CC_CANOE Stereo Rendering Library

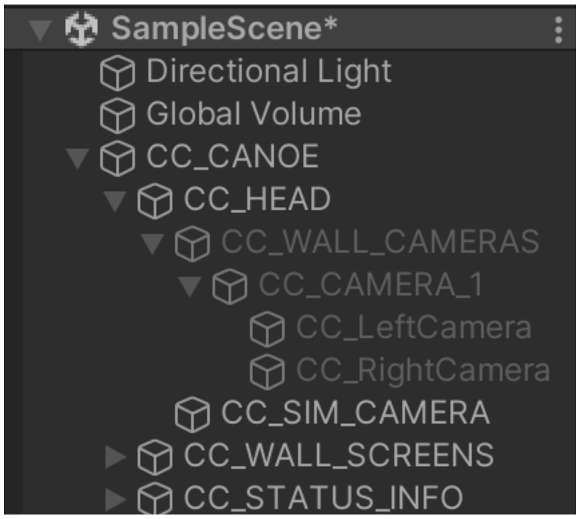
© July 1, 2022 - Jason Leigh
Laboratory for Advanced Visualization & Applications
University of Hawaiʻi at Mānoa

This code base provides off-axis stereoscopic 3D rendering for Unity.
This is the basis for proper 3D rendering in systems like display walls, CAVEs, CAVE2s, CyberCANOEs, where the stereo image displayed is correctly rendered according to a user's head position and orientation.

Using the Code

To use the code, start with the project template (preferably using Unity 2021 and up). This code only supports Unity's Universal Render Pipeline (URP) as Unity is gradually deprecating the old built-in render pipeline. So if you using / purchasing assets on the Unity asset store to use with CC_CANOE, make sure they support URP, especially those that have shaders.

When you open the project you'll see the following in the scene hierarchy.



```
SampleScene*
    Directional Light
    Global Volume
    CC_CANOE
        CC_HEAD
            CC_WALL_CAMERAS
                CC_CAMERA_1
                    CC_LeftCamera
                    CC_RightCamera
            CC_SIM_CAMERA
        CC_WALL_SCREENS
        CC_STATUS_INFO
```

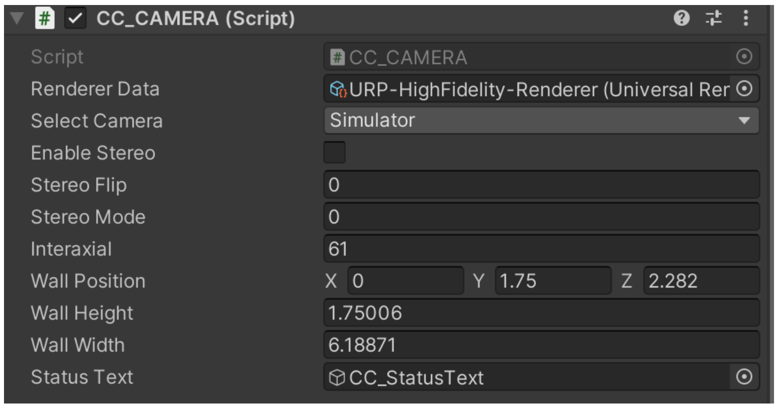
CC_CANOE is the main hierarchy. Think of CC_CANOE as the "vehicle" that moves through the virtual space and you are standing inside that vehicle. To move CC_CANOE through space you simply adjust the position and orientation of the CC_CANOE gameobject.

CC_HEAD is the gameobject representing your head. Your head resides within the CC_CANOE. If you are using some kind of tracker (like Vive or Kinect), you will need to feed the tracker's head data to CC_HEAD by directly setting its position and orientation. If you don't intend to do any kind of head tracking or tracking in general, just leave CC_HEAD alone. It is currently set to place you at the center of the CC_CANOE looking along the positive Z axis. It is also defaulted to a Y value (height) of 1.75m to emulate the perspective of an average 5'10" male.

CC_HEAD camera should remain OFF. It doesn't really hurt to leave it on. But there is no need. Just leave it off.

