First Person View
Version 2.1

Manual

(Don’t forget to rate the asset)

Introduction

First Person View is an asset that offers a solution for first person perspective in Unity, where first person objects won't clip through the environment, have a separate field of view and receive shadows from the environment.

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

Because this asset includes examples from Unity's Standard Assets, it can cause problems in some projects, so don't import the example folder in your project. You should check the examples in a separate project to avoid any issues.

Check out my other asset on the Asset Store!

My other asset is called Object Construction Kit, and allows you to create highly detailed and efficient scenes with ease.

You can check it out on the asset store through this link:
https://forum.unity3d.com/threads/40-off-sale-released-construction-kit-create-highly-detailed-and-efficient-scenes-in-minutes.473139/#post-3086466

First Person View 2 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

First Person View 2.1 Update

- Added partial support for temporal camera effects (TAA for example)
- Fixed how the Depth Normals Texture is rendered for FPV objects
- Added support for FPV objects to cast correct shadows onto the environment
- Added support for Forward Shaders and Forward Rendering

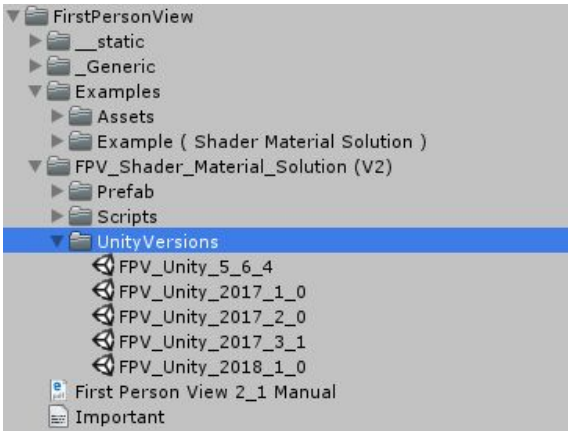
- Removed First Person View Layer requirement. It's no longer needed and should be done on a per project basis.
- Removed FPV V1 solution (Multi-Camera solution), because it does not work with Unity anymore. V2 is a superior version and much more efficient.

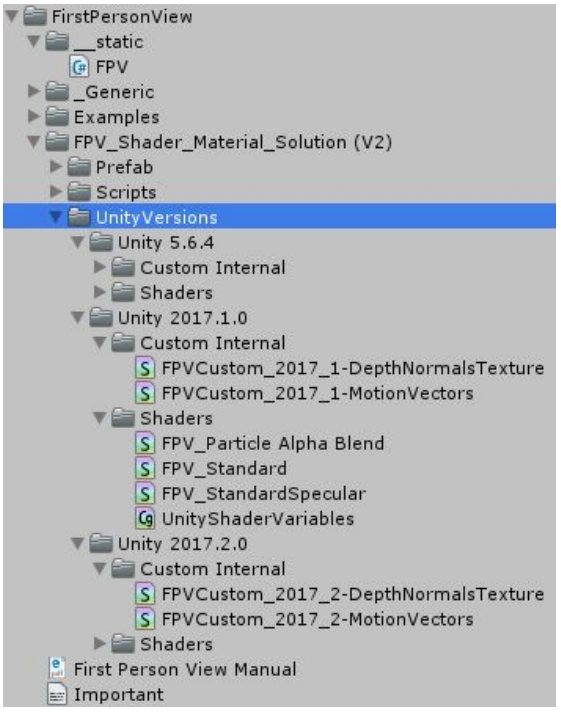
Known Issues

- Shadows/light are still stretched on FPV objects
- Motion Vectors are partially supported. It "resets" the buffer, but in my tests i don't see any issue. Contact me if any issues occur.

First Person View Tutorial

Step 1





Inside the UnityVersions folder, you will find UnityPackages containing shaders/CGIncludes for those specific Unity versions. After importing the contents for the Unity version you're using, you will see files that the asset will use that are specific to those versions of Unity.

