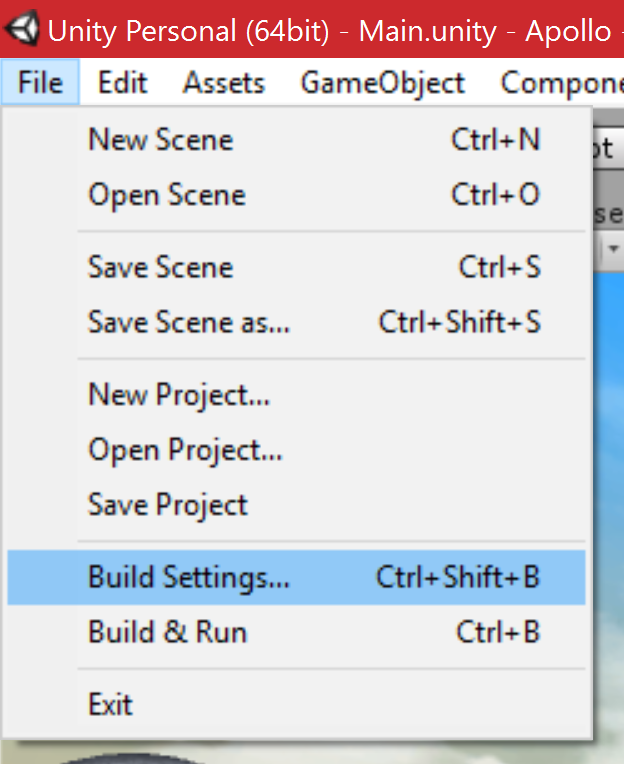
Challenge 2: Shader and Graphic debugging

**Step 1: Identify issue**

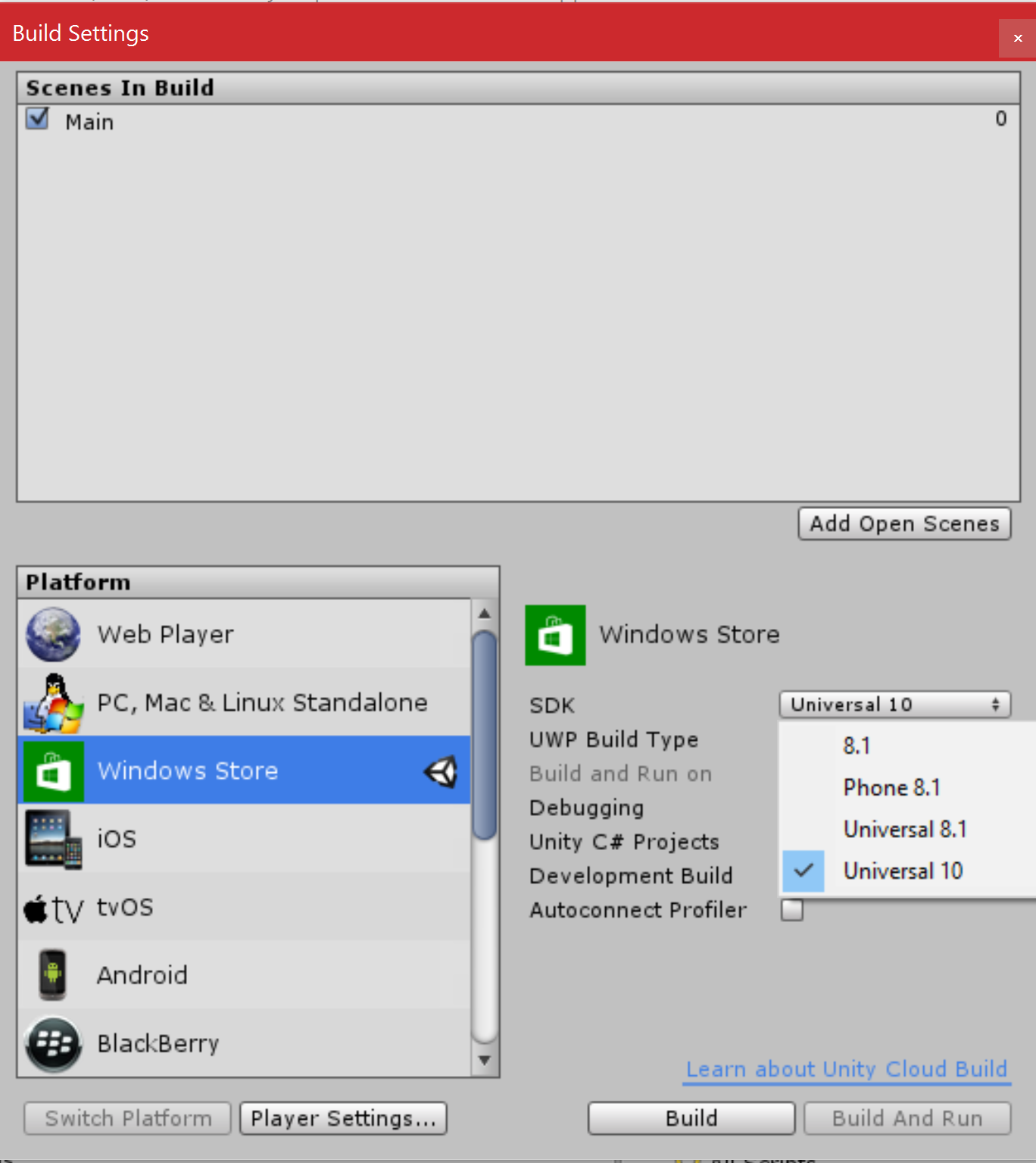
Open the Start Project in Unity and play another round of the game. Focus on the moon, and notice that it looks a bit weird.

The VS Graphics Diagnostics tools comes with a set of tools that enables you to see all the drawcalls as well as the history of selected pixels, as well as shader debugging and shader editing.

