First Person View
Version 3

Manual

(Don’t forget to rate the asset)

Introduction

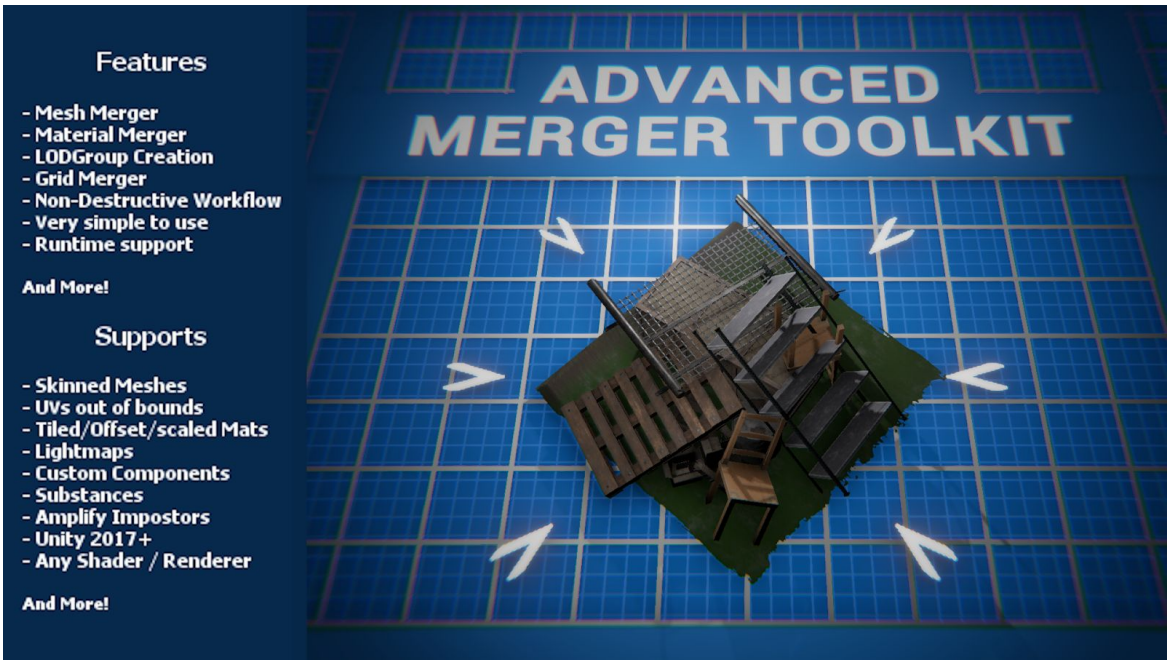
First Person View 3 offers a first person perspective where first person objects have a different field of view from the rest of the environment, won't clip the environment, and receive shadows from the environment.

With First Person View 3, solutions for Unity's Default Renderer, URP and HDRP are offered.

Every script is well documented to make it easier to learn and understand how it works.

Check out my other assets on the Asset Store!

Advanced Merger Toolkit http://u3d.as/1k1d



Youtube tutorial Videos https://www.youtube.com/watch?v=T6kc6xJViKg

PBR Game-Ready Desks Pack (FREE) http://u3d.as/19C0

First Person View 3 Features

Game Features

- First Person View models don't clip through the environment
- First Person View models receive shadows from the environment
- Independent Field-of-View between First Person View and the World View

Script Features

- Ability to automatically change between World View + First Person View and World View only. (useful when the player interacts with the environment)
- Ability to assign and remove objects to/from the First Person View perspective
- Automatic Layer assignment for First Person Objects. It preserves original layers when changing the objects back to World View
- Transform a point and/or direction from First Person View to World View
- Easily fade in/out of First Person View. Useful when interacting with the environment.

FPV Effects on models:

- First person models will have their shadow/light calculations stretched, due to the shortening of their depth on screen. In URP/HDRP you can control how much this can affect FPV models.

Default Renderer Solution:

- Due to Unity's default rendering pipeline, it's not possible to fully support temporal effects. Although, in order to not encounter some issues with using the motion vectors, we make it so we "disable" the motion, which resolves a few issues, but without the motion, some temporal effects won't simply work.

URP and HDRP:

- With Unity's new rendering pipeline, this limitation no longer applies! With the solution in FPV3 for URP and HDRP, you won't have any limitation with your post processing effects.

HDRP + URP Solution

If you are using the HDRP or URP in your Unity project, you'll want to follow these instructions.

HDRP and URP are only officially supported for Unity 2019.3 and up.

For the default renderer, Skip this section

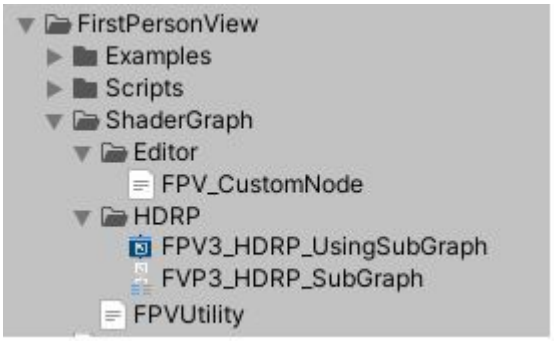
HDRP/URP Tutorial

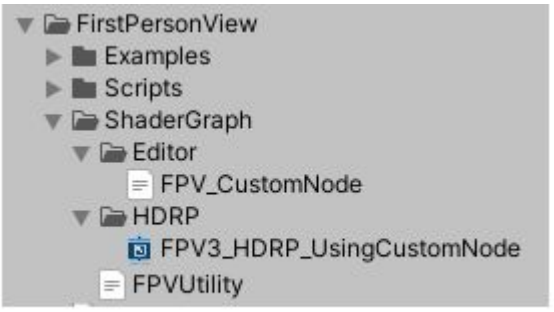
Step 1

Depending on if you're using HDRP or URP in your project, import the package related to the SRP you're using. Both solutions are the same, but need different assets compiled under their SRP.

Import the package FPV3_HDRP_2019_3 located at FirstPersonView/ShaderGraph.

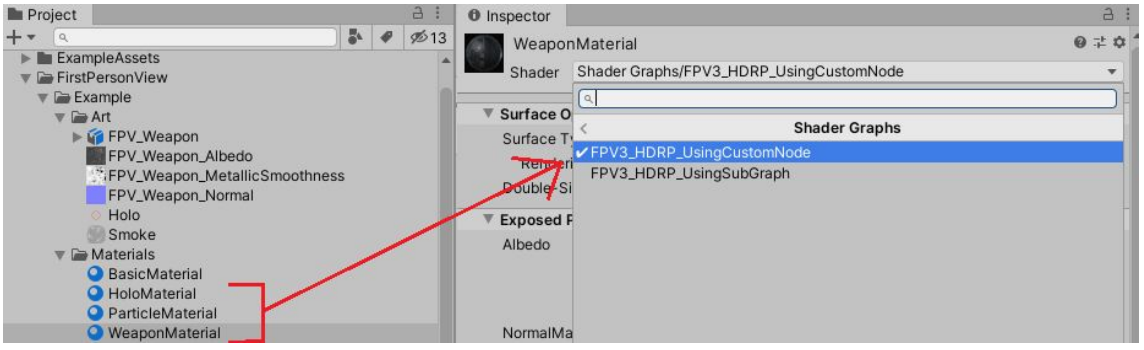
You should have either one like this after importing:





Step 2

Go to the example materials and change all of them to use the FPV3_HDRP_UsingSubGraph shader, except for the Default material. That one change it to the default HDRP/URP Lit shader:



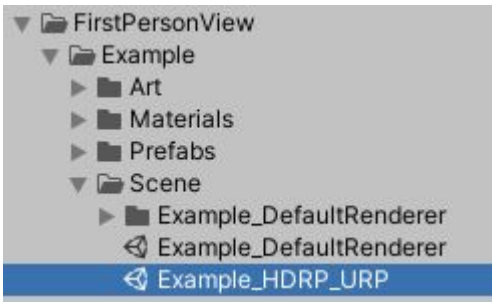
For the Holo and Particle materials, set them as transparent, and assign the Holo texture to the Albedo field of the HoloMaterial, and the Smoke texture to the Albedo field of the ParticleMaterial.

Step 3

Test the demo!

And that's it for the setup to test the demo provided with FPV3.

Open the demo scene at Examples/Scene/Example_HDRP_URP.



In the demo, you can walk around, aim, shoot, drop, grab the weapon.

You can see how everything was setup by checking the provided scripts (Will also be explained later on in the document)

Controls:

- WASD – Move
- Space – Jump
- Left Mouse Button – Fire
- Right Mouse Button – Aim
- "I" key – throw/grab weapon

How it works

FPV3 for HDRP and URP works by applying FoV and depth modification at the shader level, requiring a few modifications to the ShaderGraphs.

There are two ways to use FPV3: Using a custom shadergraph node, or a subGraph. Both of these do the same, but the custom node is more optimized and more future proof.