- The Custom Internal folder contains the DepthNormalsTexture and MotionVectors files that will be used to override Unity's own files. This will be done through the Graphics Window.
- The shaders folder contains the shaders for that particular unity version. Make sure to always keep your shaders updated with your version of Unity (for example, don't use the standard shader from 5.6 on 2017.2). The appropriate UnityShaderVariables.CGInc is also included in the folder. This file is specific for that version of Unity, so only use the same major version of your Unity version.

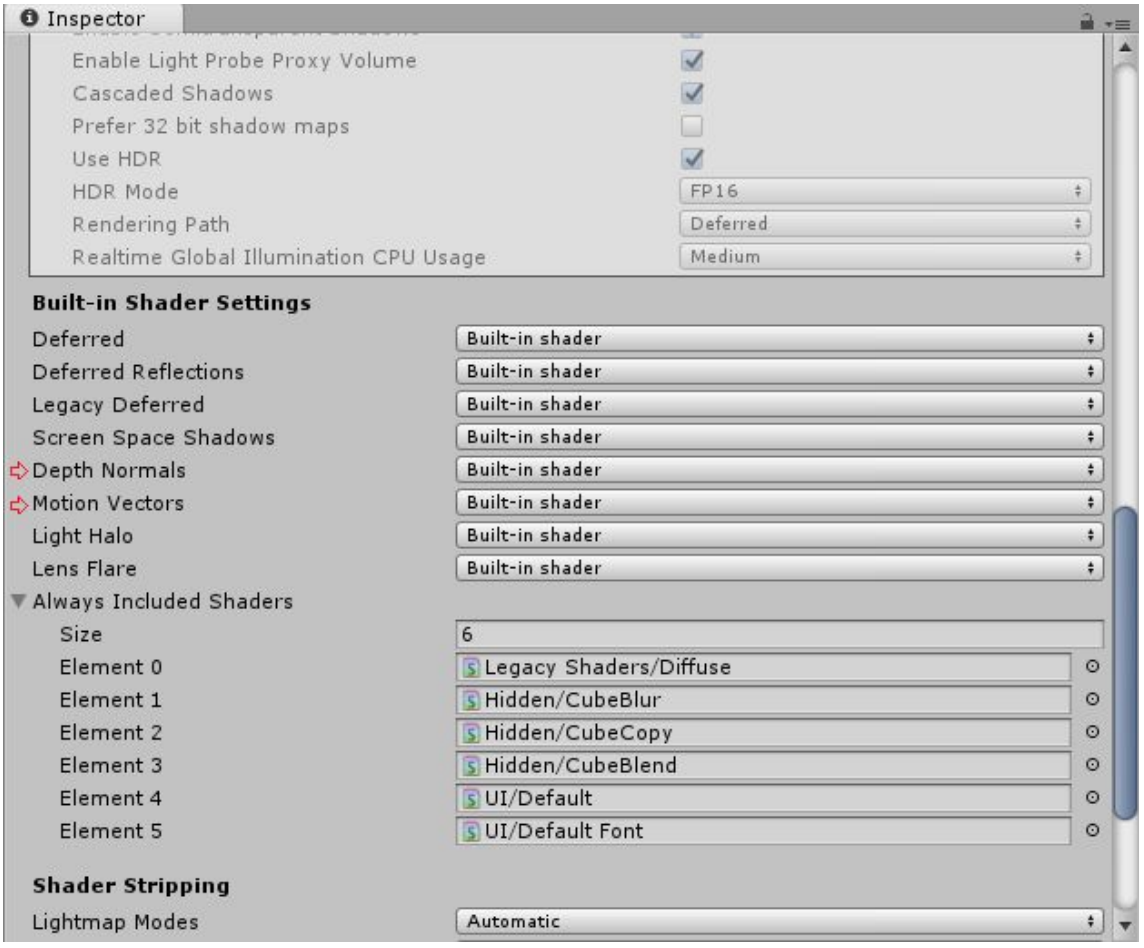
Make sure to use the same files as the Unity version being used.

If you have a different Unity Version, you can easily make the changes yourself by reading the rest of the manual and applying the same changes on your Unity version files.