**Step 2: Export as a UWP**

Let’s go ahead and build the solution.  
In Unity, click File->Build Settings  


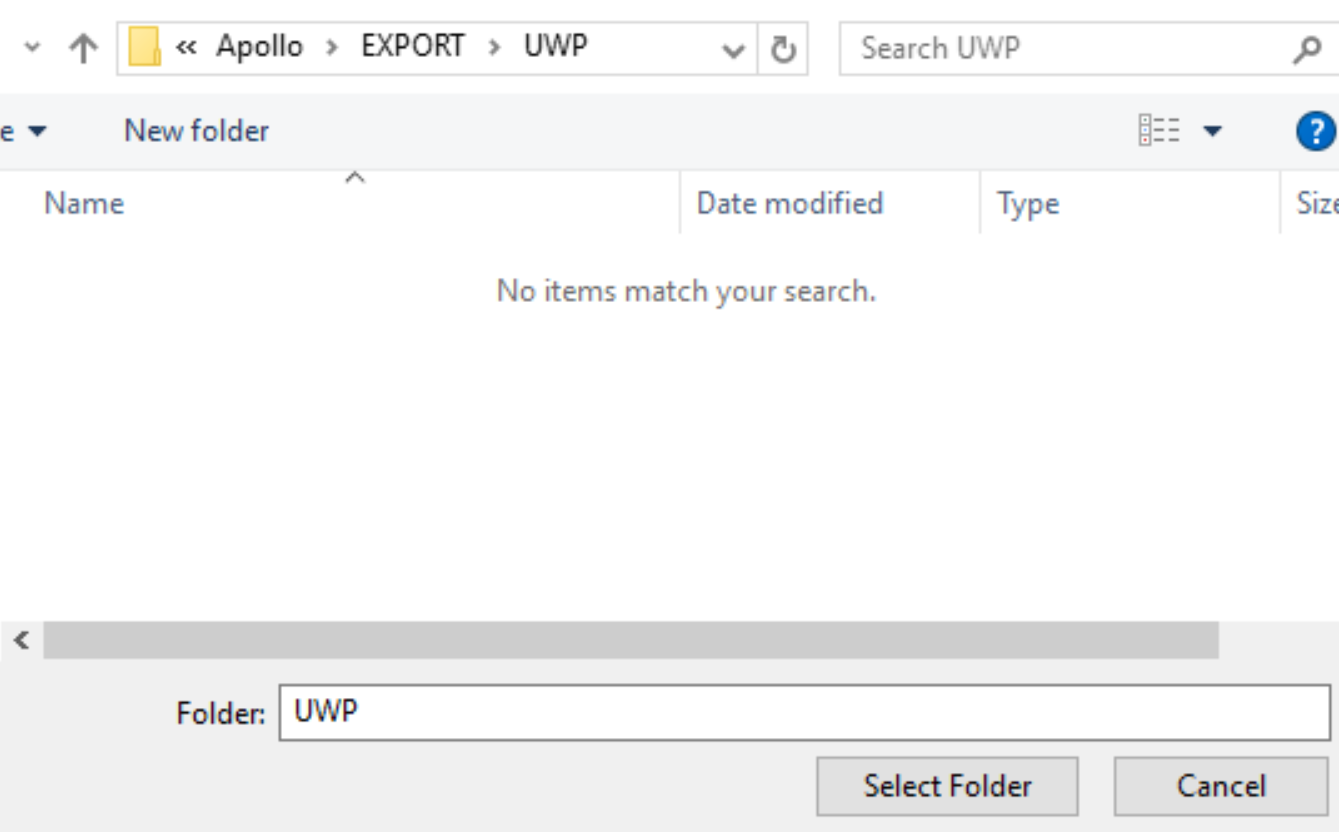
Ensure that Windows Store, and from the dropdown, Universal 10 is selected



Also, make sure the Unity C# Projects and the Development Build checkboxes is checked.

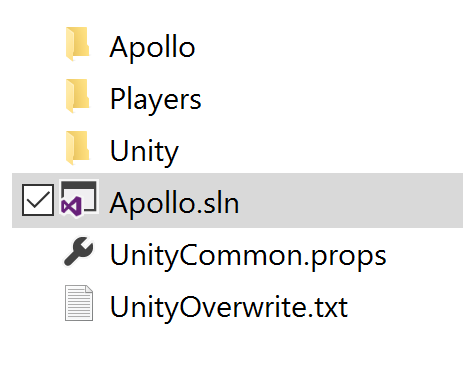


Click Build and navigate into EXPORT\UWP and click Select Folder to start the export.

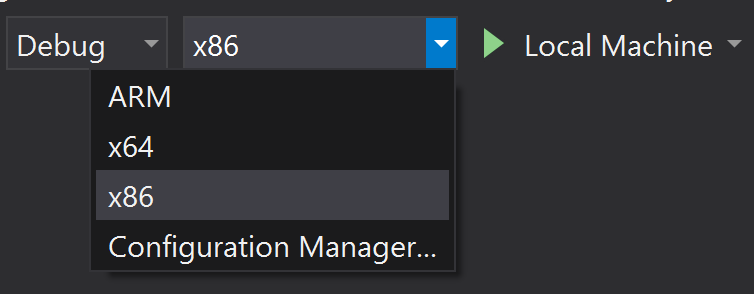


**Step 3: Opening the Visual Studio solution**

Once the export is done, the new folder should automatically open. Open the Apollo.sln.



Once Visual Studio loads the project, change the build configuration to x86, and make sure that it is in Debug. Click the Local Machine button to build, deploy and run the solution and test that it works:



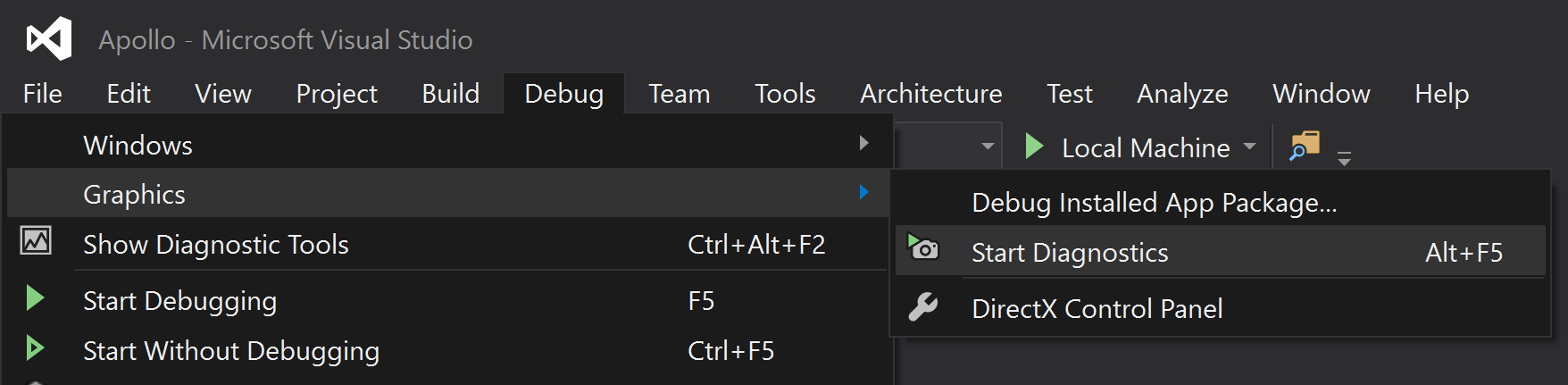
Note: Some build error messages might appear since this is the first time we are building the solution. This is normal as long as the build process is continuing.

