```
Shader "Name" {

    Properties {

        _Name ("display name", Range (min, max)) = number
        _Name ("display name", Float) = number
        _Name ("display name", Int) = number
        _Name ("display name", Color) = (number,number,number,number)
        _Name ("display name", Vector) = (number,number,number,number)

        // 2D can use built-in default textures: “white”, “black”, “gray” or “bump”
        _Name ("display name", 2D) = "" {}
        _Name ("display name", Cube) = "" {}
        _Name ("display name", 3D) = "" {}

        // Property attribute drawers
        // [HideInInspector] - does not show the property value in the material inspector.
        // [NoScaleOffset] - material inspector will not show texture tiling/offset fields for
texture properties with this attribute.
        // [Normal] - indicates that a texture property expects a normal-map.
        // [HDR] - indicates that a texture property expects a high-dynamic range (HDR) texture.
        // [Gamma] - indicates that a float/vector property is specified as sRGB value in the UI
(just like colors are), and possibly needs conversion according to color space used. See
Properties in Shader Programs.
        // [PerRendererData] - indicates that a texture property will be coming from per-renderer
data in the form of a MaterialPropertyBlock. Material inspector changes the texture slot UI for
these properties.
        // [Enum(UnityEngine.Rendering.BlendMode)] _Blend ("Blend mode", Float) = 1 - blend modes
selection.
        // [Toggle(ENABLE_FANCY)] _Fancy ("Fancy?", Float) = 0 - Will set "ENABLE_FANCY" shader
keyword when set.
        // [KeywordEnum(None, Add, Multiply)] _Overlay ("Overlay mode", Float) = 0 - Display a
popup with None,Add,Multiply choices. Each option will set _OVERLAY_NONE, _OVERLAY_ADD,
_OVERLAY_MULTIPLY shader keywords.
        // [Enum(UnityEngine.Rendering.CompareFunction)] _ZTest("ZTest", Float) = 0
        // [Enum(UnityEngine.Rendering.CullMode)] _CullMode("Cull Mode", Int) = 0
        // Later on in the shader’s fixed function parts, property values can be accessed using
property name in square brackets: [name] e.g. Blend [_SrcBlend] [_DstBlend].
    }

    SubShader {

        Tags { "TagName1" = "Value1" "TagName2" = "Value2" }

        // Queue tag:
        // Background - this render queue is rendered before any others. You’d typically use this
for things that really need to be in the background.
        // Geometry (default) - this is used for most objects. Opaque geometry uses this queue.
        // AlphaTest - alpha tested geometry uses this queue. It’s a separate queue from Geometry
one since it’s more efficient to render alpha-tested objects after all solid ones are drawn.
        // Transparent - this render queue is rendered after Geometry and AlphaTest, in back-to-
front order. Anything alpha-blended (i.e. shaders that don’t write to depth buffer) should go here
(glass, particle effects).
        // Overlay - this render queue is meant for overlay effects. Anything rendered last should
go here (e.g. lens flares).

        // RenderType tag: (Can you custom values and use Camera.RenderWithShader or
Camera.SetReplacementShader to render camera with selected subshader e.g.
replacementTag="RenderType=Transparent"
        // Opaque: most of the shaders (Normal, Self Illuminated, Reflective, terrain shaders).
        // Transparent: most semitransparent shaders (Transparent, Particle, Font, terrain
additive pass shaders).
        // TransparentCutout: masked transparency shaders (Transparent Cutout, two pass vegetation
shaders).
        // Background: Skybox shaders.
        // Overlay: GUITexture, Halo, Flare shaders.
        // TreeOpaque: terrain engine tree bark.
        // TreeTransparentCutout: terrain engine tree leaves.
        // TreeBillboard: terrain engine billboarded trees.
        // Grass: terrain engine grass.
        // GrassBillboard: terrain engine billboarded grass.

        // RequireOptions tag:
        // SoftVegetation: Render this pass only if Soft Vegetation is on in Quality Settings.

        // DisableBatching tag
        // True, False, LODFading

        // ForceNoShadowCasting tag

        // IgnoreProjector tag

        // CanUseSpriteAtlas tag

        // PreviewType tag
        // Sphere, Plane, Skybox

        GrabPass { }
        // Just GrabPass { } will grab current screen contents into a texture. The texture can be
accessed in further passes by _GrabTexture name.
        // Note: this form of grab pass will do the expensive screen grabbing operation for each
object that uses it!
        // GrabPass { "TextureName" } will grab screen contents into a texture, but will only do
that once per frame for the first object that uses the given texture name.
        // The texture can be accessed in further passes by the given texture name. This is a more
performant way when you have multiple objects using grab pass in the scene.
        // Additionally, GrabPass can use Name and Tags commands.

        UsePass "Shader/Name"
        // Inserts all passes with a given name from a given shader. Shader/Name contains the name
of the shader and the name of the pass, separated by a slash character.
        // Note that only first supported subshader is taken into account.

        Pass {
            Name "Pass Name"
            Tags { "TagName1" = "Value1" "TagName2" = "Value2" }

            // LightMode tag:
            // Always: Always rendered; no lighting is applied.
            // ForwardBase: Used in Forward rendering, ambient, main directional light, vertex/SH
lights and lightmaps are applied.
            // ForwardAdd: Used in Forward rendering; additive per-pixel lights are applied, one
pass per light.
            // Deferred: Used in Deferred Shading; renders g-buffer.
            // ShadowCaster: Renders object depth into the shadowmap or a depth texture.
            // PrepassBase: Used in legacy Deferred Lighting, renders normals and specular
exponent.
            // PrepassFinal: Used in legacy Deferred Lighting, renders final color by combining
textures, lighting and emission.