Custom ShaderGraph Node

In a nutshell, the custom FPV3 node function “reverts” the field of view of the projection at the shader level and applies the custom field of view.

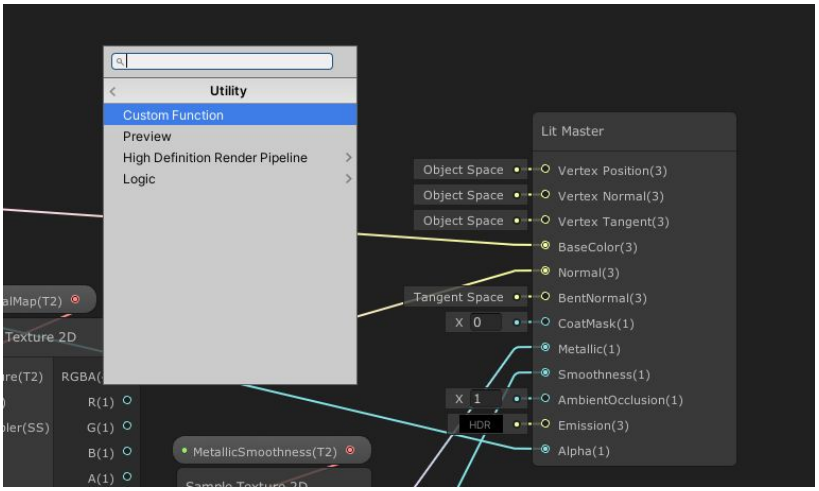
Code

The function can be found in ShaderGraph/FPVUtility.cs:

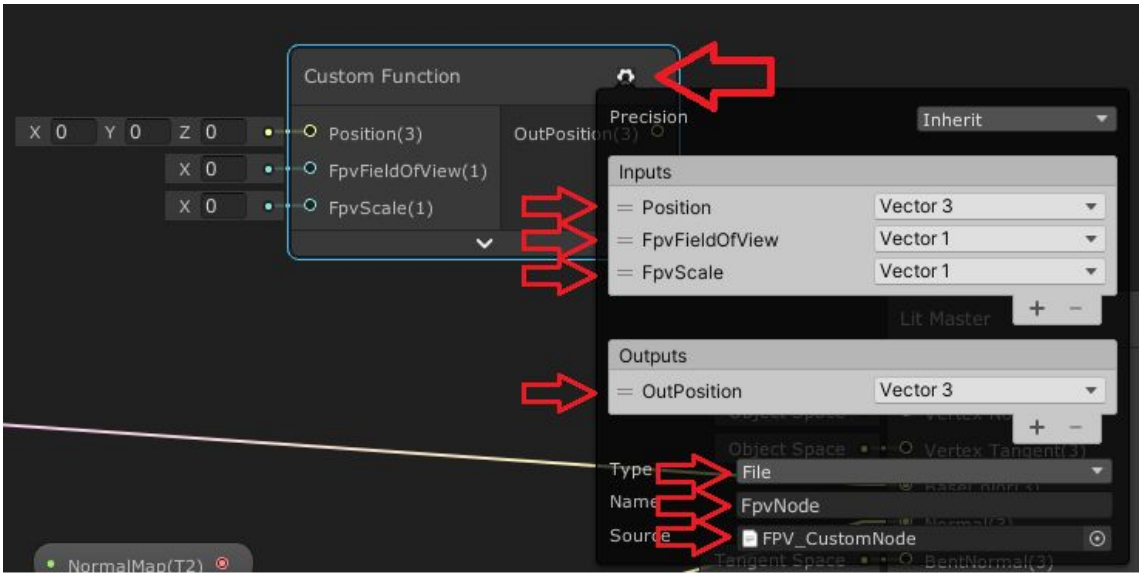
```
1   //UNITY_SHADER_NO_UPGRADE
2
3   #ifndef FPV_UTILITIES_INCLUDED
4   #define FPV_UTILITIES_INCLUDED
5
6   inline float3 ConvertPositionToFPV(float3 Position, float FpvFieldOfView, float FpvScale)
7   {
8       float FovProj = 1.0 / (-1.0 * tan((FpvFieldOfView / 360.0) * 3.14159265359f) * UNITY_MATRIX_P[1][1]);
9
10      float3 Pos_View = mul(mul(UNITY_MATRIX_V, UNITY_MATRIX_M), float4(Position, 1.0)).xyz;
11      Pos_View.x *= FovProj;
12      Pos_View.y *= FovProj;
13
14      Pos_View.x *= -_ProjectionParams.x;
15      Pos_View.y *= -_ProjectionParams.x;
16
17      Pos_View *= FpvScale;
18      Pos_View = mul(mul(UNITY_MATRIX_I_M, UNITY_MATRIX_I_V), float4(Pos_View, 1.0)).xyz;
19
20      return Pos_View;
21  }
22
23  inline float3 FPV_ConvertPositionToFPV(float3 Position, float FpvFieldOfView, float FpvScale) {
24  #if defined(BOOLEAN_FIRSTPERSONVIEW_ON) || defined(BOOLEAN_FPV_ALWAYS_ON)
25      return ConvertPositionToFPV(Position, FpvFieldOfView, FpvScale);
26  #else
27      return Position;
28  #endif
29  }
30
31  #endif
```

Using the node

To use the node, go to your ShaderGraph, or create a new one, and add the node “Utility/CustomFunction”:

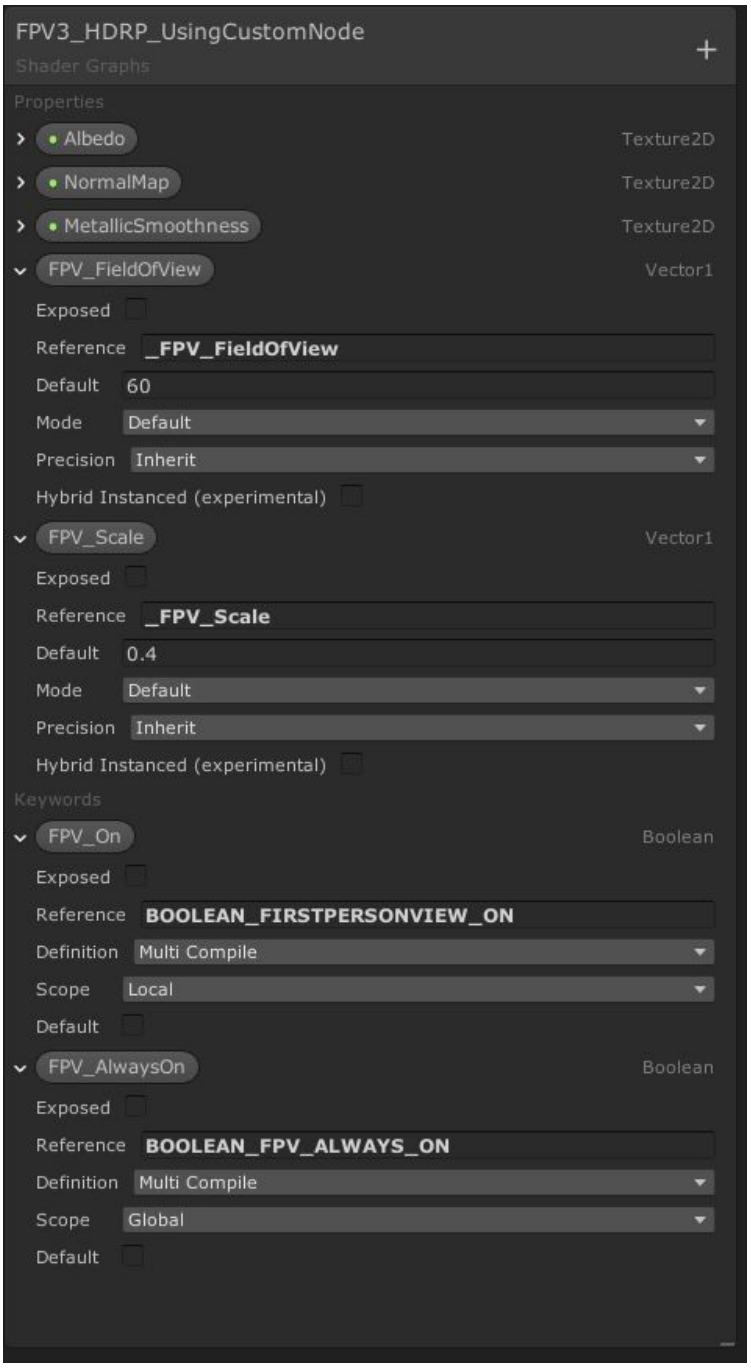


On the node, click on the setting icon, select the FPV_CustomNode file, use the name of the function “FpvNode” and add the parameters of the function to the inputs and outputs:



(Unity doesn’t do it automatically for some reason, so they need to be done manually)