CC_WALL_SCREENS - contains a CC_SCREEN_1.
These determine the position of the projection screen and therefore it should be set to the same position as the *physical* display wall. By default it is set to 6.18m x 1.75m to emulate LAVA's 3D display wall. Also by default this display is placed at position (0,1.75,2.282).

When using the Unity *editor*, you can adjust the size and position of CC_WALL_SCREENS to match your physical wall by adjusting Wall Position, Wall Width and Wall Height in the CC_CANOE script (attached to CC_HEAD)



CC_CAMERA (Script)

Script	CC_CAMERA
Renderer Data	URP-HighFidelity-Renderer (Universal Rer
Select Camera	Simulator
Enable Stereo	
Stereo Flip	0
Stereo Mode	0
Interaxial	61
Wall Position	X 0 Y 1.75 Z 2.282
Wall Height	1.75006
Wall Width	6.18871
Status Text	CC_StatusText

When running your *standalone application* however you can set this information in the XML file (take a look at CCUnityConfig.xml for an example) so that the proper stereo configuration is loaded during runtime. You can of course still change them by pressing the keys (below) to override whatever configuration you just loaded.

Regardless of whether you are working in Unity's Editor or running the standalone application, you use the following keys to control the rendering:

0 = toggle wall vs simulator view
9 = toggle stereo on/off
8 = cycle through stereo rendering modes
7 = swap left and right eye images

These settings can also be set in the CC_UnityConfig.xml script (see example file) for the markups to use.

How It All Works (if you care)

CC_WALL_CAMERAS are the cameras that support rendering of the virtual world. The scene is rendered in 3 ways:

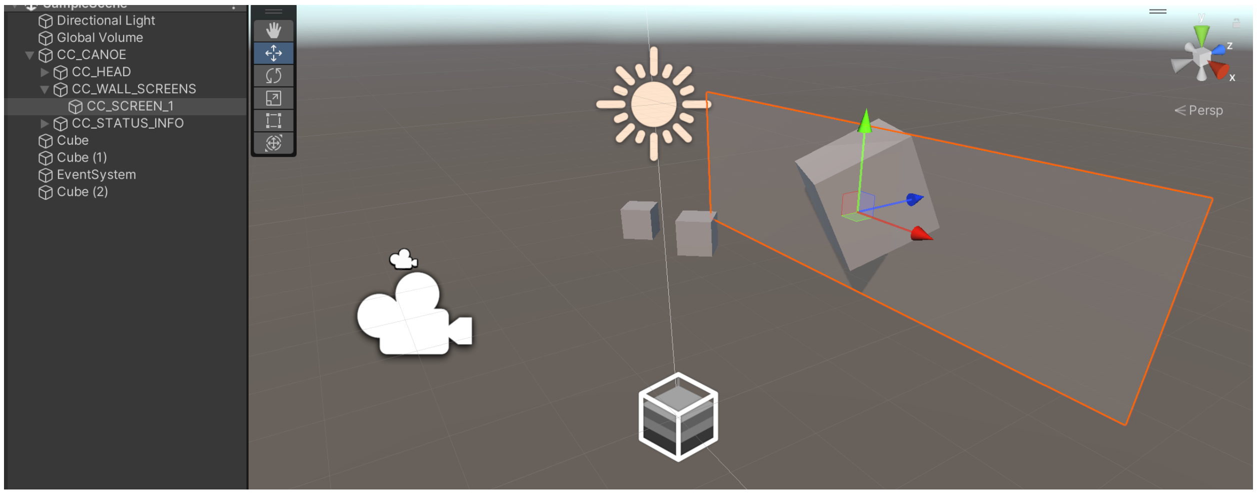
1. Simulator view (monoscopic view which is essentially same as CC_HEAD)
2. Wall view (CC_HEAD view adjusted for off-axis projection)
3. Stereo Wall views - which includes off-axis projection with interleaving, side-by-side, top-bottom rendering modes.