            // Vertex: Used in legacy Vertex Lit rendering when object is not lightmapped; all
vertex lights are applied.
            // VertexLMRGBM: Used in legacy Vertex Lit rendering when object is lightmapped; on
platforms where lightmap is RGBM encoded (PC & console).
            // VertexLM: Used in legacy Vertex Lit rendering when object is lightmapped; on
platforms where lightmap is double-LDR encoded (mobile platforms).

            // Render setup
            // Cull Back | Front | Off
            // ZTest (Less | Greater | LEqual | GEqual | Equal | NotEqual | Always)
            // ZWrite On | Off
            // Blend SourceBlendMode DestBlendMode
            // Blend SourceBlendMode DestBlendMode, AlphaSourceBlendMode AlphaDestBlendMode
            // ColorMask RGB | A | 0 | any combination of R, G, B, A
            // Offset OffsetFactor, OffsetUnits

            CGPROGRAM
            #pragma vertex name // compile function name as the vertex shader.
            #pragma fragment name // compile function name as the fragment shader.
            #pragma geometry name // compile function name as DX10 geometry shader. Having this
option automatically turns on #pragma target 4.0, described below.
            #pragma hull name // compile function name as DX11 hull shader. Having this option
automatically turns on #pragma target 5.0, described below.
            #pragma domain name // compile function name as DX11 domain shader. Having this option
automatically turns on #pragma target 5.0, described below.

            // Other compilation directives:
            // #pragma target name - which shader target to compile to. See Shader Compilation
Targets page for details.
            // #pragma only_renderers space separated names - compile shader only for given
renderers. By default shaders are compiled for all renderers. See Renderers below for details.
            // #pragma exclude_renderers space separated names - do not compile shader for given
renderers. By default shaders are compiled for all renderers. See Renderers below for details.
            // #pragma multi_compile - for working with multiple shader variants.
                        // #pragma shader_feature - for working with multiple shader variants.
(unused variants of shader_feature shaders will not be included into game build)
            // #pragma enable_d3d11_debug_symbols - generate debug information for shaders
compiled for DirectX 11, this will allow you to debug shaders via Visual Studio 2012 (or higher)
Graphics debugger.
            // #pragma multi_compile_instancing

            // VARIABLES
            // Color and Vector properties map to float4, half4 or fixed4 variables.
            // Range and Float properties map to float, half or fixed variables.
            // Texture properties map to sampler2D variables for regular (2D) textures; Cubemaps
map to samplerCUBE; and 3D textures map to sampler3D.

            // Shader property values are found and provided to shaders from these places:
            // Per-Renderer values set in MaterialPropertyBlock. This is typically “per-instance”
data (e.g. customized tint color for a lot of objects that all share the same material).
            // Values set in the Material that’s used on the rendered object.
            // Global shader properties, set either by Unity rendering code itself (see built-in
shader variables), or from your own scripts (e.g. Shader.SetGlobalTexture).

            // Simple VS and FS example
            sampler2D _MainTex;

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

            struct v2f {
                float4 vertex : SV_POSITION;
                float2 uv : TEXCOORD0;
            };

            v2f vert(appdata v) {
                v2f o;
                o.vertex = mul(UNITY_MATRIX_MVP, v.vertex);
                o.uv = v.uv;
                return o;
            }

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

            // Multitarget (Deferred)
            void frag(v2f i, out half4 outDiffuse : SV_Target0, out half4 outSpecSmoothness :
SV_Target1, out half4 outNormal : SV_Target2, out half4 outEmission : SV_Target3) {
                // ...
            }
            ENDCG
        }
    }

    Fallback "Diffuse"
}

// Other stuff

// Single Pass Stereo Rendering
// https://docs.unity3d.com/Manual/SinglePassStereoRendering.html
// Functions in UnityCG.cginc

//      uv - UV texture coordinates. Either a float2 for a standard UV or a float4 for a packed
pair of two UVs.
//      sb - A float4 containing a 2D scale and 2D bias to be applied to the UV, with scale in xy
and bias in zw.
//      Description: Returns the result of applying the scale and bias in sb to the texture
coordinates in uv.
//      This only occurs when UNITY_SINGLE_PASS_STEREO is defined, otherwise the texture
coordinates are returned unmodified.
//      This is often used to apply a per-eye scale and bias only when in Single-Pass Stereo
rendering mode.
UnityStereoScreenSpaceUVAdjust(uv, sb)

//      uv - UV texture coordinates. Either a float2 for a standard UV or a float4 for a packed
pair of two UVs.
//      Description: Returns the result of applying the current eye’s scale and bias to the
texture coordinates in uv.
//      This only occurs when UNITY_SINGLE_PASS_STEREO is defined, otherwise the texture
coordinates are returned unaltered.
UnityStereoTransformScreenSpaceTex(uv)

//      uv - UV texture coordinates. Either a float2 for a standard UV or a float4 for a packed
pair of two UVs.
//      sb - A float4 containing a 2D scale and 2D bias to be applied to the UV, with scale in xy
and bias in zw.
//      Description: Returns the uv clamped in the x value by the width and bias provided by sb.
This only occurs
//      when UNITY_SINGLE_PASS_STEREO is defined, otherwise the texture coordinates are returned
unmodified.
//      This is often used to apply a per-eye clamping in Single-Pass Stereo rendering mode to
avoid color bleeding between eyes.
UnityStereoClamp(uv, sb)

ComputeNonStereoScreenPos
ComputeScreenPos

//      Additionally, the constant unity_StereoEyeIndex is exposed in Shaders, so eye-dependent
calculations can be performed.
//      The value of unity_StereoEyeIndex is 0 for rendering of the left eye, and 1 for rendering
of the right eye.
unity_StereoEyeIndex
```