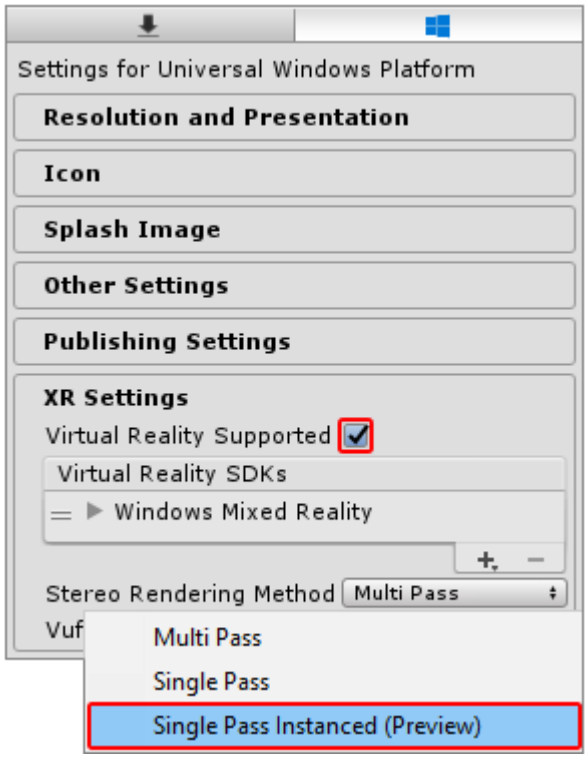
Single Pass Instanced rendering

With **Single Pass Instanced rendering**The process of drawing graphics to the screen (or to a render texture). By default, the main camera in Unity renders its view to the screen. More info
See in Glossary (also known as Stereo Instancing), the GPU performs a single render pass, replacing each draw call with an instanced draw call. This heavily decreases CPU use, and slightly decreases GPU use, due to the cache coherency between the two draw calls. This significantly reduces power consumption of your application.

Supported platforms

- PlayStation VR
- **Oculus**A VR platform for making applications for Rift and mobile VR devices. More info
  See in Glossary Rift (DirectX 11)
- HoloLens
- Magic Leap
- Android devices that support the Multiview extension
- For DirextX on desktop, the GPU must support Direct3D 11 and the `VPAndRTArrayIndexFromAnyShaderFeedingRasterizer` extension.
- For OpenGL on desktop, the GPU must support one of the following extensions:
  - `GL_NV_viewport_array2`
  - `GL_AMD_vertex_shader_layer`
  - `GL_ARB_shader_viewport_layer_array`

To enable this feature, open **Player** settings (go to **Edit** > **Project Settings**A broad collection of settings which allow you to configure how Physics, Audio, Networking, Graphics, Input and many other areas of your Project behave. More info
See in Glossary, then select the **Player** category). In the **Player** settings, navigate to the **XR Settings** panel at the bottom, check the **Virtual Reality Supported** option, then select **Single Pass Instanced (Preview)** from the **Stereo Rendering Method** drop-down menu.



In the XR Settings panel, set the Stereo Rendering Method to Single Pass Instanced (Preview)

The default **Stereo Rendering Method** is **Multi Pass**. This setting is slower, but usually works better with custom **shaders**A small script that contains the mathematical calculations and algorithms for calculating the Color of each pixel rendered, based on the lighting input and the Material configuration. More info
See in Glossary. If you have custom **shaders**, you might need to change them to make them compatible with **Single Pass Instanced** rendering.

Custom shaders

Before you follow the instructions below, update your custom shaders to use instancing (see GPU Instancing).

Next, you need to make two additional changes in the last shader stage used before the **fragment shader**The "per-pixel" part of shader code, performed every pixel that an object occupies on-screen. The fragment shader part is usually used to calculate and output the color of each pixel. More info
See in Glossary (Vertex/Hull/Domain/Geometry) for any of your custom shaders.