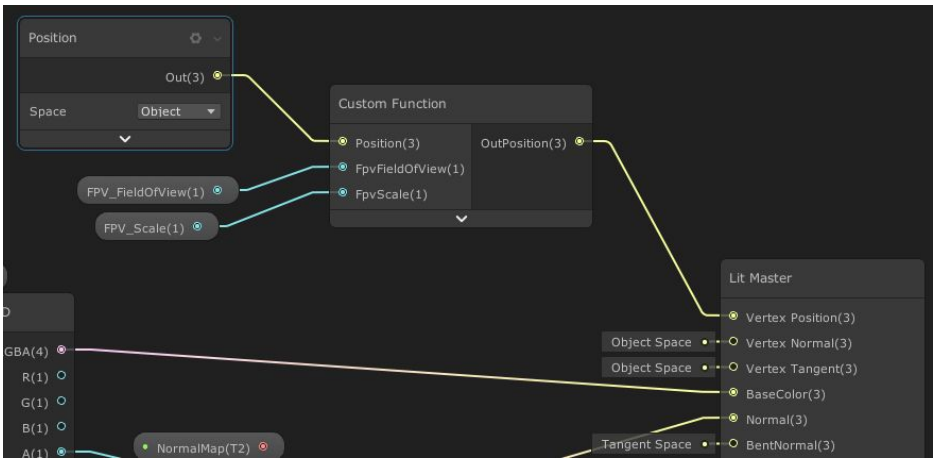
Next up we'll add 3 new Parameters to the shader graph itself, containing the FPV field of view, depth scale and a flag which tells the shader graph if FPV should be activated.



The last flag, "BOOLEAN_FPV_ALWAYS_ON", is not necessary if you're not planning to have that material with the "Always On" value from the FPV Camera parameter.

Make sure to use the same Reference names for each one, since the FPV scripts use those values to update them. Also make sure the Keywords are set to Multi-Compile.

Lastly, we assign those values to the Custom Node, add the Position Node to the Position input of the Custom Node, and connect it to the “Vertex Position” input of the “Lit Master” parameter:

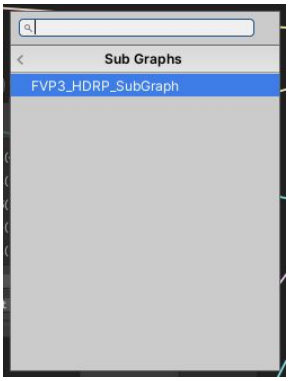


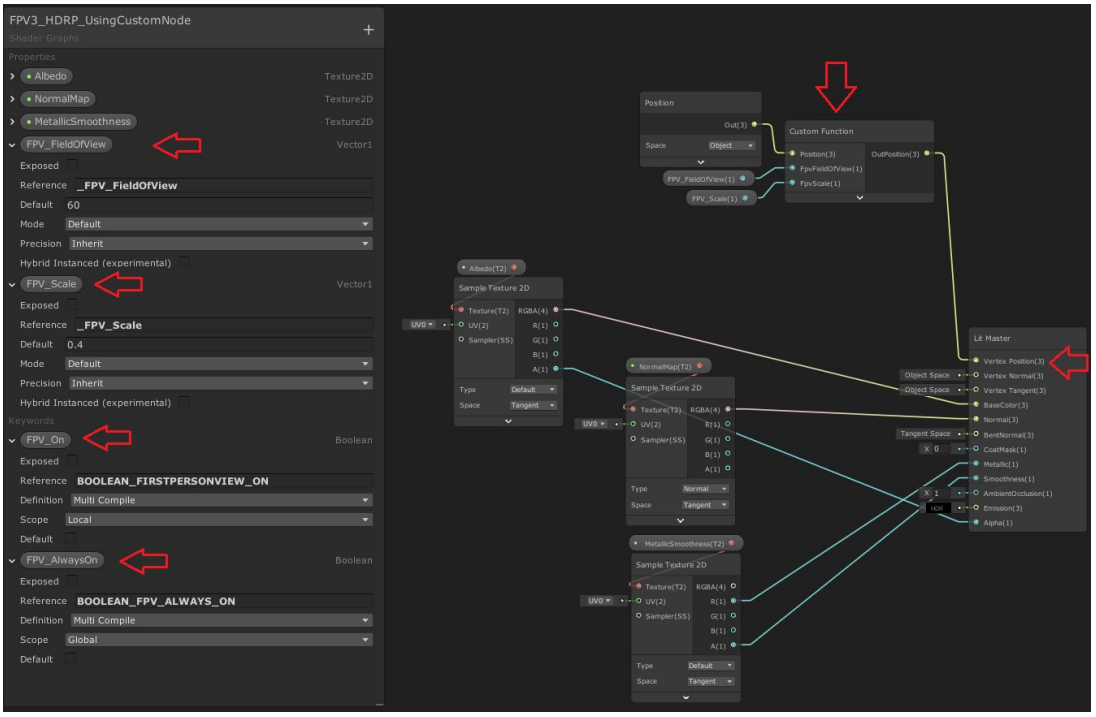
And that’s all that’s necessary to modify your ShaderGraph to support FPV3!

Custom SubGraph

The custom Subgraph works exactly the same as the custom Node, but it's made in a ShaderGraph SubGraph.

To use it, just drag the subGraph file into your shader graph, add the same variables to the graph, and link it all together just like the custom node:





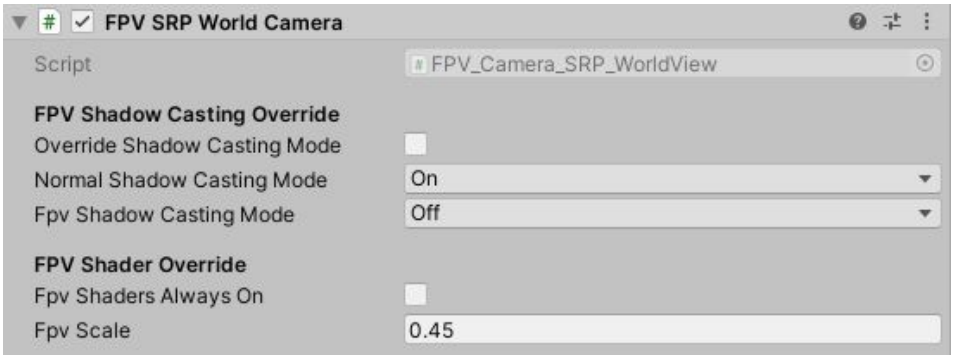
This SubGraph is easier to add to the ShaderGraph, since you don't need to manually add the parameters to the node.

Camera Setup

Since for HDRP and URP shaders we only provide the first person field of view, unlike the default renderer solution, then we use a different World View camera component attached to the World Camera.

SRP World Camera:

To setup the Camera for HDRP/URP, add FPV_Camera_SRP_WorldView component:



The “FPV Shaders Always On” sets the global keyword that forces all FPV shaders to always be active. This way you can manually swap the materials you want between non-FPV and FPV shaders.

The “FPV Scale” property will let you reduce the size of FPV objects at the shader level, allowing those objects to not clip through the environment.

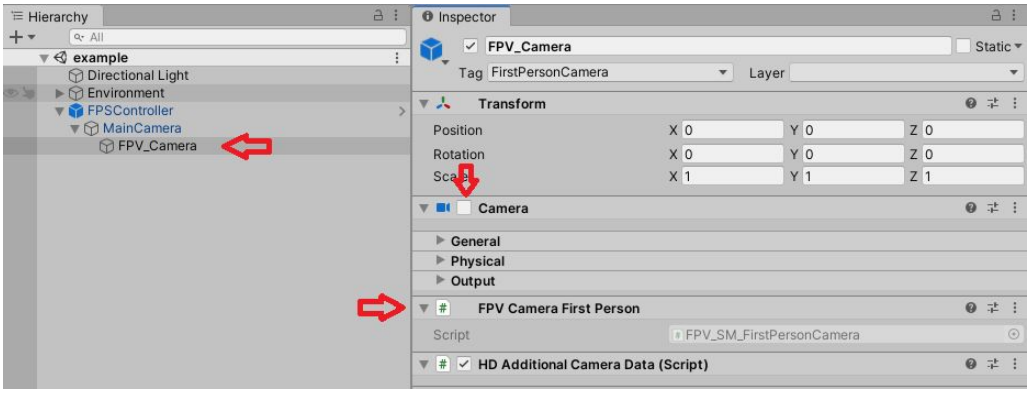
The smaller the number, the smaller the chance of not clipping the environment, but reducing it too much will introduce rendering problems, so the value should never be too small. The default value of 0.45 seems to work pretty well for most cases.

Second Camera:

To simplify various calculations to convert FPV points into world points, and vice versa, we add a dummy disabled Camera child to the SRP World Camera.

This camera will be the one used to change the FPV Field of View. Since it’s disabled, it won’t have any performance cost.