CC_CAMERA_1 is the center off-axis camera. CC_CAMERA_1's camera characteristics (like near and far clipping plane) are a copy of CC_HEAD. So whatever you set in CC_HEAD will be copied by your CC_CAMERA_1 cameras.

CC_LeftCamera and CC_RightCamera are created dynamically by CC_CAMERA_1. They are a copy of CC_CAMERA1 except they are offset horizontally by a distance determined by eye separation distance (Interaxial distance).

CC_SIM_CAMERA is the simulator camera. This is really the same as the camera for CC_HEAD. I am not sure why we need this. We could just use CC_HEAD. <— may have to look into stripping this out for efficiency.

The simulator camera is used to make it easier for developers to develop apps while being conscious of the physical position of the display wall (drawn as a ghosted panel CC_SCREEN_1).



Scripts:
CC_CAMERA.cs - this script coordinates everything: setting up the cameras, checking for key presses to switch between different rendering modes, telling the render pipeline which material to use to render the correct stereo mode, resizing the render textures whenever the screen size changes.

CC_CAMERAOFFSET.cs - this script does the off-axis projection calculations for a camera. This is embedded in every camera that requires off-axis projection rendering, namely: CC_LeftCamera, CC_RightCamera, and CC_CAMERA_1.

CC_CAMERASTEREO.cs - this script does the work of creating and setting up the cameras. The script sets up CC_CAMERA_1 as the center camera showing the non-stereo off-axis projection image. The off-axis projection image is the actual correct image that is projected on the display wall. Not the simulator camera.

The script also dynamically creates CC_LeftCamera and CC_RightCamera. It is important to note that these are the only cameras that have Post Processing enabled as they are the ones which are part of the stereo rendering pipeline (more later).

Stereo is rendered as follows:

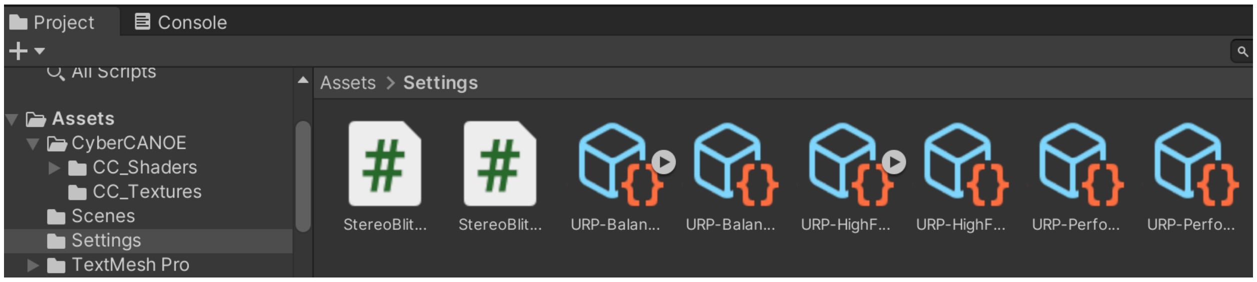
Stereo rendering is performed by taking the left and right render textures from CC_LeftCamera and CC_RightCamera and giving them to a shader to merge into a stereo picture.

3 shaders support the 3 main stereo rendering types: StereoInterlaceSG, StereoSideBySideSG, StereoTopDownSG. These are all implemented in Unity as shader graphs.

Shaders can only do the work they need to do by being attached to materials.
Therefore there are 3 materials: CC_INTERLACE, CC_SIDEBYSIDE, CC_TOPBOTTOM.

All shaders take the same inputs except the interlace shader which also takes the vertical resolution of the screen as input. The vertical resolution is given to the interlace material in the CC_CAMERA script (look at SetStereoMode function).