For each custom shader you want to support Single Pass Instancing, carry out the following steps:

Step 1: Add `UNITY_VERTEX_INPUT_INSTANCE_ID` to the `appdata struct`.

Example:

```
struct appdata

{

    float4 vertex : POSITION;

    float2 uv : TEXCOORD0;



    UNITY_VERTEX_INPUT_INSTANCE_ID //Insert

};
```

Step 2: Add `UNITY_VERTEX_OUTPUT_STEREO` to the `v2f output struct`.

Example:

```
struct v2f

{

    float2 uv : TEXCOORD0;

    float4 vertex : SV_POSITION;



    UNITY_VERTEX_OUTPUT_STEREO //Insert

};
```

Step 3: Add the `UNITY_SETUP_INSTANCE_ID()` macro at the beginning of your main `vert` method, followed by a call to `UNITY_INITIALIZE_OUTPUT(v2f, o)` and `UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO()`.

`UNITY_SETUP_INSTANCE_ID()` calculates and sets the built-in `unity_StereoEyeIndex` and `unity_InstanceID` Unity shader variables to the correct values based on which eye the GPU is currently rendering.

`UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO` tells the GPU which eye in the texture array it should render to, based on the value of `unity_StereoEyeIndex`. This macro also transfers the value of `unity_StereoEyeIndex` from the **vertex shader**A program that runs on each vertex of a 3D model when the model is being rendered. More info
See in Glossary so that it will be accessible in the fragment shader only if `UNITY_SETUP_STEREO_EYE_INDEX_POST_VERTEX` is called in the fragment shader `frag` method.

`UNITY_INITALIZE_OUTPUT(v2f,o)` initializes all `v2f` values to 0.

Example:

```
v2f vert (appdata v)

{

    v2f o;

    UNITY_SETUP_INSTANCE_ID(v); //Insert

    UNITY_INITIALIZE_OUTPUT(v2f, o); //Insert

    UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(o); //Insert

    o.vertex = UnityObjectToClipPos(v.vertex);

    o.uv = v.uv;

    return o;

}
```

Post-Processing shaders

If you want your Post-Processing shaders to support Single Pass Instancing, follow the steps in Custom shaders as well as the steps below . You can download all Unity base shader scripts from the Unity website.

Do the following for each Post-Processing shader that you want to support Single Pass Instancing:

Step 1: Add the UNITY_DECLARE_SCREENSPACE_TEXTURE(tex) macro outside the frag method (see the placement example below) in your Shader script, so that when you use a particular stereo rendering method the GPU uses the appropriate texture sampler. For example, if you use Multi-Pass rendering, the GPU uses a texture 2D sampler. For single pass instancing or multi-view rendering, the texture sampler is a texture array.

Step 2: Add `UNITY_SETUP_STEREO_EYE_INDEX_POST_VERTEX(i)` at the beginning of the fragment shader frag method (See the placement example below). You only need to add this macro if you want to use the `unity_StereoEyeIndex` built-in shader variable to find out which eye the GPU is rendering to. This is useful when testing post processing effects.

Step 3: Use the `UNITY_SAMPLE_SCREENSPACE_TEXTURE()` macro when sampling 2D textures (See the placement example below). Standard shaders use a 2D texture-based back buffer to sample textures. Single Pass Stereo Instancing does not use this type of back buffer, so if you do not specify a different method for 2D texture sampling, your shader does not render correctly. To prevent rendering issues, the `UNITY_SAMPLE_SCREENSPACE_TEXTURE()` macro detects which stereo rendering path you are using and then automatically samples the texture in the correct manner. See Unity documentation on HLSLSupport.cginc to learn more about similar macros used for depth textures and screen-space shadow maps.