To setup this Camera, add a child Camera to the SRP World Camera, reset the Transform, deactivate the camera component and add the FPV_Camera_FirstPerson:



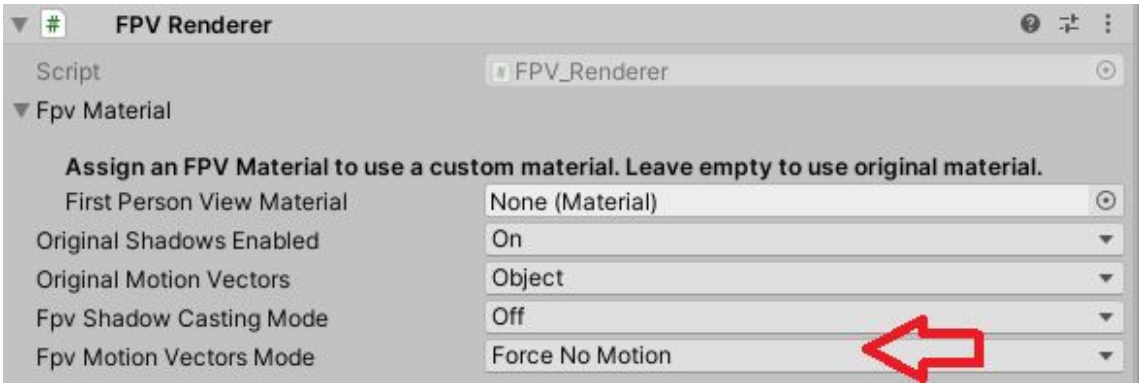
And this is all the setup needed for setting up the camera/shaders with FPV3 for HDRP and URP!

For more information on the rest of the components you'll need to set up, check the section Component Setup, after the Default Renderer Solution.

Skinned-Meshes, Depth and Motion Vectors

In Unity, if you're using FPV materials for skinned meshes and temporal effects, and you decrease the Depth value in the camera properties too much, you'll notice issues with those meshes. This is due to Unity not correctly applying vertex modifiers for skinned meshes's motion vectors.

If this is an issue on your end, you'll have to disable motion vectors specifically for skinned meshes. You do this by adding the FPV_Renderer component to the GameObject that contains that Skinned Mesh, and set the FPV Motion Vectors to "Force No Motion".



Some temporal effects might not work correctly with this change, like temporal motion blur, but TAA still works, although not as good.

Default Renderer Solution

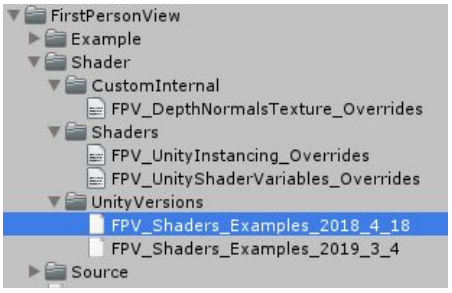
First Person View Tutorial

Step 1

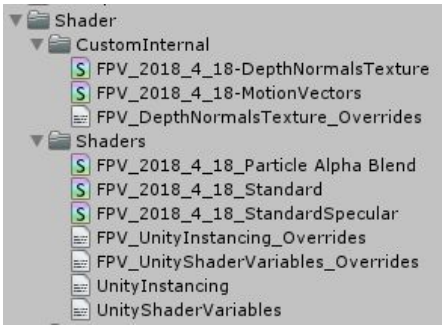
For this example, I'll be using Unity 2018.4.18, which is included in the asset. Later on, this manual will explain how you can set up FPV3 for every other Unity Version.

Inside the folder “Shader” you’ll encounter the folders:

- Custom Internal: Will contain the custom MotionVectors and DepthNormalTextures CGInc files.
- Shaders: Will contain the custom UnityShaderVariabes and UnityInstancing CGInc files, along with all your FPV shaders.
- Unity Versions: Contains the included examples already setup for the version in their name.



To start, import all files in this Unity Package, resulting in the following hierarchy:



All the files imported were taken from Unity’s Built-In Shaders and modified for FPV3. You can find these Built-in shaders on Unity’s Download website.

After import, right-click on the Shader folder and click re-import. This will make sure everything’s compiled.

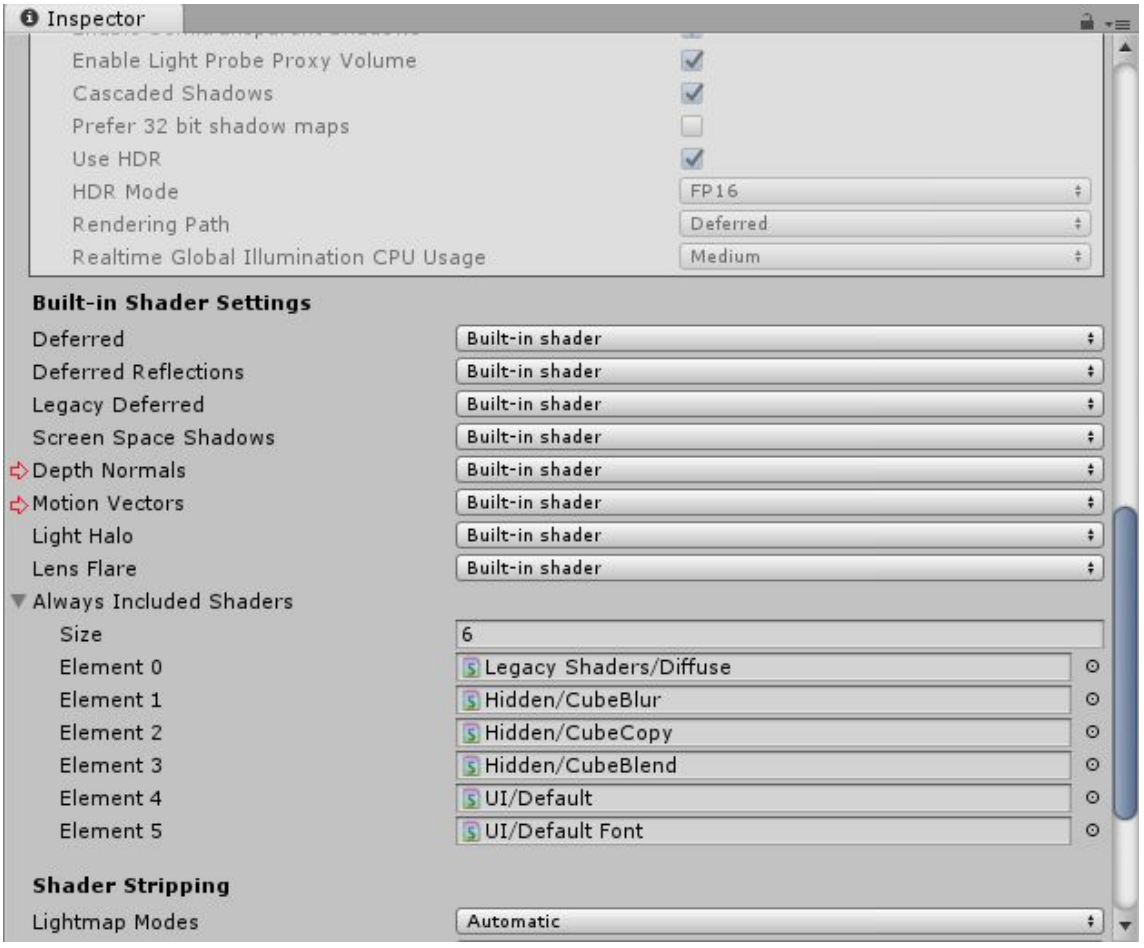
This will force the shader recompile and will use the custom UnityShaderVariables and UnityInstancing instead of Unity’s default one.

All FPV type shaders must be inside the same folder as the custom UnityShaderVariables.CGInc and UnityInstancing.CGInc.

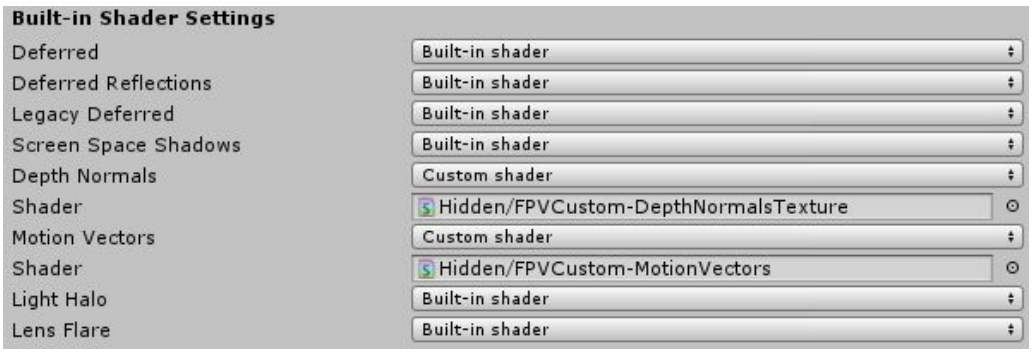
Step 2 (Depth Normals Texture and Motion Vectors)

In this step, we will be overwriting what Unity uses for DepthNormalsTexture and Motion Vector.