There are also scripts in Assets>Settings:

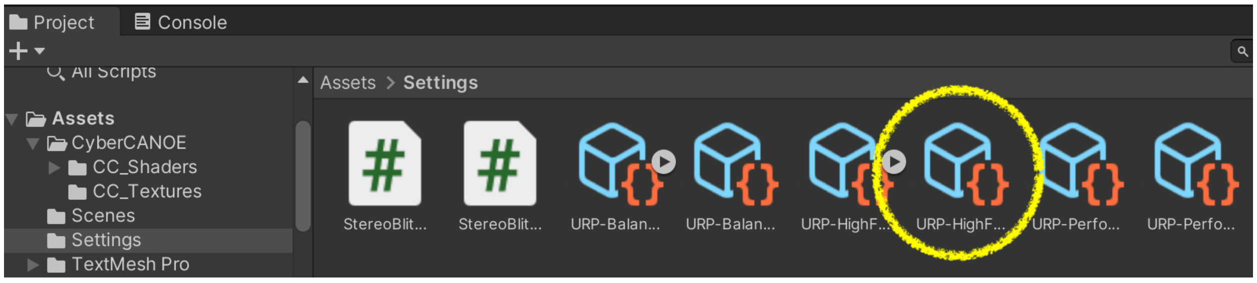


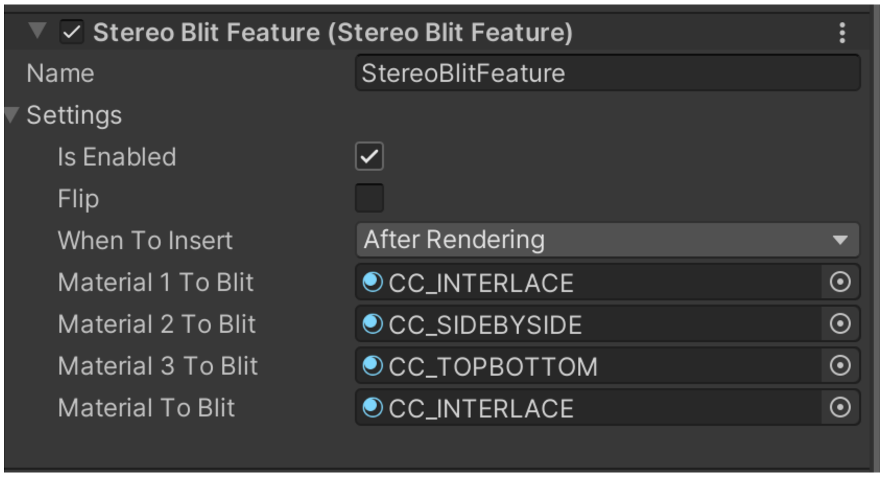
StereoBlitFeature.cs - this sets up the custom rendering pipeline to accept the 3 types of materials (Material1ToBlit, Material2ToBlit, Material3ToBlit). Material1ToBlit is the material for interlaced stereo. Material2ToBlit is material for the side-by-side stereo. Material3ToBlit is material for the tip-down stereo. MatertialToBlit is the current material being rendered in the render pipeline.

The Flip parameter is set to true if you wish to invert the stereo (i.e. swap left and right eye images). In the CC_CAMERA script it is toggled by pressing “7”.

StereoBlitRenderingPass.cs - this does the work of taking the left and right eye images and running them through the chosen shader, and blitting the image back to the display for viewing.

Also in the Settings folder you will see URP-High Fidelity-Renderer. This is how we pass the rendering parameters into the pipeline. If you click on it and look at the bottom of the Inspector, you will see the settings exposed by StereoBlitFeature. There is no need to alter any of this.





Stereo Blit Feature (Stereo Blit Feature)