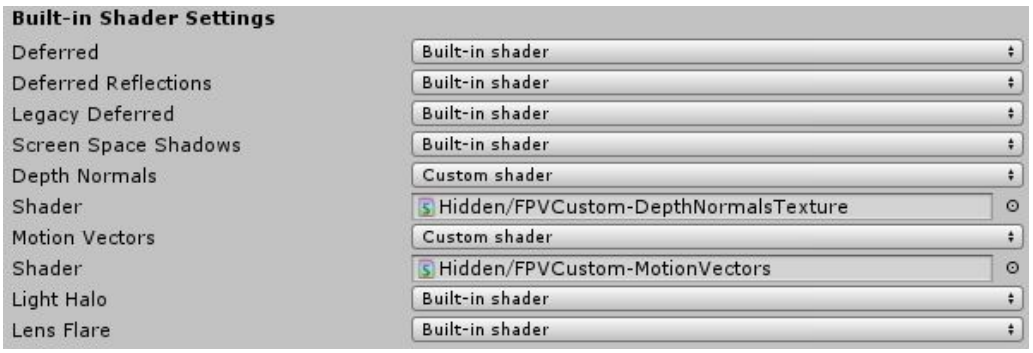
Step 2 (Depth Normals Texture and Motion Vectors)

For this tutorial, i will be using the Unity 2017.1.0 version. The process is the same for the other versions.

In this step, we will be overwriting the Unity's DepthNormalsTexture and Motion Vector shader files. 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:



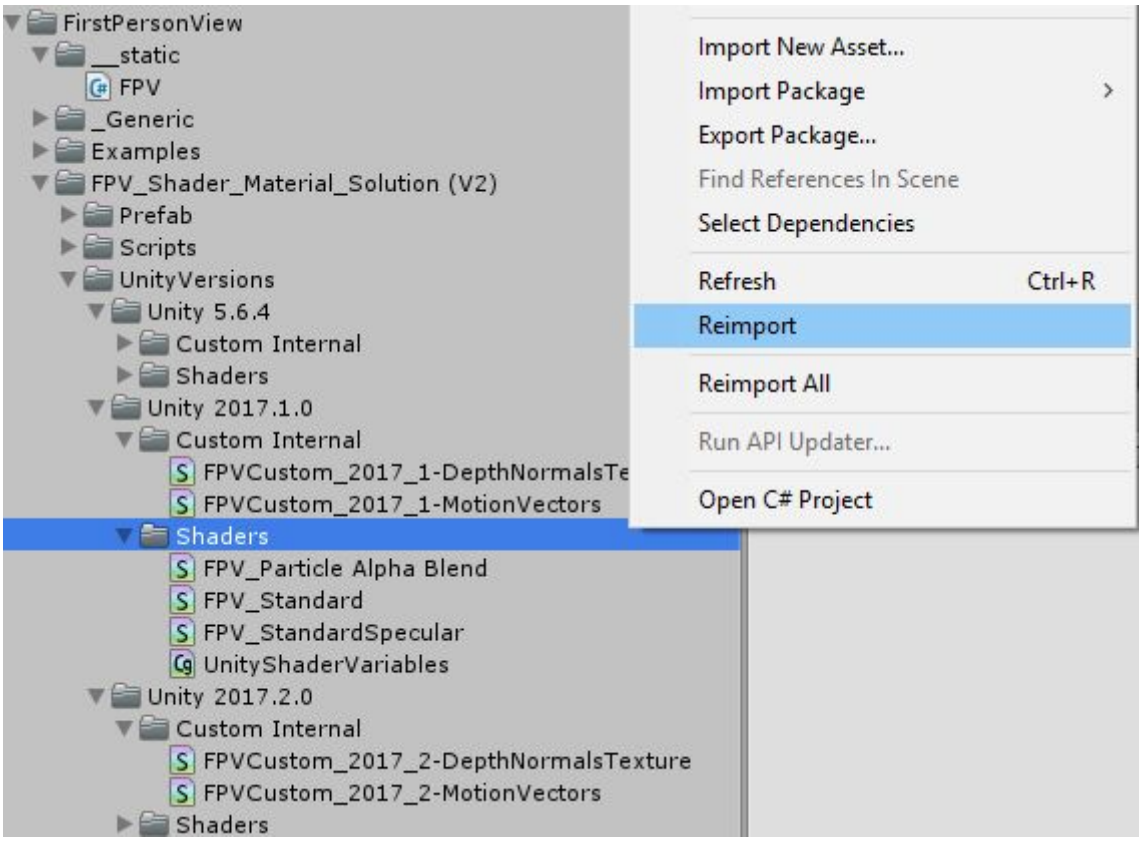
Step 3 (Shader Setup)