Go the to Graphics Window and scroll down to the shader options (Edit/ProjectSettings/Graphics)

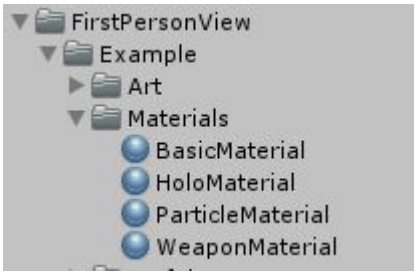


On the Depth Normals and Motion Vectors, click on the Built-in shader button and select the "Custom Shader" Option. Then, drag the correct shaders from the Custom Internal folder shown before to the correct place:



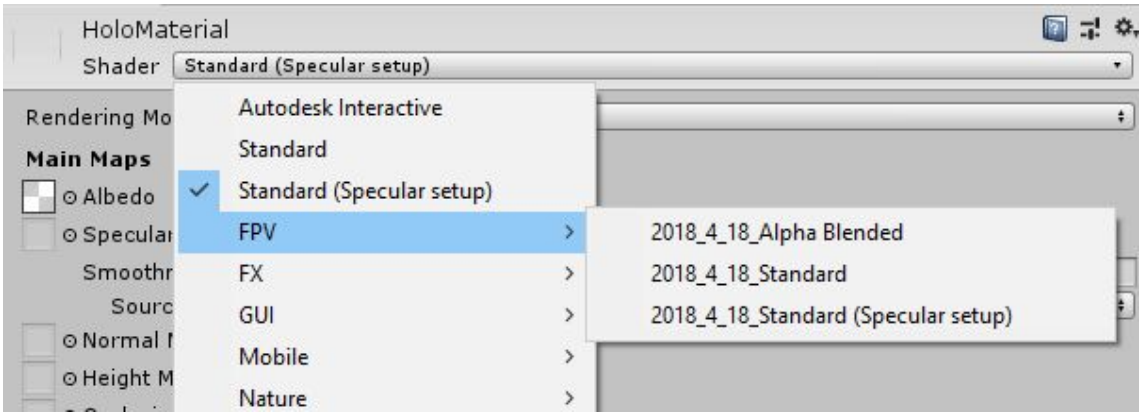
Step 3 (Material Setup)

Navigate to the folder that contains all FPV materials used in the demo:



For all materials, except “DefaultMaterial”, change their shader to the same major Unity version FPV shader as the editor.

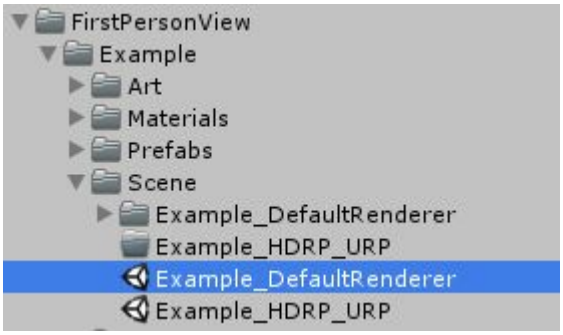
You can find all FPV shaders in the FPV shader section:



Step 4 (Test Demo)

And that's it for the setup to test the demo provided with FPV3.

Open the demo scene at Examples/Scene/Example_DefaultRenderer.



Make sure the camera is in Deferred Rendering mode for this demo.

In the demo, you can walk around, aim, shoot, drop, grab the weapon.

You can see how everything was setup by checking the provided scripts (Will also be explained later on in the document)

Controls:

- WASD – Move
- Space – Jump
- Left Mouse Button – Fire
- Right Mouse Button – Aim
- "I" key – throw/grab weapon

How It Works

In FPV3, the setup for Unity's CGinclude files has been greatly simplified by putting the custom logic and pre-processor defines into their own cginc files.

This allows us to simply include these files directly in Unity's CGInclude files in order to support FPV, not requiring any further change.

FPV_UnityShaderVariables_Overrides.cginc

```
1
2     //UNITY_SHADER_NO_UPGRADE
3
4   #ifndef FPV_OVERRIDE_INCLUDED
5     #define FPV_OVERRIDE_INCLUDED
6
7     float4x4 _firstPersonProjectionMatrix;
8
9   #if defined(BOOLEAN_FIRSTPERSONVIEW_ON) || defined(BOOLEAN_FPV_ALWAYS_ON)
10    #ifdef FPV_LIGHT // Used to enable/disable the projection matrix used during shadows calculations
11      #define UNITY_MATRIX_P glstate_matrix_projection
12
13    #else
14      #define UNITY_MATRIX_P _firstPersonProjectionMatrix
15    #endif
16
17    #define UNITY_MATRIX_VP mul(UNITY_MATRIX_P, UNITY_MATRIX_V)
18    #define UNITY_MATRIX_MVP mul(UNITY_MATRIX_VP, unity_ObjectToWorld)
19  #endif
20
21    #endif
```

This CGInc file includes the custom projection matrix used for FPV models, and includes a few pre-processor defines, which allows us to change what projection matrix is used for FPV models depending on if the FIRSTPERSONVIEW keyword is enabled in the Shader/Material.

UnityShaderVariables.cginc

(found in Unity's "builtin_shaders/CGIncludes/UnityShaderVariables.cginc")

```
336   #define UNITY_MATRIX_TEXTURE2 float4x4(1,0,0,0, 0,1,0,0, 0,0,1,0, 0,0,0,1)
337   #define UNITY_MATRIX_TEXTURE3 float4x4(1,0,0,0, 0,1,0,0, 0,0,1,0, 0,0,0,1)
338
339   #include "FPV_UnityShaderVariables_Overrides.cginc"
340
341   #endif
342
```

In UnityShaderVariables, we simply include the "FPV_UnityShaderVariables_Overrides.cginc" at the end of the file. This will override all of the pre-processor defines from UnityShaderVariables when the FIRSTPERSONVIEW keyword is activated.

FPV_UnityInstancing_Overrides.cginc

```
1
2     //UNITY_SHADER_NO_UPGRADE
3
4   #ifndef FPV_INSTANCING_OVERRIDE_INCLUDED
5     #define FPV_INSTANCING_OVERRIDE_INCLUDED
6
7   #if defined(BOOLEAN_FIRSTPERSONVIEW_ON) || defined(BOOLEAN_FPV_ALWAYS_ON)
8     unity_MatrixMVP_Instanced = mul(UNITY_MATRIX_VP, unity_ObjectToWorld);
9     #endif
10
11    #endif
```

For UnityInstancing, we only need to modify a specific matrix variable so that is uses our custom projection matrix in UnityShaderVariables.

UnityInstancing.cginc

(found in Unity's "builtin_shaders/CGIncludes/UnityInstancing.cginc")

For UnityInstancing.cginc, we include the above cginc inside the function UnitySetupCompoundMatrices(), at the end of the file, right after the unity_MatrixMVP_Instanced is assigned.

Our include will override that variable if the FIRSTPERSONVIEW keyword is enabled.

```
377         static float4x4 unity_MatrixTMV_Instanced;
378         static float4x4 unity_MatrixITMV_Instanced;
379         void UnitySetupCompoundMatrices()
380         {
381             unity_MatrixMVP_Instanced = mul(unity_MatrixVP, unity_ObjectToWorld);
382 #include "FPV_UnityInstancing_Overrides.cginc"
383             unity_MatrixMV_Instanced = mul(unity_MatrixV, unity_ObjectToWorld);
384             unity_MatrixTMV_Instanced = transpose(unity_MatrixMV_Instanced);
385             unity_MatrixITMV_Instanced = transpose(mul(unity_WorldToObject, unity_MatrixInvV));
386         }
387         #undef UNITY_MATRIX_MVP
388         #undef UNITY_MATRIX_MV
```

DepthNormalsTexture.shader

(found in Unity's "builtin_shaders/DefaultResourcesExtra/internal-DepthNormalsTexture.shader")

```
36      }
37
38      SubShader {
39          Tags { "RenderType"="FPV" }
40          Pass {
41      CGPROGRAM
42      #pragma vertex vert
43      #pragma fragment frag
44      #include "UnityCG.cginc"
45      #include "FPV_DepthNormalsTexture_Overrides.cginc"
46      ENDCG
47          }
48      }
49
50      SubShader {
51          Tags { "RenderType"="TransparentCutout" }
```

The changes performed on the Depth Normals Texture shader fixes issues with Forward type shaders and normals.

This happens because when Unity calculates the Depth Normals Texture, it uses Unity's own UnityShaderVariables and not our custom one, so things are not aligned on the screen.

To fix this, we add a new SubShader used for non-transparent shaders. Our FPV shaders should then use this new "FPV" tag.