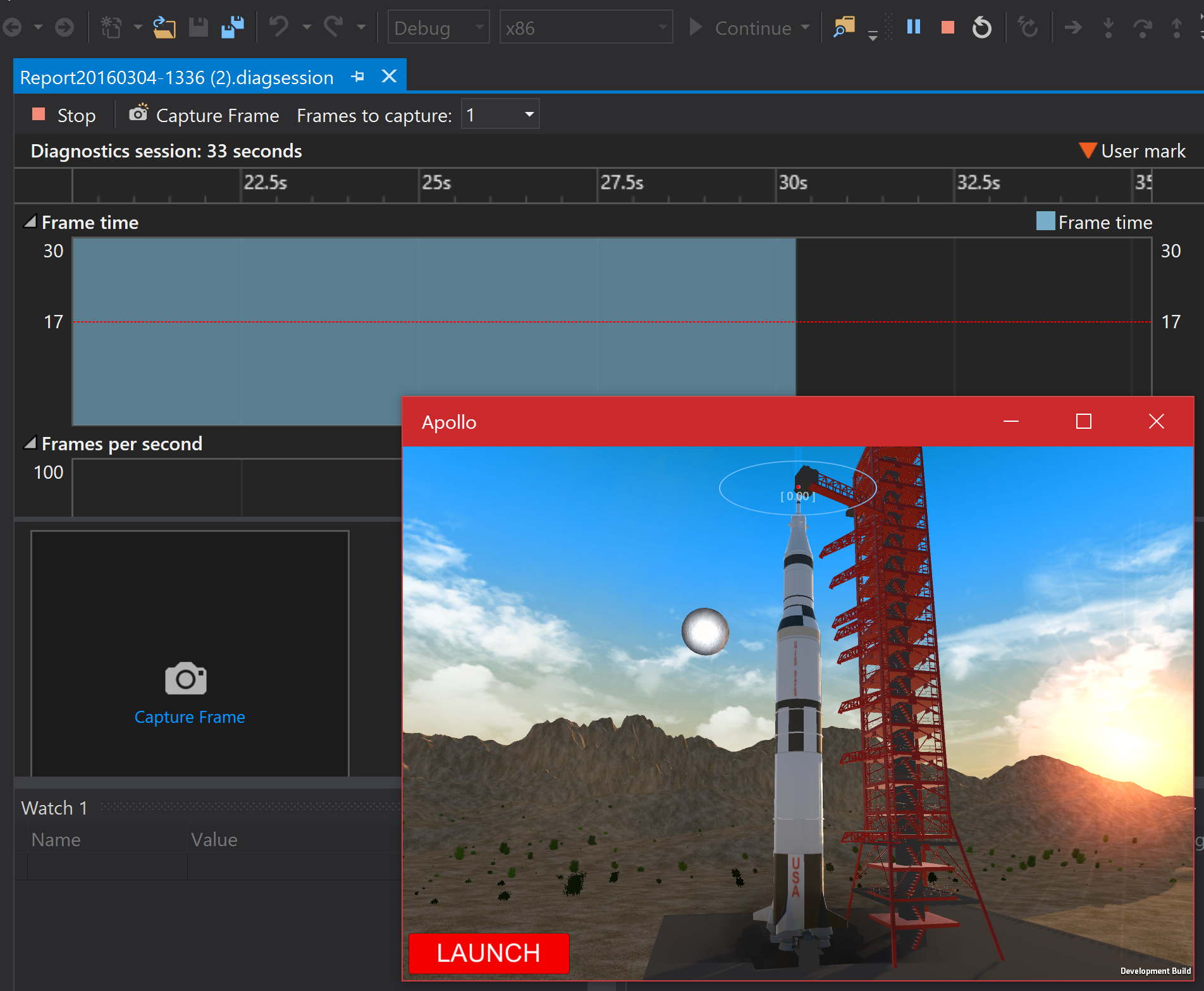
Feel free to try the game and ensure that the moon is looking strange.

Stop the game.

**Step 4: Run in Graphics Debug mode**

Let’s run this with the Graphics Diagnostics by clicking Debug->Graphics->Start Diagnostics



When running the game now, you are presented with performance graphs, as well as a Capture Frame button:  


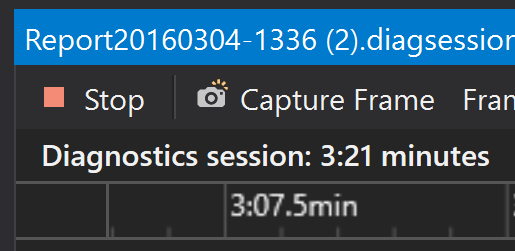
To capture a frame, you can click the Capture Frame button, or press Print screen on the keyboard when having the app in focus.

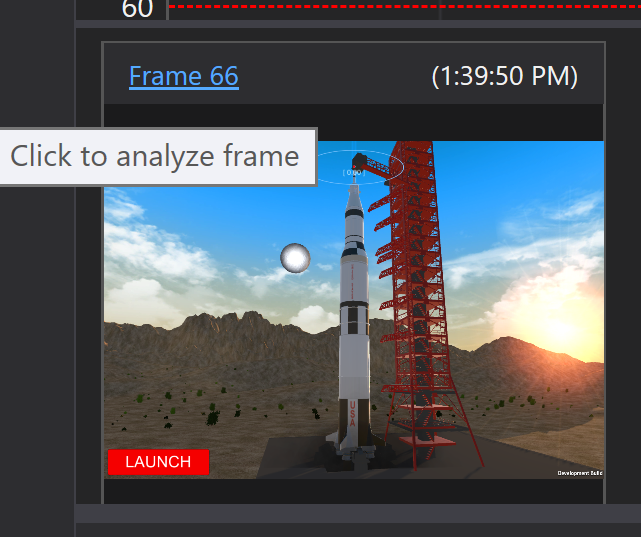
We are interested in finding out what’s happening with the moon. When the game starts, it’s stationary on the Launchpad, and the broken white moon is visible in the background.

When the moon is visible, capture a frame by clicking the button or using Print screen.