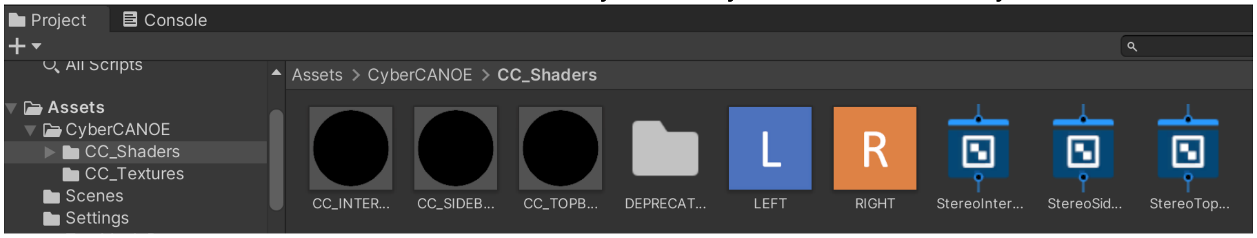
Name	StereoBlitFeature
Settings	
Is Enabled	✓
Flip	
When To Insert	After Rendering
Material 1 To Blit	CC_INTERLACE
Material 2 To Blit	CC_SIDEBYSIDE
Material 3 To Blit	CC_TOPBOTTOM
Material To Blit	CC_INTERLACE

CC_CONFIG.cs - loads XML file of configurations to set up the wall size, wall resolution, wall position, stereo mode, etc.. This file is only opened when you are running in Application mode. It will be ignored when you are running in the Unity editor.

Look at CCUnityConfig.xml as an example. This file should be placed in a subdirectory called CC_UnityConfig, which itself should reside in the same location as your app's .exe file.

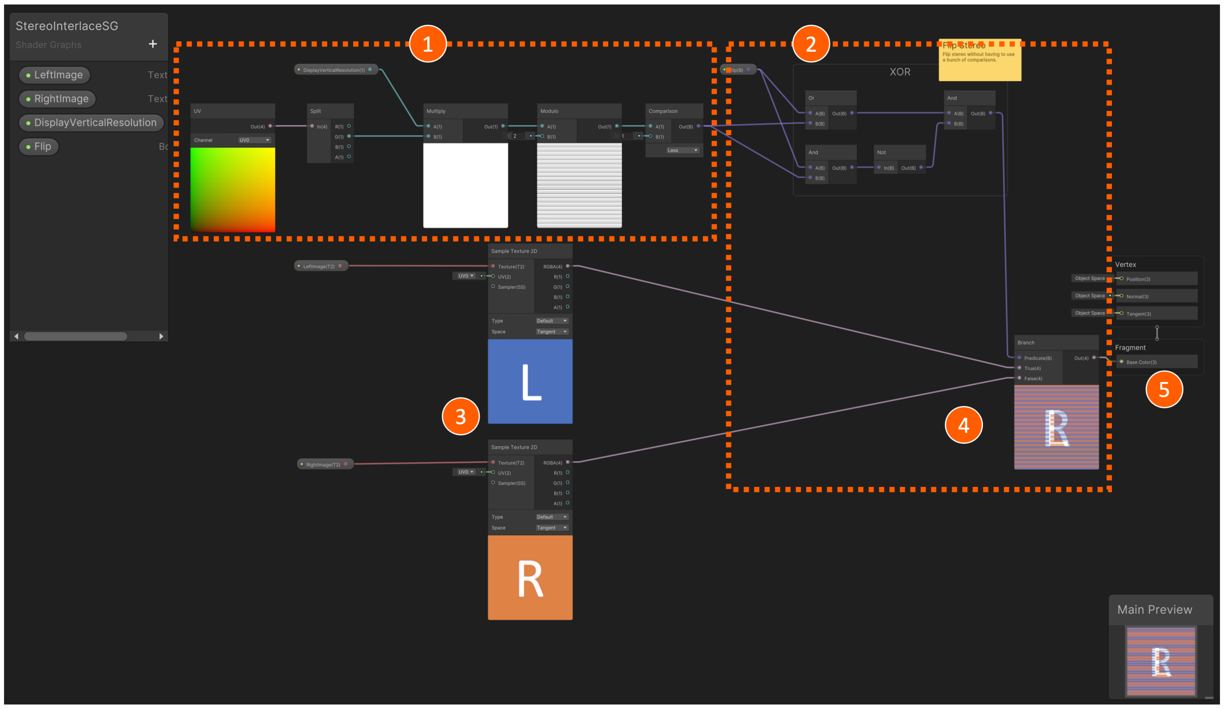
The Stereo Rendering Shaders

These are housed in the CC_Shaders directory of the CyberCANOE directory.