FPV_DepthNormalsTexture_Overrides.cginc

```
1
2     //UNITY_SHADER_NO_UPGRADE
3
4   #ifndef FPV_DEPTHNORMALSTEXTURE_OVERRIDE_INCLUDED
5     #define FPV_DEPTHNORMALSTEXTURE_OVERRIDE_INCLUDED
6
7   struct v2f {
8       float4 pos : SV_POSITION;
9       float4 nz : TEXCOORD0;
10      UNITY_VERTEX_OUTPUT_STEREO
11    };
12
13    float4x4 _firstPersonProjectionMatrixCustom;
14
15  v2f vert( appdata_base v ) {
16      v2f o;
17      UNITY_SETUP_INSTANCE_ID(v);
18      UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(o);
19      o.pos = mul(mul(_firstPersonProjectionMatrixCustom, UNITY_MATRIX_V), mul(unity_ObjectToWorld, v.vertex));
20      o.nz.xyz = COMPUTE_VIEW_NORMAL;
21      o.nz.w = COMPUTE_DEPTH_01;
22      return o;
23    }
24  fixed4 frag(v2f i) : SV_Target {
25      return EncodeDepthNormal (i.nz.w, i.nz.xyz);
26    }
27
28    #endif
29
```

This file contains a direct copy of the SubShader for Opaque render type in DepthNormalsTexture shader, modifying only the projection matrix used to calculate the vertex positions.

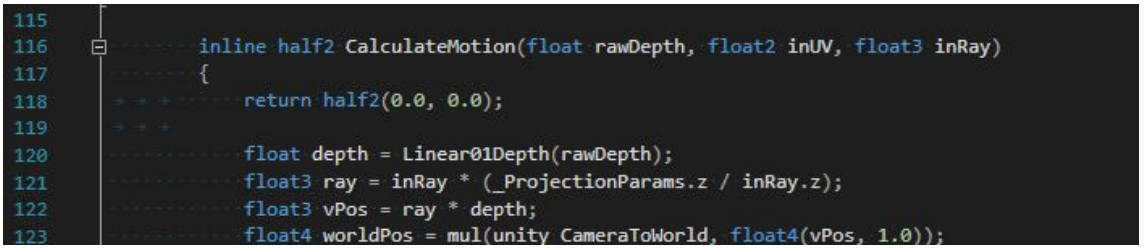
MotionVectors.shader

(found in Unity's "builtin_shaders/DefaultResourcesExtra/internal-MotionVectors.shader")

Because the FPV solution for default renderer does not fully support motion vectors, we simply need to "disable" the motion produced by this shader.

Temporal effects will be affected by this in some ways, for example TAA will not have much of anti-aliasing when the camera moves rapidly.

To apply this change, simply add "return half2(0.0, 0.0);" to the start of the function CalculateMotion:



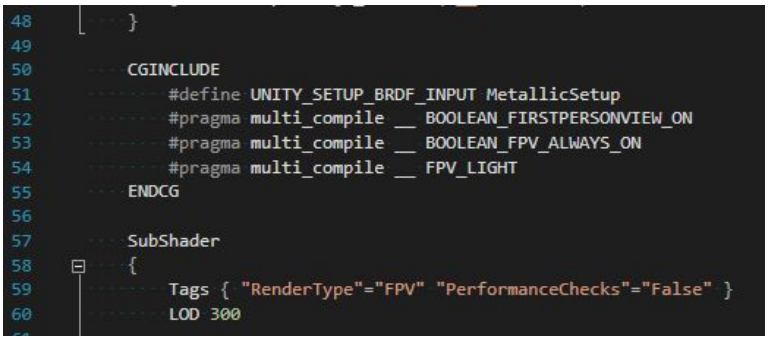
Shaders

This whole process was done so that the process of altering the shaders becomes easy.

The process is similar to different types of shaders.

To modify the shaders into FPV shaders, we just need to add the following:

Standard Specular Shader

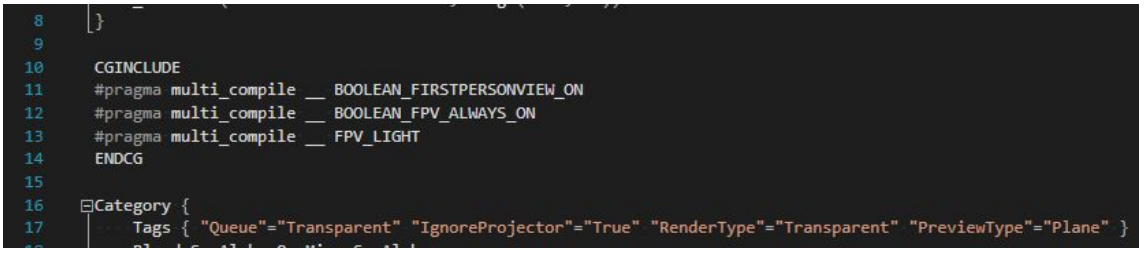


```
48      }
49
50      CGINCLUDE
51          #define UNITY_SETUP_BRDF_INPUT MetallicSetup
52          #pragma multi_compile __ BOOLEAN_FIRSTPERSONVIEW_ON
53          #pragma multi_compile __ BOOLEAN_FPV_ALWAYS_ON
54          #pragma multi_compile __ FPV_LIGHT
55      ENDCG
56
57      SubShader
58      {
59          Tags { "RenderType"="FPV" "PerformanceChecks"="False" }
60          LOD 300
```

We add the “#pragma multi_compile __ FIRSTPERSONVIEW” and “#pragma multi_compile __ FPV_Light” to the CGINCLUDE section of the shader.

We then add the tag “RenderType”=”FPV” to the tags section of the SubShader.

Particle Alpha



```
8   }
9
10  CGINCLUDE
11  #pragma multi_compile __ BOOLEAN_FIRSTPERSONVIEW_ON
12  #pragma multi_compile __ BOOLEAN_FPV_ALWAYS_ON
13  #pragma multi_compile __ FPV_LIGHT
14  ENDCG
15
16  Category {
17      Tags { "Queue"="Transparent" "IgnoreProjector"="True" "RenderType"="Transparent" "PreviewType"="Plane" }
```

Camera Setup

First Person View uses just one active camera and a disabled camera.

The disabled camera is used to get the first person projection matrix with ease.

The main camera will contain the component FPV_Camera_DR_WorldView, that is responsible for updating the global shader variables and preparing command buffers for shadow map calculations.

If you want correct FPV shadows onto the environment, and are in Deferred mode, enable the “Cast Correct Fpv Shadows” toggle on the MainCamera component.

The FPV_Camera object will be inside the MainCamera Object, and it will have its Camera component disabled. It will also have the FPV_Camera_DR_FirstPerson component attached to it.

This component is used for the MainCamera to easily access the FPV camera properties and for easily change the FOV of the camera.

FPV_Camera_DR_WorldView

This component will update the FPV projection matrix on the shaders among other things.

```
39      void OnPreCull() {
40          //Before rendering the scene, update the global first person projection on the shaders.
41          Matrix4x4 fpvProjection = FPV.firstPersonCamera.GetProjectionMatrix();
42
43          Matrix4x4 fpvProjectionCustom = fpvProjection;
44
45          //DIRECTX
46          if (d3d) {
47              fpvProjection.m11 *= -1; // Invert Y value of the MVP vertex position
48              fpvProjectionCustom.m11 *= -1; // Invert Y value of the MVP vertex position
49
50              if (usesReversedZBuffer) { //Inverted Z-Buffer
51                  fpvProjection.m22 *= 0.3f; // Shorten the Z value of the MVP vertex position
52                  fpvProjection.m23 *= -1;
53                  fpvProjectionCustom.m23 *= -1;
54              }
55              else { //Z-Buffer not inverted
56                  fpvProjection.m22 *= 0.7f; // Shorten the Z value of the MVP vertex position
57              }
58          }
59          //OPEN GL
60          else {
61              if (usesReversedZBuffer) { //Inverted Z-Buffer
62                  fpvProjection.m22 *= 0.3f; // Shorten the Z value of the MVP vertex position
63                  fpvProjection.m23 *= -1;
64                  fpvProjectionCustom.m23 *= -1;
65              }
66              else { //Z-Buffer not inverted
67                  fpvProjection.m22 = fpvProjection.m22 * 0.7f + 0.3f; // Shorten the Z value of the MVP vertex position
68              }
69          }
70
71          //Update the global shader variable
72          Shader.SetGlobalMatrix("_firstPersonProjectionMatrix", fpvProjection);
73          Shader.SetGlobalMatrix("_firstPersonProjectionMatrixCustom", fpvProjection);
74      }
```

Inside the PreCull method, we update both global shader matrices with the FPV camera projection matrix modified so its Z value is shortened.

```
76      private void PrepareLightingCommandBuffer() {
77          _commandBufferBefore = new CommandBuffer();
78          _commandBufferAfter = new CommandBuffer();
79
80          _commandBufferBefore.EnableShaderKeyword("FPV_LIGHT");
81          _commandBufferAfter.DisableShaderKeyword("FPV_LIGHT");
82
83          _camera.AddCommandBuffer(CameraEvent.BeforeLighting, _commandBufferBefore);
84          _camera.AddCommandBuffer(CameraEvent.AfterLighting, _commandBufferAfter);
85      }
```

This method prepares the command buffers used to force lights to use the default UNITY_MATRIX_P matrix.