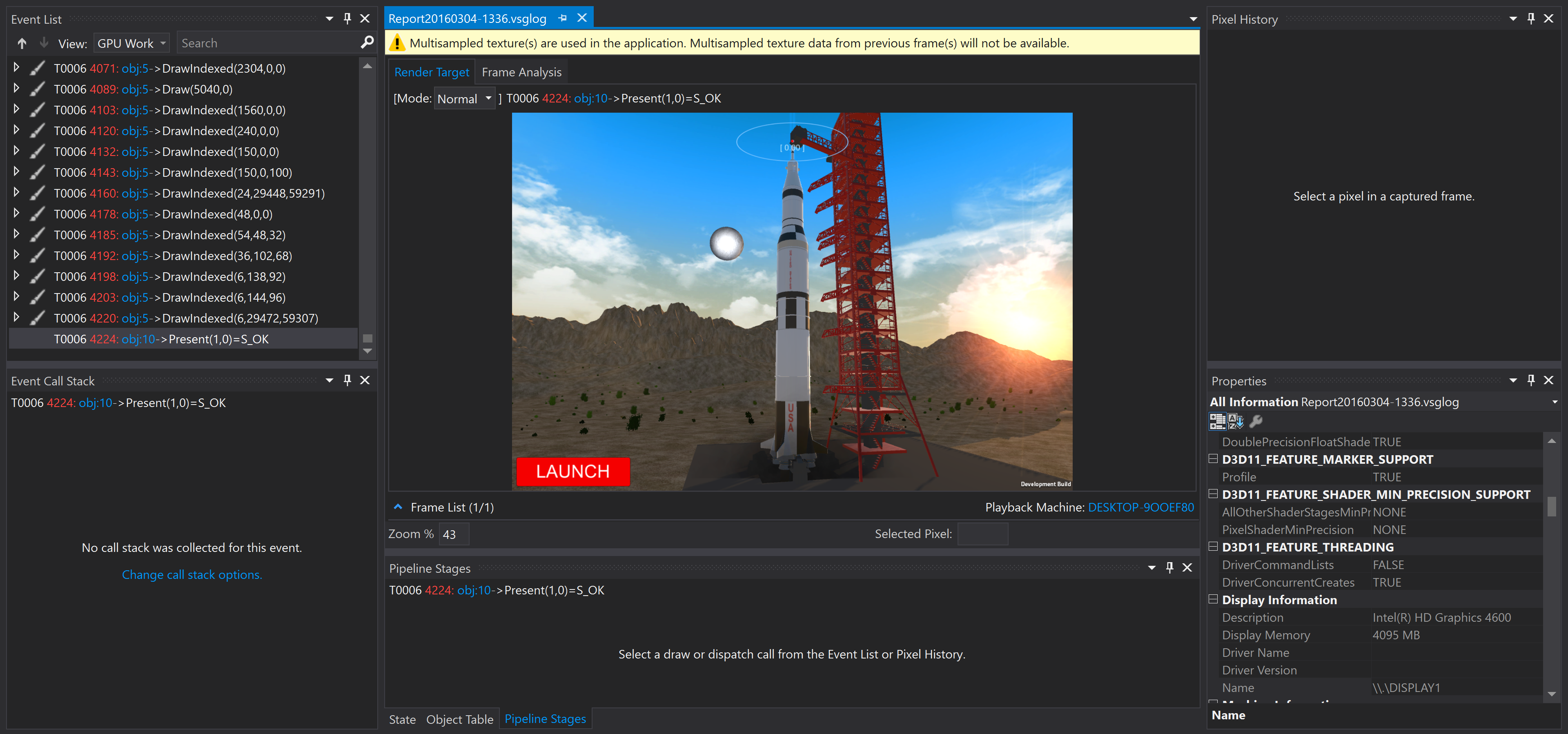
Then, when the frame is captured, you can see it visible where the capture frame button was.

Stop the session by clicking the Stop button:



Once the app stops, you can click the captured frame by clicking on the frame header:  


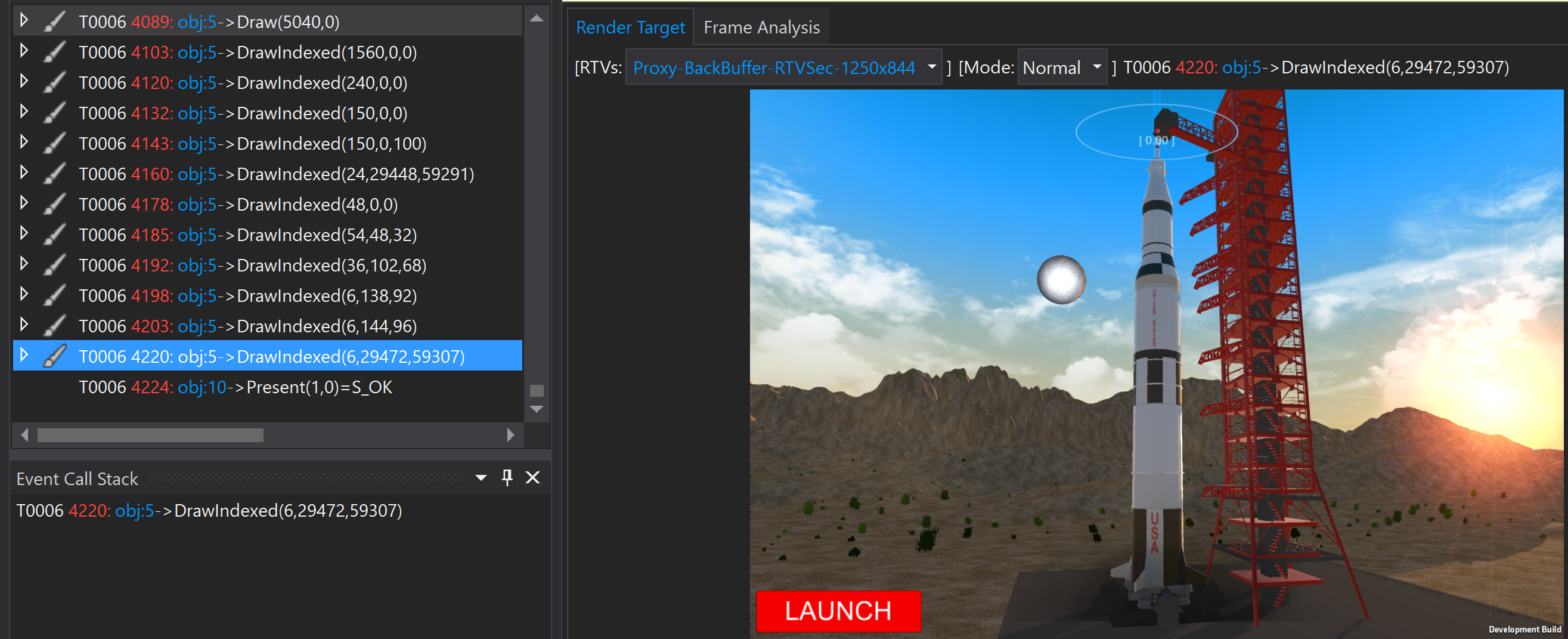
A new tab will open in visual studio, containing a snapshot of the system, processes, state of the app at that time the frame was captured.



