objects,
* Transparent (3000): used for materials with transparent properties, such glass, fire, particles and water;
* Overlay (4000): used for effects such as lens flares, GUI elements and texts.

**ZTest**

It is important to remember, however, that an object from Transparent doesn’t necessarily always appear above an object from Geometry. The GPU, by default, performs a test called ZTest which **stops hidden pixels from being drawn.** To work, it uses an extra buffer with the same size of the screen its rendering to. Each pixel contains the depth (distance from the camera) of the object drawn in that pixel. If we are to write a pixel which is further away than the current depth, the pixel is discarded. The ZTest culls the pixel which are hidden by other object, regardless the order in which they are drawn onto the screen.

**The surface shader**

Whenever the material you want to simulate needs to **be affected by lights in a realistic way**, chances are you’ll need a surface shader.

CGPROGRAM

// Uses the Lambertian lighting model

#pragma surface surf Lambert

sampler2D \_MainTex; // The input texture

struct Input {

float2 uv\_MainTex;

};

void surf (Input IN, inout SurfaceOutput o) {

o.Albedo = tex2D (\_MainTex, IN.uv\_MainTex).rgb;

}

ENDCG

**The vertex and fragment shader**

Vertex and fragment shaders work close to the way the GPU renders triangles and have no built-in concept of how light should behave.

The geometry of your model is first passed through a function called **vert** which can alter its vertices. Then, individual triangles are passed through another function called **frag** which decides the final RGB colour for every pixel. They are useful for **2D effects, postprocessing and special 3D effects** which are too complex to be expressed as surface shaders.

Pass {

CGPROGRAM

#pragma vertex vert

#pragma fragment frag

struct vertInput {

float4 pos : POSITION;

};

struct vertOutput {

float4 pos : SV\_POSITION;

};

vertOutput vert(vertInput input) {

vertOutput o;

o.pos = mul(UNITY\_MATRIX\_MVP, input.pos);

return o;

}

half4 frag(vertOutput output) : COLOR {

return half4(1.0, 0.0, 0.0, 1.0);

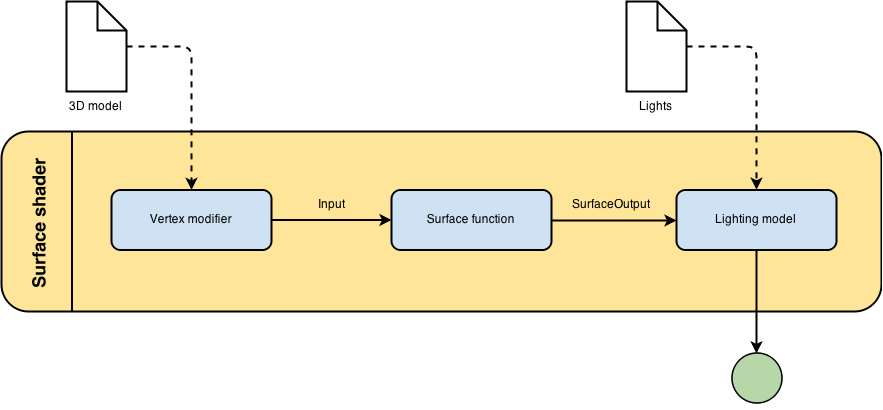
}

ENDCG

}

The vert function converts the vertices from their native 3D space to their final 2D position on the screen. Unity introduces the UNITY\_MATRIX\_MVP to hide the maths behind it. After this, the return of the frag function gives a red colour to every pixel. Just remember that the Cg section of vertex and fragment shaders need to be enclosed in a Pass section. This is not the case for simple surface shaders, which will work with or without it.

**Part2. Surface Shader in Unity3D**



**The surface functions**

The heart of a surface shader is its *surface function*. It takes data from the 3D model as input, and outputs its rendering properties. The following surface shader gives an object a diffuse white colour:

|  |  |
| --- | --- |
|  | Shader "Example/Diffuse Simple" {      SubShader {        Tags { "RenderType" = "Opaque" }        CGPROGRAM        #pragma surface surf Lambert        struct Input {            float4 color : COLOR;        };        void surf (Input IN, inout SurfaceOutput o) {            o.Albedo = 1; // 1 = (1,1,1,1) = white        }        ENDCG      }      Fallback "Diffuse"    } |

Cg / HLSL requires a input struct to be defined.

**Surface output**

The struct SurfaceOutput has several other properties which can be used to determine the final aspect of a material:

fixed3 Albedo: the base colour / texture of an object,

fixed3 Normal: the direction of the face, which determines its reflection angle,

fixed3 Emission: how much light this object is generating by itself,

half Specular: how well the material reflects lights from 0 to 1,

fixed Gloss: how diffuse the specular reflection is,

fixed Alpha: how transparent the material is.

Cg supports the fixed type. It spans at least from -2 to +2, and it uses 10 bits

**Sampling Textures**

Shader "Example/Diffuse Texture" {

    Properties {

      \_MainTex ("Texture", 2D) = "white" {}

    }

    SubShader {

      Tags { "RenderType" = "Opaque" }

      CGPROGRAM

      #pragma surface surf Lambert

      struct Input {

          float2 uv\_MainTex;

      };

      sampler2D \_MainTex;

      void surf (Input IN, inout SurfaceOutput o) {

          o.Albedo = tex2D (\_MainTex, IN.uv\_MainTex).rgb;

      }

      ENDCG

    }

    Fallback "Diffuse"

  }

tex2D: given a texture and some UV coordinate, it returns the RGBA colour. tex2D takes into account other parameters which can be set directly from Unity3D, when importing the texture.

float3 viewDir: the direction of the camera (view direction);

float4 name : COLOR: by using this syntax, the variable name will contain the colour of the vertex;

float4 screenPos: the position on the current pixel on the screen;

float3 worldPos: the position of the current pixel, in world coordinates.