This is only done if the public variable “castCorrectFpvShadows” is enabled.

FPV Shadows

To be able to have correct FPV shadows onto the environment, we need to “reset” the projection matrix that is used to render each FPV Object.

Since we override the main shader variables on the UnityShaderVariables.CGinc like shown before, we need a way to revert that change when it’s time for the lights to calculate their shadow map.

This method uses Command Buffers, but it’s slightly different between Deferred Rendering and Forward Rendering.

Note:

Remember that if you don’t need/want FPV shadow casting, then never use these components, and make sure it’s disabled so it won’t add any command buffers that could affect performance.

Deferred Rendering

In Deferred Rendering, we can fix this by adding two command buffers on the Camera Component.

```
76      private void PrepareLightingCommandBuffer() {
77          _commandBufferBefore = new CommandBuffer();
78          _commandBufferAfter = new CommandBuffer();
79
80          _commandBufferBefore.EnableShaderKeyword("FPV_LIGHT");
81          _commandBufferAfter.DisableShaderKeyword("FPV_LIGHT");
82
83          _camera.AddCommandBuffer(CameraEvent.BeforeLighting, _commandBufferBefore);
84          _camera.AddCommandBuffer(CameraEvent.AfterLighting, _commandBufferAfter);
85      }
```

We simply enable the FPV_LIGHT shader keyword, shown in the UnityShaderVariables section, and that will revert the UNITY_MATRIX_P variable to use the default matrix, which the lights use to calculate their shadow maps.

We enable this keyword just before the Lighting Pass and then disable the keyword after the Lighting Pass.

If you don’t want FPV objects to cast shadows, simply disable shadow casting AND disable the variable “castCorrectFpvShadows” on the FPV_WorldCamera component. This way the command buffers aren’t used and there is no overhead.

Forward Rendering

In forward rendering we can't use Command Buffers on the Camera because there isn't an available event that doesn't interfere with the rendering of the objects.

So, for forward rendering, we need to create command buffers for the lights.

By attaching the FPV_LightCommandBuffer to any light source, and assigning the light component to the light variable field on the component, it will automatically add the Command Buffers on the Start method.

```
6   public class FPV_LightCommandBuffer : MonoBehaviour
7   {
8       [Header("Only use this script when using Forward Rendering")]
9       [SerializeField]
10      private Light _light;
11
12      private CommandBuffer _commandBufferBefore;
13      private CommandBuffer _commandBufferAfter;
14
15      void Start() {
16          _commandBufferBefore = new CommandBuffer();
17          _commandBufferAfter = new CommandBuffer();
18
19          _commandBufferBefore.EnableShaderKeyword("FPV_LIGHT");
20          _commandBufferAfter.DisableShaderKeyword("FPV_LIGHT");
21
22          _light.AddCommandBuffer(LightEvent.BeforeShadowMapPass, _commandBufferBefore);
23          _light.AddCommandBuffer(LightEvent.AfterShadowMapPass, _commandBufferAfter);
24      }
25  }
```

For these Command Buffers, we add them before and after the Shadow Map Pass.

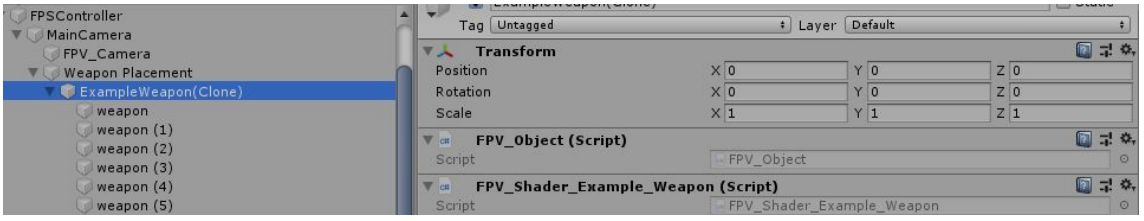
FPV Common Components

Component Setup

Next up, we will be looking at how the FPV objects are set up. We will use the provided ExampleWeapon prefab as the example on how to set up a simple weapon. (located at Examples/Example (...)/Prefabs/ExampleWeapon)

FPV_Object

The simplest way to prepare the object will be to attach the FPV_Object component to the root of the Object you want to be in FPV. This can be a weapon, attachment, character, etc.



This component will search through all child objects of the gameobject to find all renderers it contains and will add them to a list that is used to enable/disable the first person perspective for those objects.

This process won't include child objects that also contain another FPV_Object component, that way you can have things organised.

Note:

If you want to instantiate objects and dynamically add them to the first person perspective, you can add renderers directly to the FPV_Object component by using a public method called AddRenderer(Transform trans).

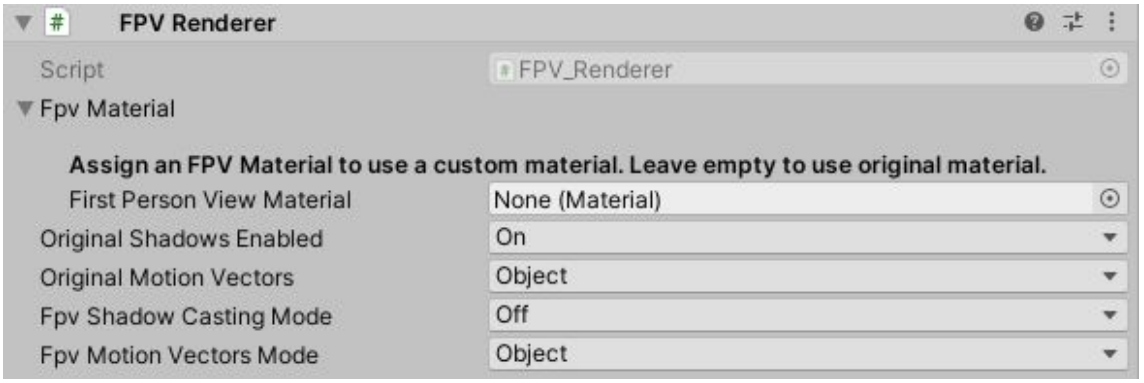
Make sure to remove the renderer if you destroy the object.

For each renderer found, this component will add a new component to each renderer. This will be either a FPV_Renderer if the renderer has just 1 material, or the FPV_Renderer_MultiMaterials if the renderer has more than 1 material.

If a Renderer already has one of those components, it will use them instead.

FPV_Renderer

This component is used to easily enable/disable the First Person Perspective on the renderer it is attached to.



By using a FPV shader, the shader can be in two modes: Normal or FIRSTPERSONVIEW.

When initializing this component, if the variable “fpvMaterial” is empty (e.g. we are using a FPV shader already), the component will duplicate the material inside the renderer, enable the FIRSTPERSONVIEW keyword on that material, and store it for later use.

When enabling FirstPersonPerspective, the component simply switches between the normal and FPV material.

When disabling the FirstPersonPerspective, the component switches the materials back.

FPV Material - If you don’t want to use the same material/shader for both normal and FPV mode, you can attach the FPV_Renderer component to the gameobject with the renderer component, and assign a FPV material to the “fpvMaterial” variable.

Original/FPV properties: By default, FPV will disable the shadows when FPV is active on this renderer, but you can override this in case you want different values for shadows/motion vectors when in/out of FPV mode.

FPV_Renderer_MultiMaterial

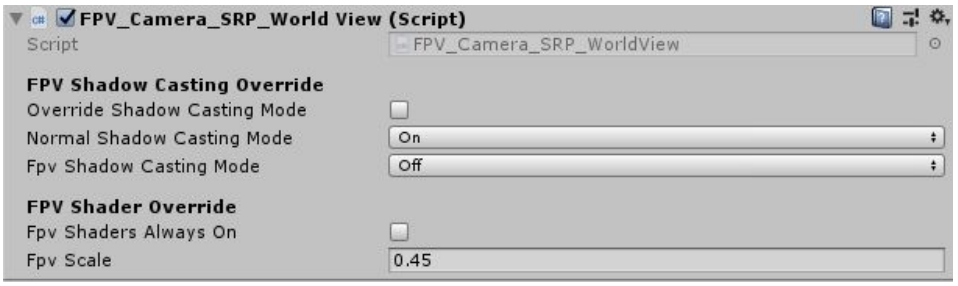
This component does basically the same as the FPV_SM_Renderer component, but instead it stores a list of normal and FPV materials.

When enabling/disabling FPV mode, it assigns the appropriate array to the renderer.

FPV Camera

For the camera, there are a few extra functions that can be used.

First, both types of cameras (SRP and Default Renderer) contain a variable “FPV Shaders Always On”. If this value is True, it will make all FPV materials On by default OnAwake, and can only be turned off by calling the “SetFpvAlwaysOn”:



By default FPV will turn off the shadow casting mode when in FPV, and re-enable it when disabling FPV. You can change this global behaviour by setting “Override Shadow Casting Mode” to True, and change the Normal and FPV Shadow Casting Mode.