**Step5 – Navigating around in the captured frame**

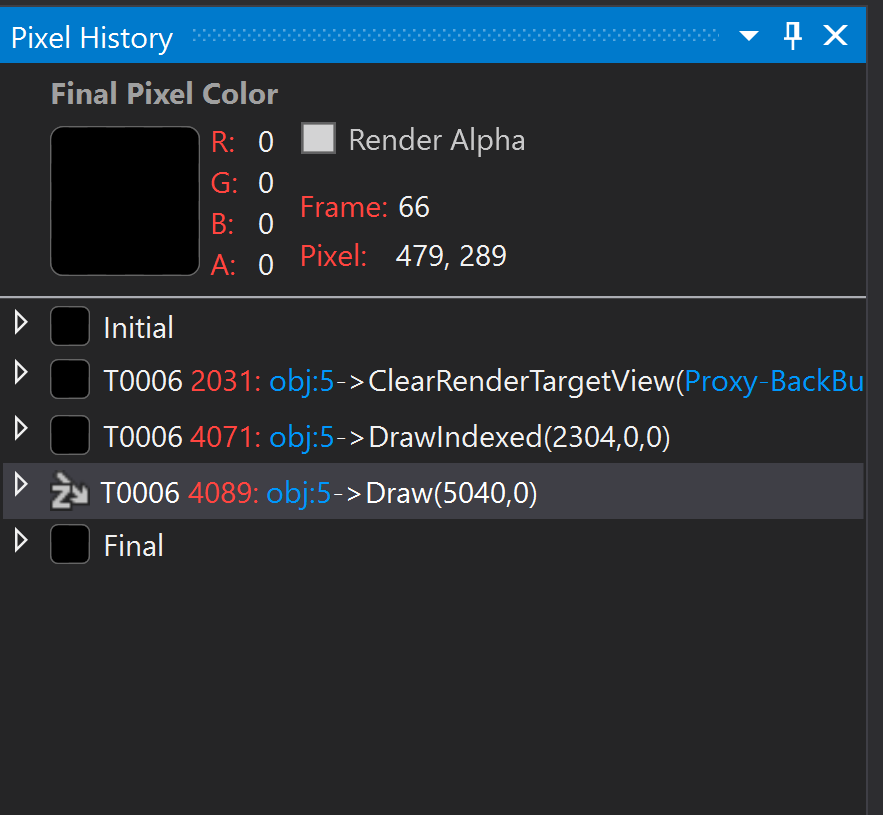
On the left side, there is a long list of events like draw calls, and in the center we are having a picture that shows the state of the frame at a selected event.

**Step 6: Finding the moon**

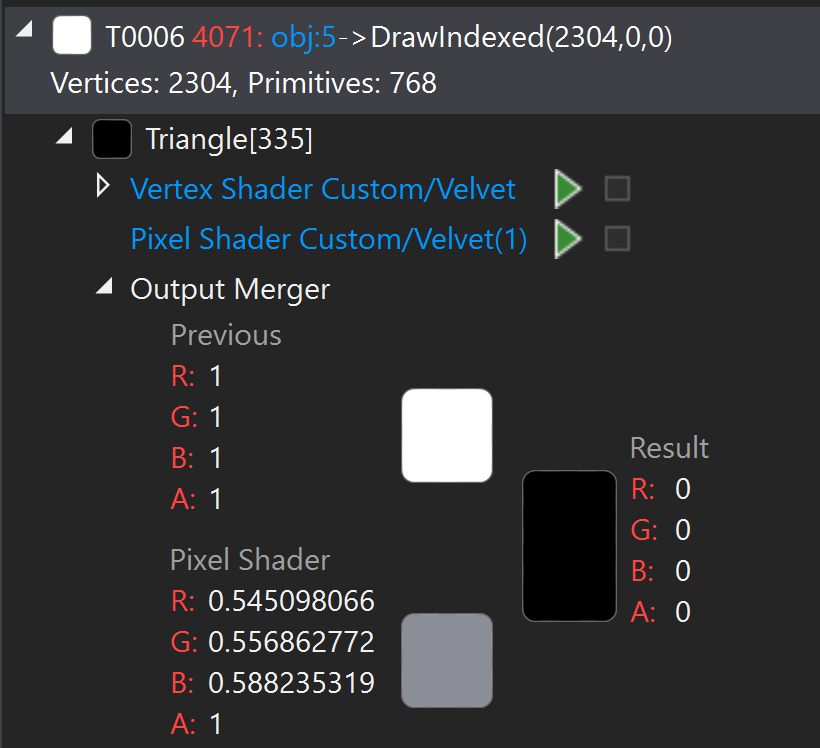
We are interested in finding the draw call that draws our moon. Click the last event before the Present event to be sure that we are having a preview that has the moon visible. 

Now, on the frame view, click a pixel inside of the moon. A red marker will show what pixel you have selected:  

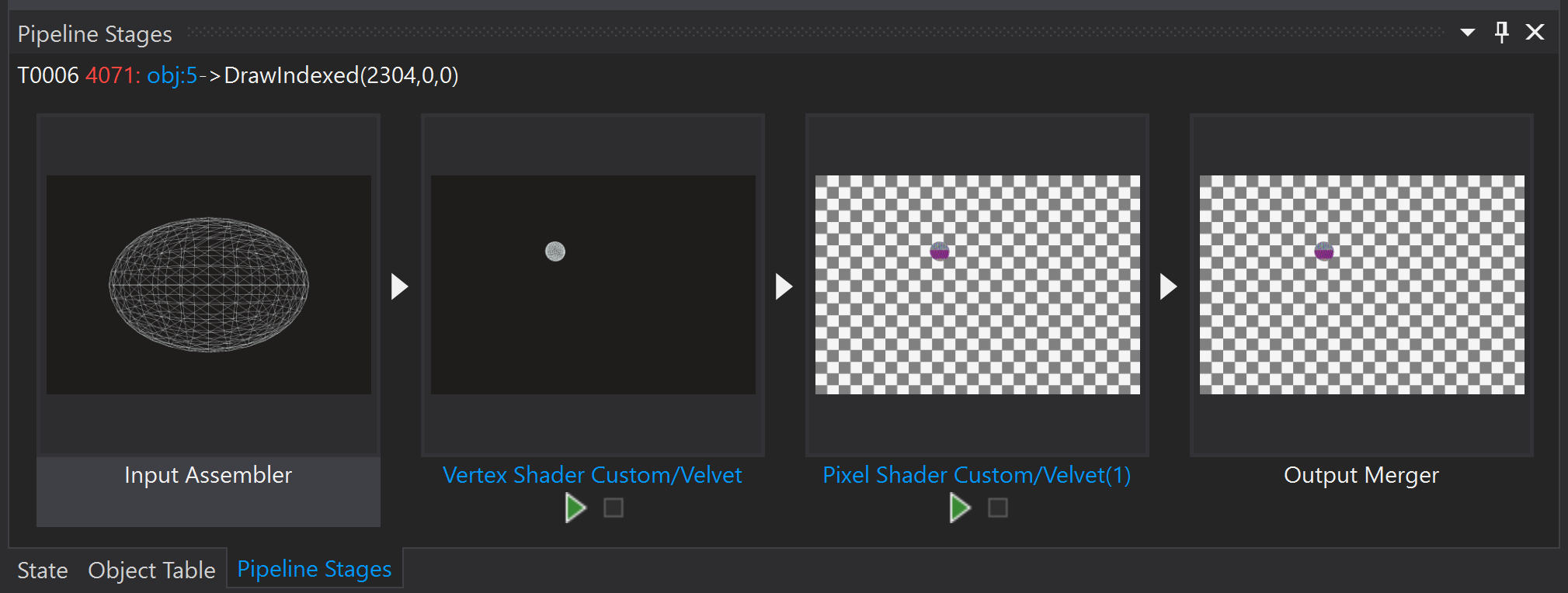

One the right side of this, in the properties view, you can see all the events that are related to this pixel:



The first one is the one that clears the screen to blue, the next draws the moon and the last draws the skybox. You can click each of these to see how it looks in the frame preview.