There are 3 shader graphs (StereoInterlaceSG, StereoSideBySideSG, StereTopBottomSG) and 3 materials that use the shaders (CC_INTERLACE, CC_SIDEBYSIDE, CC_TOPBUTTOM).

Interlace Shader



[1] Take the UV whose x and y values range between 0 and 1, and extract the y component (i.e. the vertical component of the texture). Multiply this floating point value by the vertical height in pixels of the input image and take the modulo of 2. This yields a sequential series of 0s and 1s.

[2] We feed this series of 0s and 1s into the Branch node to select the scanlines from the left image and the right image[3], and then place them into the resulting texture[4] that is finally fed to the fragment shader [5].

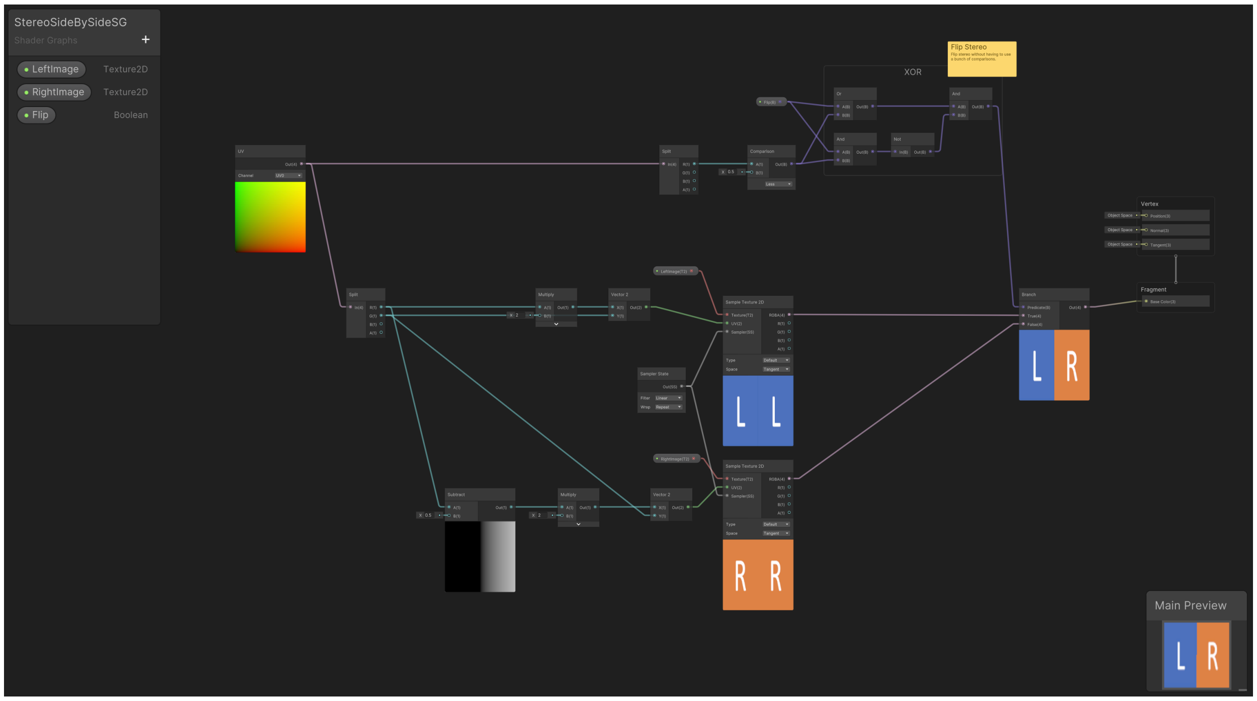
This is essentially what the shader code would look like in CG:

```
float4 frag(v2f_img IN) : COLOR0
{

    if (fmod(IN.uv.y * InterlaceValue, 2.0) < 1.0)
    {
        return tex2D(LeftTex, IN.uv);
    }
    else
    {
        return tex2D(RightTex, IN.uv);
    }
}
```

The XOR in [2] used to create the XOR logic to flip the interlacing if the Flip stereo option is set to true. Flip means we swap the left and right eye images. Unity does not have an XOR operator node.