Now we are going to be setting up the FPV shaders correctly.

Go to the same major Unity version you are using, and reimport the shaders contained inside the Shaders folder.

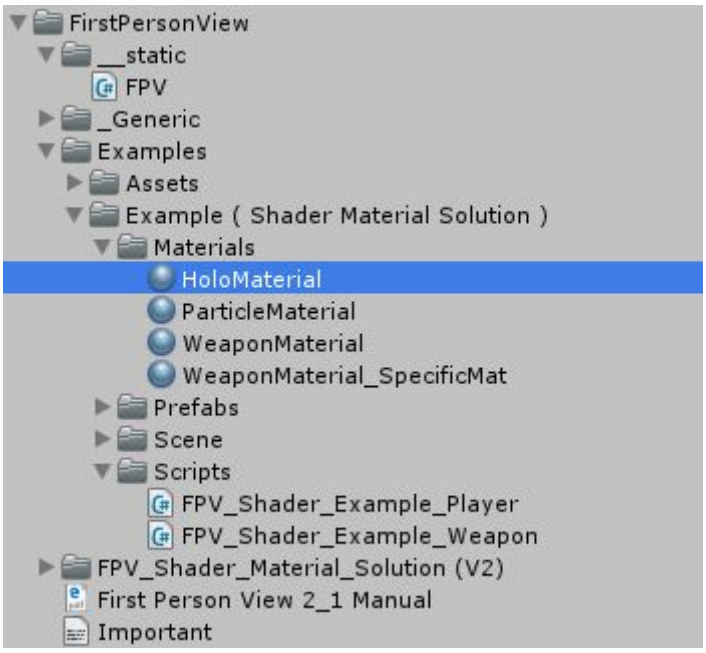
This will force the shader recompile and will use the custom UnityShaderVariables instead of Unity's default one.