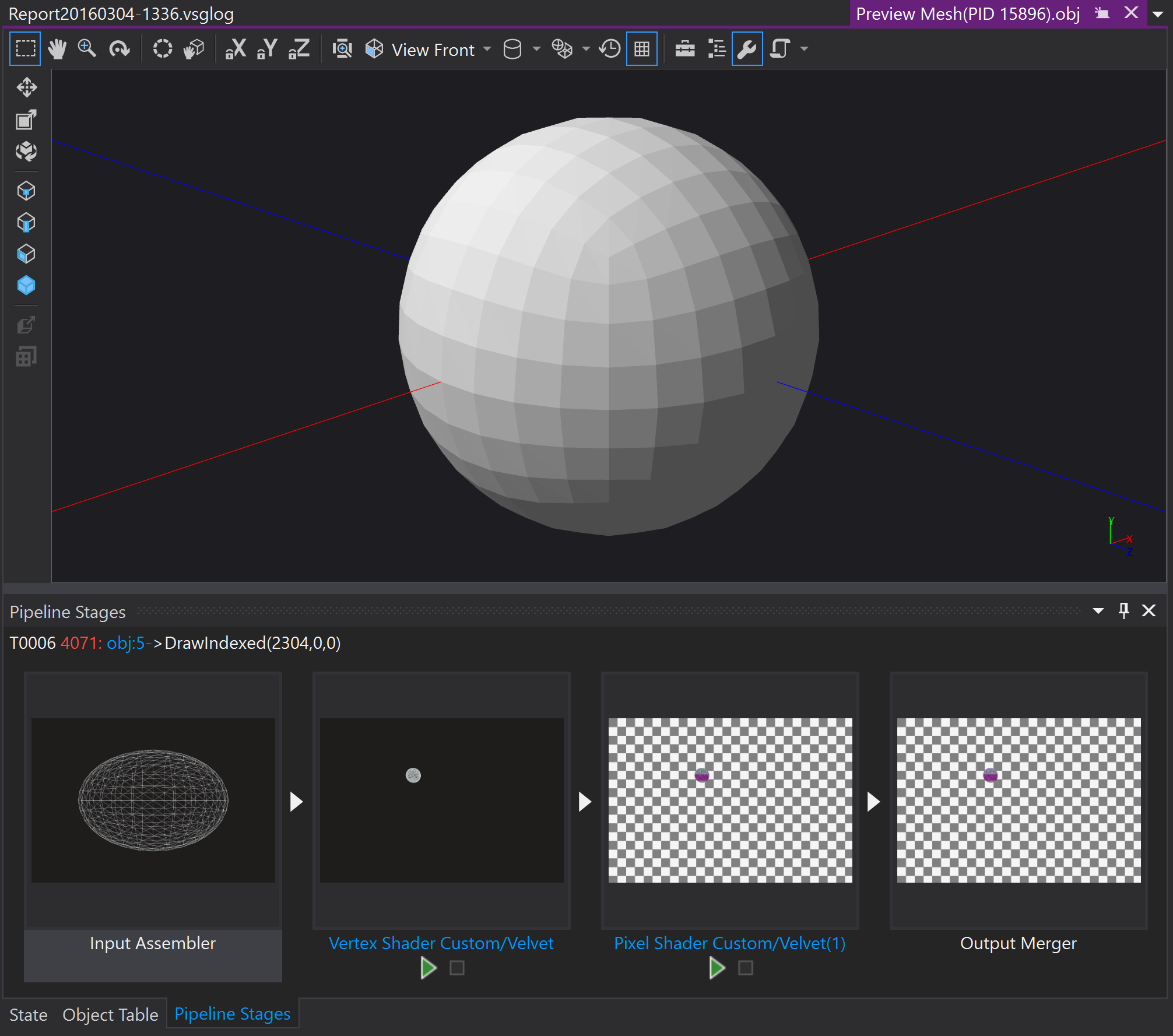
Open the event by clicking on the arrow of the event in the pixel history, and open the Triangle header. This will display information about how the pixel is affected, what shaders are being used and what color the various stages produce.  


**Step 7: Checking the graphics pipeline**Below the preview, all the way to the bottom of the screen, there are a few tabs where one is gamed Pipeline Stages. Click this:



This will show how all the pipeline stages the selected event is having. This one is having one Input stage where all the vertices that are being used is passed into the shader stages. The next is a vertex shader, that is executed once per vertex from the input stage, then there is a pixel shader that gives a color to every pixel, and then finally an output merger that marks the end of the graphics pipeline.

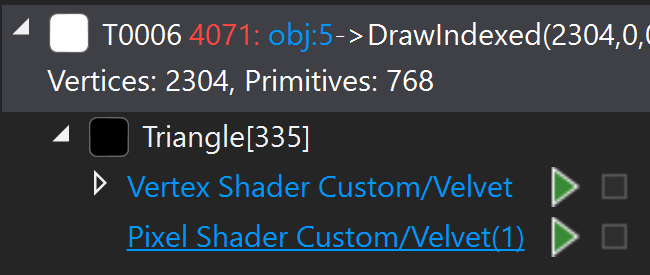
If you click on one of the stages, like the Input Assembler, you can see how the model currently in process looks like in the built in Visual Studio Model Viewer. You can see that this is a sphere, from what the moon is built up from.