Side-by-Side Shader

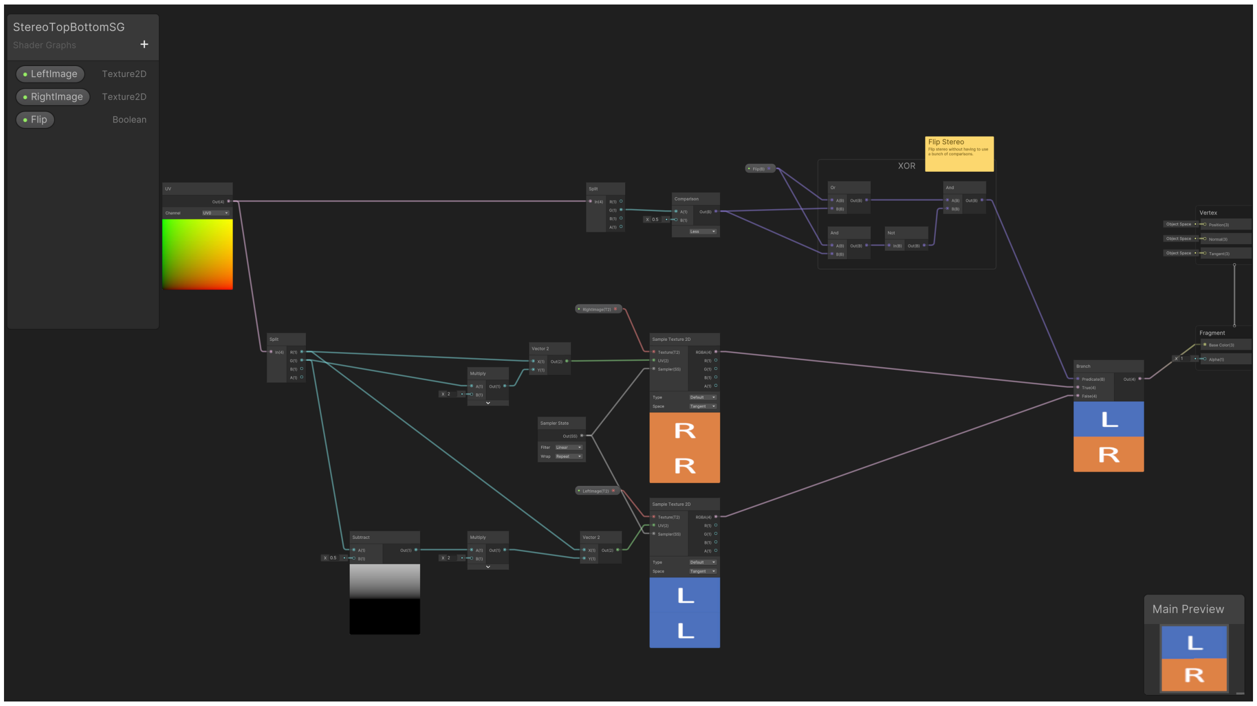


This is the CG code we are essentially achieving:

```
            float4 frag(v2f_img IN) : COLOR0
            {
                // If considering a pixel on the left half of screen
                if (IN.uv.x < 0.5)
                {
                    // Retrieve texel from left texture map at texture
location
                    // x * 2, y
                    return tex2D(LeftTex, float2(IN.uv.x*2.0,
IN.uv.y));
                }
                else {
                    // Retrieve texel from right texture map at
texture location
                    // (x - 0.5)*2, y
                    return tex2D(RightTex, float2((IN.uv.x-0.5)*2.0,
IN.uv.y));
                    }
            }
```

Top-Bottom Shader

The top-bottom shader is essentially the same as the side-to-side shader except we are stacking the left eye image on top of the right eye image.



Flip stereo without having to use a bunch of comparisons.

LeftImage	Texture2D
RightImage	Texture2D
Flip	Boolean