You can have finer control for these properties in the FPV_Renderer Component if you want to control per Renderer.

```
public void SetFpvAlwaysOn(bool IsOn)
{
    if (IsOn)
    {
        Shader.EnableKeyword("BOOLEAN_FPV_ALWAYS_ON");
    }
    else
    {
        Shader.DisableKeyword("BOOLEAN_FPV_ALWAYS_ON");
    }
}
```

Next, in FPV3 you can now change the percentage of the FPV being used by the shaders by calling the functions available in the FPV Camera component. You can choose if you want to affect both FOV and Depth, or just one of them.

This is useful when you want to interact with the environment without the FPV applied, and when the interaction is done, change it back to FPV.

```
public void SetFirstPersonViewPercentage(float percentage)
{
    SetFirstPersonViewFovPercentage(percentage);
    SetFirstPersonViewDepthPercentage(percentage);
}
public virtual void SetFirstPersonViewFovPercentage(float percentage)
{
    _fpvFovPercentage = percentage;
}
public virtual void SetFirstPersonViewDepthPercentage(float percentage)
{
    _fpvDepthPercentage = percentage;
}
```

If you want to transition in/out of FPV through time, you can start the coroutine TransitionFirstPersonView, and pass in the values you want. This allows for the change to be performed in a given amount of time, instead of it being instant.

Note that while both TranslateFov and TranslateDepth work for HDRP/URP, only TranslateFov will work for the Default Renderer.

```
public IEnumerator TransitionFirstPersonView(float seconds, bool toFirstPerson, bool translateFov, bool translateDepth)
{
    float startTime = Time.unscaledTime;
    float endTime = startTime + seconds;

    while(true)
    {
        float currentTime = Time.unscaledTime;
        float percentage = (currentTime - startTime) / (endTime - startTime);
        float targetPercentage = toFirstPerson ? Mathf.Clamp01(percentage) : 1.0f - Mathf.Clamp01(percentage);

        if(translateFov){ SetFirstPersonViewFovPercentage(targetPercentage); }
        if(translateDepth){ SetFirstPersonViewDepthPercentage(targetPercentage); }

        if (percentage > 1.0f){ yield break; }

        yield return null;
    }
}
```

Static FPV Class

The static FPV class contains references to the MainCamera and the FirstPersonCamera for easy access to these important cameras.

```
public static class FPV
{
    /// <summary>
    /// Reference to the Main Camera that renders the environment
    /// </summary>
    public static IFPV_Camera mainCamera { get; set; }

    /// <summary>
    /// Reference to the First Person Camera.
    /// </summary>
    public static IFPV_Camera firstPersonCamera { get; set; }
```

Three important methods are provided in this static class:

```
26          public static Vector3 FPVPointToWorldPoint(Vector3 point)
27          {
28              point = firstPersonCamera.GetCamera().WorldToScreenPoint(point);
29              return mainCamera.GetCamera().ScreenToWorldPoint(point);
30          }
```

FPVPointToWorldPoint converts a 3D point from the “view” of the First Person Camera into a correct point that would be seen in the same place in the MainCamera.

This is useful to find the “normal” position of some FPV point. Since the FPV view is not the same as the MainCamera view, if you spawn normal objects, do raycasts, or other things using a FPV object, these won’t look correct in the MainCamera view.

For example, if you want to spawn a bullet at the end of the FPV barrel, you will need to find the real position of that barrel as it is viewed by the FPV camera. You can find that real point by calling the method above.

```
39      public static void FPVToWorld(Vector3 point, Vector3 direction, out Vector3 resPoint, out Vector3 resDirection)
40      {
41          resPoint = FPVPointToWorldPoint(point);
42
43          Vector3 pointForward = point + direction;
44          pointForward = FPVPointToWorldPoint(pointForward);
45          resDirection = pointForward - resPoint;
46      }
```

FPVToWorld converts a point and direction from a first person perspective into the normal world.

Again, since the FPV mode distorts where the weapon is actually pointing at, you will need to also find the correct direction of the barrel as it is seen through the FPV camera.

```
54      public static void FPVToWorld(Transform trans, out Vector3 resPoint, out Vector3 resDirection)
55      {
56          FPVToWorld(trans.position, trans.forward, out resPoint, out resDirection);
57      }
```

The last method, FPVToWorld, does the same thing as the FPVToWorld above, but takes in a Transform. This simplifies things if you want to use a transform to get its correct position and direction in the real world.

These methods are used by the demo scene, and are used to get the correct position and direction of the barrel to do projectile raycasting. This will be shown later on.

Enabling First Person View

Using the Demo scene as an example, let's take a look at the ExampleWeapon prefab.

Example Weapon

The main gameobject contains the FPV_Object component, so all renderers inside the object will automatically be included in this component, and it also contains the game logic component FPV_Shader_Example_Weapon.

Explaining only the important parts, the FPV_Shader_Example_Weapon contains a property IFPV_Object fpvObject, that when the component is setup, it will get and assign the FPV_Object component from the gameObject.

It then contains a SetWeaponOnPlayer method, that will set it as a FPV object, and at the end of the method, we will call the fpvObject.SetAsFirstPersonObject() to enable the FPV mode for the weapon.

```
34      public void SetWeaponOnPlayer(Transform player)
35      {
36          transform.SetParent(player);
37          transform.localPosition = Vector3.zero;
38          transform.localRotation = Quaternion.identity;
39
40          wRigidbody.isKinematic = true;
41          wCollider.enabled = false;
42
43          isOnPlayer = true;
44
45          fpvObject.SetAsFirstPersonObject();
46      }
```

It also contains a SetWeaponOnWorld method that will disable the weapon as a FPV object.

```
48      public void SetWeaponOnWorld()
49      {
50          transform.parent = null;
51
52          wRigidbody.isKinematic = false;
53          wCollider.enabled = true;
54
55          isOnPlayer = false;
56
57          fpvObject.RemoveAsFirstPersonObject();
58      }
```

To fire the weapon, we use the spawnpoint Transform to get the correct 3D point and direction of the barrel, and perform a raycast to instantiate a simple object.

Without this process, the direction and origin of the raycast would be wrong, and the object would be instantiated at the wrong location, due to the FPV camera having a different FOV.

Example Player

The player gameobject, that also contains both cameras, also contains the FPV_Shader_Example_Player component attached to it.

On Start, the component instantiates the weapon and automatically attaches it to the player. Since it attaches to the player, we want to also enable the FPV mode for the weapon.

```
78      private void CreateWeapon()
79      {
80          if (weapon != null) return;
81
82          GameObject newWeapon = GameObject.Instantiate(weaponPrefab);
83
84          weapon = newWeapon.GetComponent<FPV_Shader_Example_Weapon>();
85          weapon.Setup();
86          SetWeaponOnPlayer();
87      }
```

```
112     private void SetWeaponOnPlayer()
113     {
114         weapon.SetWeaponOnPlayer(weaponPlacement);
115     }
```

So, we first instantiate the weapon, call its setup method, and call the SetWeaponOnPlayer method that was described before.

The Demo scene also allows you to drop and grab the weapon, and when doing this, we also simply enable/disable the FPV mode on the weapon.

```
100         private void SwitchWeapon()
101         {
102             if (weapon.IsOnPlayer())
103             {
104                 SetWeaponOnWorld();
105             }
106             else
107             {
108                 SetWeaponOnPlayer();
109             }
110         }
111
112         private void SetWeaponOnPlayer()
113         {
114             weapon.SetWeaponOnPlayer(weaponPlacement);
115         }
116
117         private void SetWeaponOnWorld()
118         {
119             weapon.SetWeaponOnWorld();
120         }
```

When aiming, we simply modify the first person camera and the main camera's field of view by accessing the fieldOfView variable on each Camera component.

```
38      private void Aim()
39      {
40          if (weapon == null) return;
41
42          bool aiming = Input.GetMouseButton(1);
43
44          weapon.Aim(aiming, worldCamera.transform);
45
46          float futureFoV = aiming ? 20 : 70;
47
48          fpvCamera.fieldOfView = Mathf.Lerp(fpvCamera.fieldOfView, futureFoV, Time.deltaTime * 10);
49      }
50
51      private void ChangeFOV()
52      {
53          //World Camera FOV
54          float fOVChange = 0;
55          if (Input.GetKey(KeyCode.Comma))
56          {
57              fOVChange = -Time.deltaTime * 10;
58          }
59          else if (Input.GetKey(KeyCode.Period))
60          {
61              fOVChange = Time.deltaTime * 10;
62          }
63          worldCamera.fieldOfView = Mathf.Clamp(worldCamera.fieldOfView + fOVChange, 50, 120);
64
65          //First Person Camera FOV
66          fOVChange = 0;
67          if (Input.GetKey(KeyCode.N))
68          {
69              fOVChange = -Time.deltaTime * 10;
70          }
71          else if (Input.GetKey(KeyCode.M))
72          {
73              fOVChange = Time.deltaTime * 10;
74          }
75          fpvCamera.fieldOfView = Mathf.Clamp(fpvCamera.fieldOfView + fOVChange, 4, 70);
76      }
```