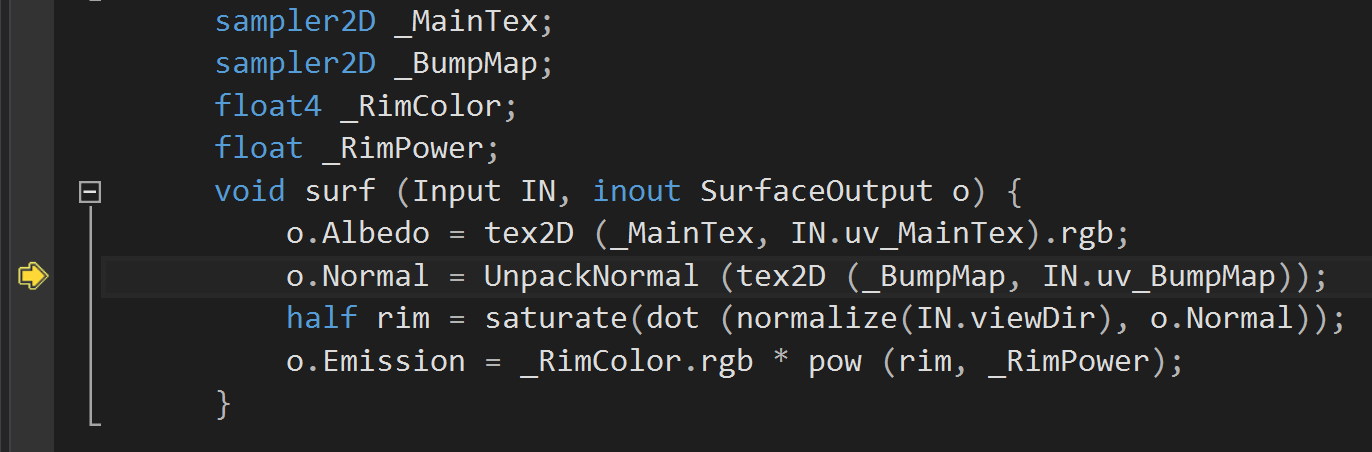
By the looks of it, the sphere looks right so the issue will probably be in the Pixel Shader where we give it color.

**Step 8: Debug shader**

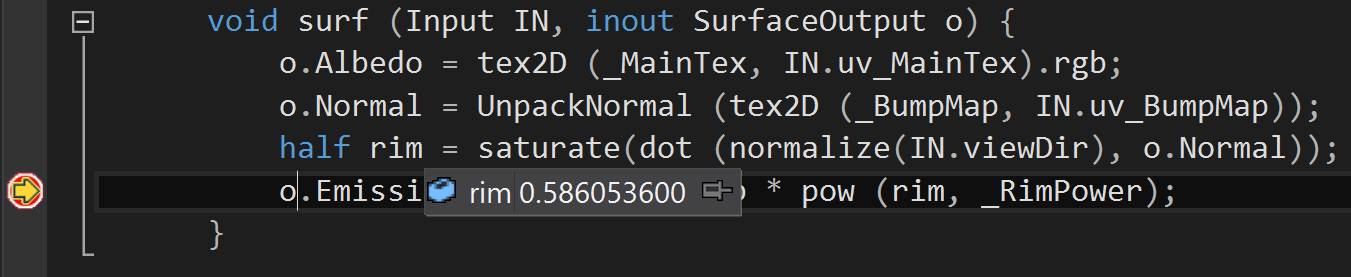
Now, below each of the stages in the Pipline Stage view, or in the Pixel History, you can see what shader is being used, as well as a green play button. By clicking on the shader name, you can see the code of the shader. This shader will create a rim around the moon, but by the looks of it, the rim color seems to be in the center, inverse.

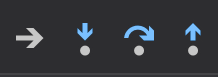
We are interested in debugging it, so click the green play button next to the Pixel Shader (the one underlined below):  


The shader code will become visible, with the possibility to step through the shader:



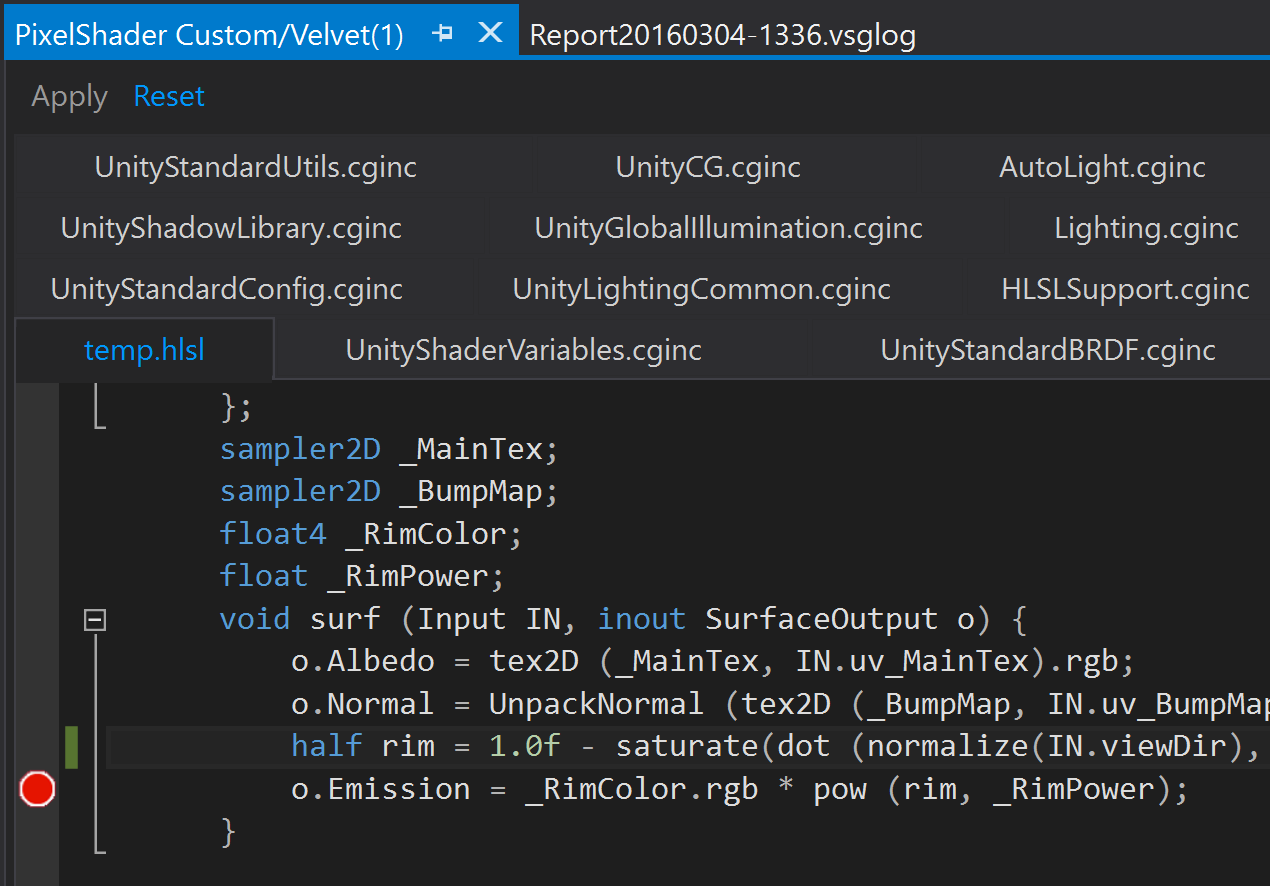
Set a breakpoint at the last line in the shader and step through until you reach this line. Look at the content of the variable named rim.