Example:

```
UNITY_DECLARE_SCREENSPACE_TEXTURE(_MainTex); //Insert

fixed4 frag (v2f i) : SV_Target
{

    UNITY_SETUP_STEREO_EYE_INDEX_POST_VERTEX(i); //Insert

    fixed4 col = UNITY_SAMPLE_SCREENSPACE_TEXTURE(_MainTex, i.uv); //Insert

        // just invert the colors

        col = 1 - col;

        return col;

}
```

Full sample shader code

Below is a simple example of the template image effect shader with all of the previously mentioned changes applied to allow Single Pass Instancing support. The additions to the shader code are marked with a comment (`//Insert`).

```
struct appdata

{

    float4 vertex : POSITION;

    float2 uv : TEXCOORD0;

        UNITY_VERTEX_INPUT_INSTANCE_ID //Insert

};

//v2f output struct

struct v2f

{

    float2 uv : TEXCOORD0;

    float4 vertex : SV_POSITION;

        UNITY_VERTEX_OUTPUT_STEREO //Insert

};

v2f vert (appdata v)

{

    v2f o;

        UNITY_SETUP_INSTANCE_ID(v); //Insert

    UNITY_INITIALIZE_OUTPUT(v2f, o); //Insert

    UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(o); //Insert
```

```
    o.vertex = UnityObjectToClipPos(v.vertex);
    o.uv = v.uv;
    return o;
}
UNITY_DECLARE_SCREENSPACE_TEXTURE(_MainTex); //Insert
fixed4 frag (v2f i) : SV_Target
{
    UNITY_SETUP_STEREO_EYE_INDEX_POST_VERTEX(i); //Insert
    fixed4 col = UNITY_SAMPLE_SCREENSPACE_TEXTURE(_MainTex, i.uv); //Insert
    // invert the colors
    col = 1 - col;
    return col;
}
```

Procedural geometry

When using the Graphics.DrawProceduralIndirect() and CommandBuffer.DrawProceduralIndirect() methods to draw fully procedural geometry on the GPU, it is important to note that both methods receive their arguments from a compute buffer. This means that it is difficult to increase the instance count at run time. To increase the instance count, you need to manually double the instance count contained in your compute buffers.

See Vertex and fragment shader examples for more information on how to write shader code.

Debugging your shader

The following shader code renders a GameObject as green for a user's left eye and red for their right eye. This shader is useful for debugging your stereo rendering, because it allows you to verify that all stereo graphics work and are functioning correctly.

```
Shader "__XR__An umbrella term encompassing Virtual Reality (VR), Augmented Reality (AR) and Mixed Reality (MR) applications. Devices supporting these forms of interactive applications can be referred to as XR devices. [More info](XR.html)<span class="tooltipGlossaryLink">See in [Glossary](Glossary.html#XR)</span>/StereoEyeIndexColor"

{

   Properties

   {

      _LeftEyeColor("Left Eye Color", COLOR) = (0,1,0,1)

      _RightEyeColor("Right Eye Color", COLOR) = (1,0,0,1)

   }

   SubShader

   {

      Tags { "RenderType" = "Opaque" }

      Pass

      {

         CGPROGRAM
```

```
        #pragma vertex vert
        #pragma fragment frag
        float4 _LeftEyeColor;
        float4 _RightEyeColor;
        #include "UnityCG.cginc"
        struct appdata
        {
            float4 vertex : POSITION;
            UNITY_VERTEX_INPUT_INSTANCE_ID
        };
        struct v2f
        {
            float4 vertex : SV_POSITION;
            UNITY_VERTEX_INPUT_INSTANCE_ID
            UNITY_VERTEX_OUTPUT_STEREO
        };
        v2f vert (appdata v)
        {
            v2f o;
            UNITY_SETUP_INSTANCE_ID(v);
            UNITY_INITIALIZE_OUTPUT(v2f, o);
            UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(o);
```

```
            o.vertex = UnityObjectToClipPos(v.vertex);



            return o;
        }
        fixed4 frag (v2f i) : SV_Target
        {
            UNITY_SETUP_STEREO_EYE_INDEX_POST_VERTEX(i);
            return lerp(_LeftEyeColor, _RightEyeColor, unity_StereoEyeIndex);
        }
        ENDCG
    }
  }
}
```