You can use the tools above the tab to step through the shader:  


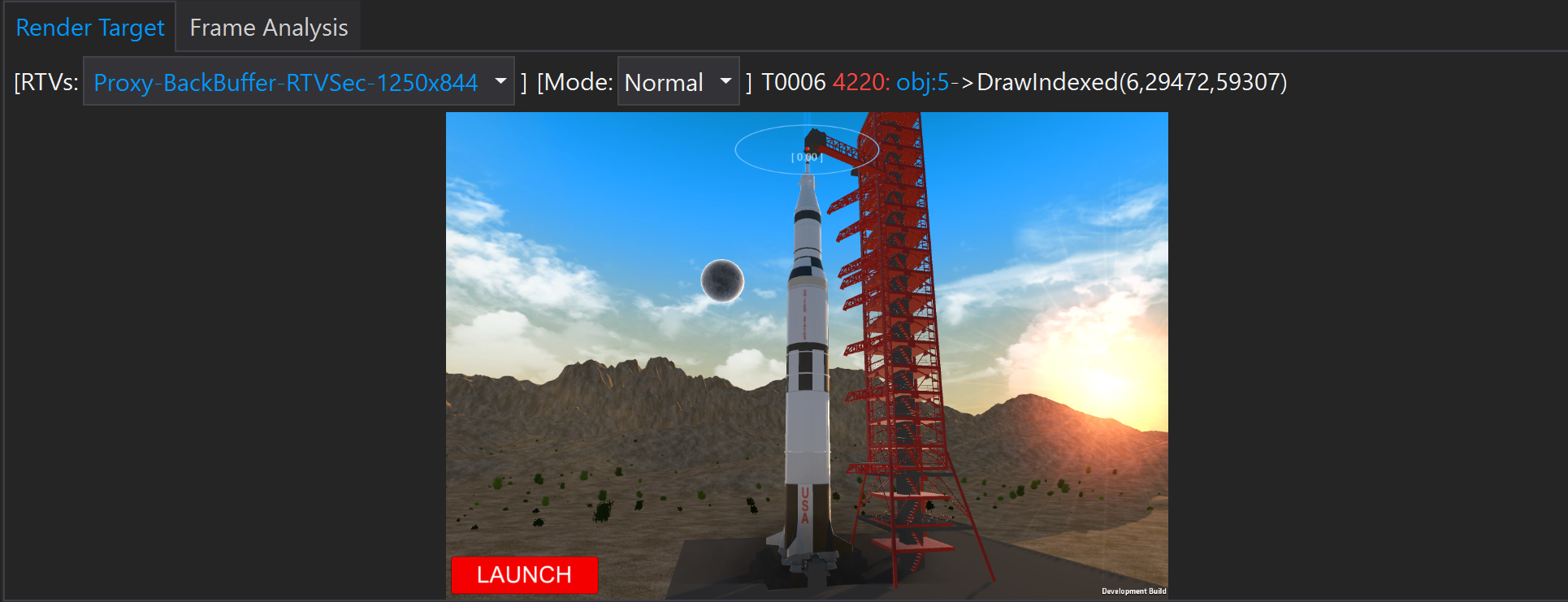
Looking at the calculations of the rim value, we have forgotten to invert it by subtracting it from one. Lets try this, so go ahead and stop the debugger and open the shader for editing.

**Step 9: Edit the shader**

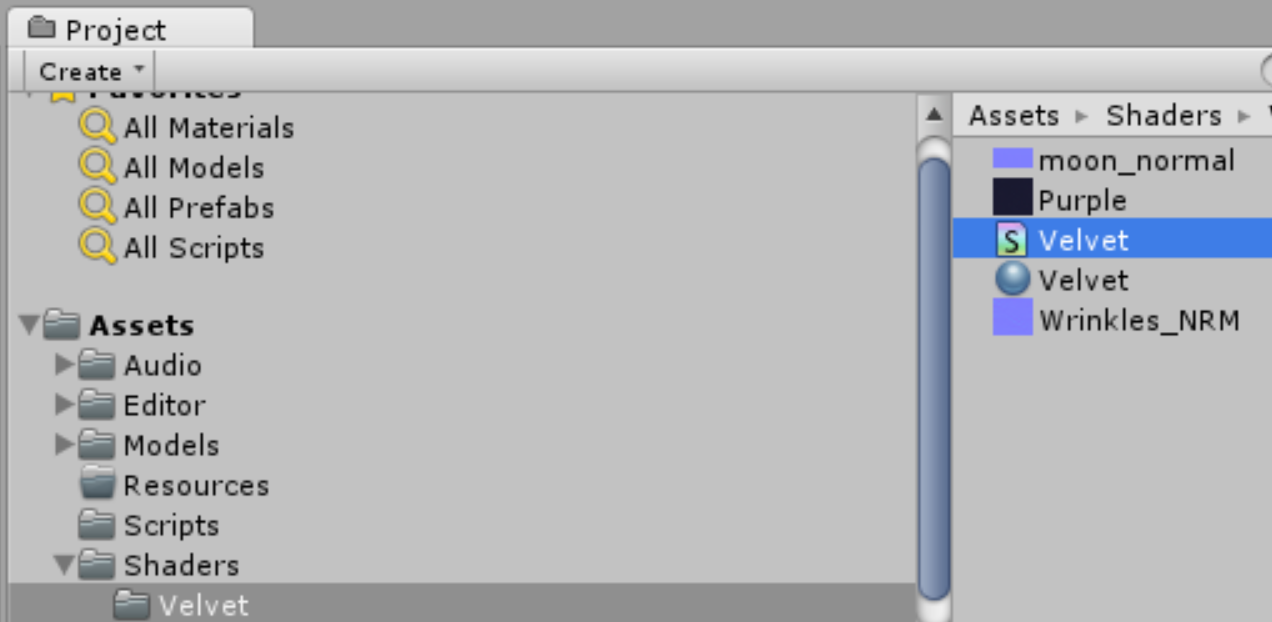
Click on the shader tab and edit the code so it looks like this:  


The only change is to write “1.0f – “ in front of saturate at the line that calculates the rim effect.

Save it, and the Frame Capture will update itself to using this shader. Click the frame preview tab, and notice that the moon is now fixed!



**Step 10: Fix issue in Unity and re-export**

Close the debugging session and go back to Unity, and find the Velvet shader we are using by navingating to the Shaders/Velvet folder:  


Open this in Visual Studio and make the same change to it, add **1.0f –** in front of saturate.

void surf (Input IN, inout SurfaceOutput o) {

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

o.Normal = UnpackNormal (tex2D (\_BumpMap, IN.uv\_BumpMap));

half rim = 1.0f - saturate(dot (normalize(IN.viewDir), o.Normal));

o.Emission = \_RimColor.rgb \* pow (rim, \_RimPower);

}

Save the change.

**Step 11: Test**

Play the game from the Unity Editor, and see that it now works well, and the moon is fixed!