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



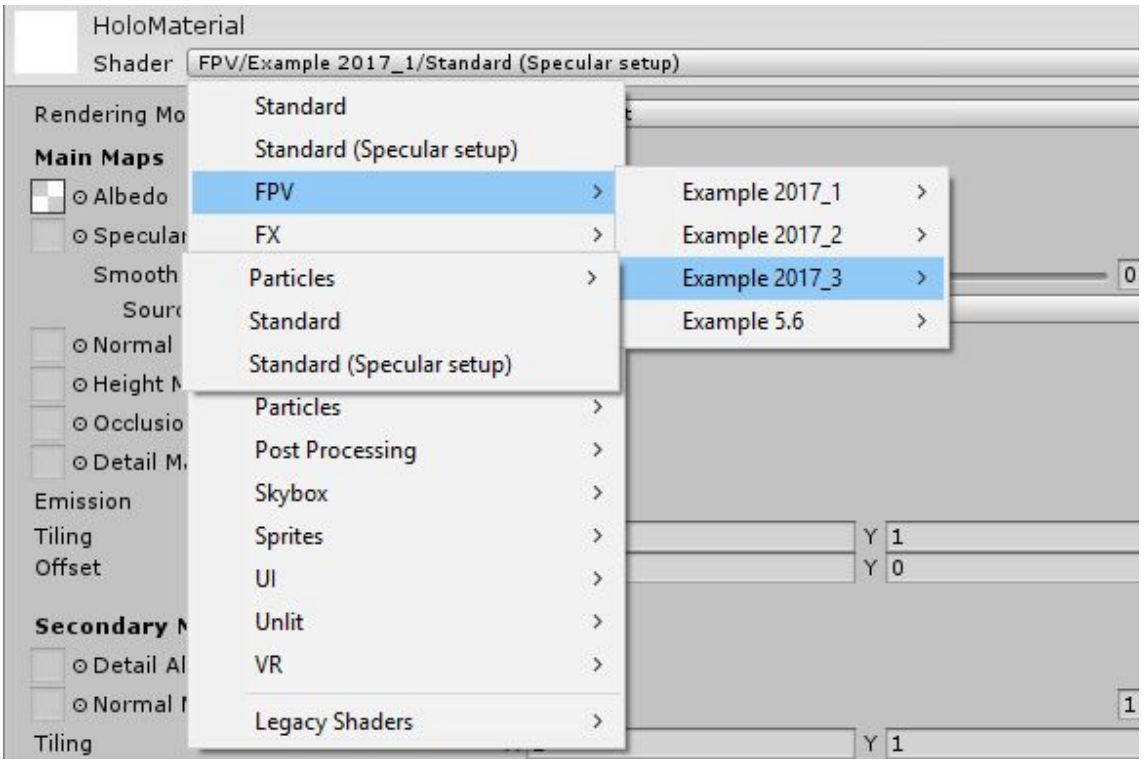
Step 4 (Material Setup)

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



For all materials, 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:

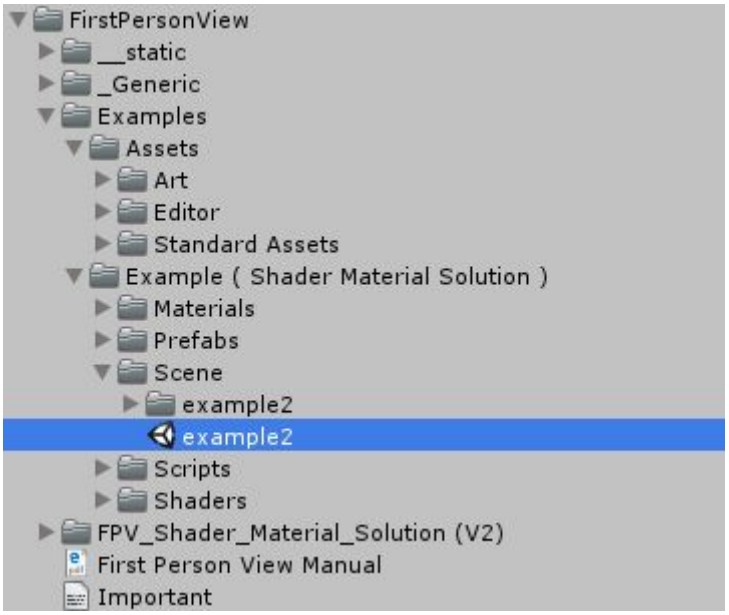


Step 5 (Test Demo)

And that's it for the setup to test the demo provided with FPV2.

Open the demo scene at Examples/Example (Shader Material Solution)/Scene/Example2.

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

UnityShaderVariables

To be able to have a different field of view for FPV objects without having to mess with the actual shader files (and having it work without massively changing shaders), the UnityShaderVariables will contain an extra Float4x4 variable that will be used for FPV Objects.

This variable is called _firstPersonProjectionMatrix and is located on the CBUFFER_START(UnityPerFrame) section of the file:

```
232  // -------------------------------------------------------------------------------
233
234  CBUFFER_START(UnityPerFrame)
235
236      fixed4 glstate_lightmodel_ambient;
237      fixed4 unity_AmbientSky;
238      fixed4 unity_AmbientEquator;
239      fixed4 unity_AmbientGround;
240      fixed4 unity_IndirectSpecColor;
241
242  #if !defined(USING_STEREO_MATRICES)
243  // ------------- FIRST PERSON VIEW 2 -------------
244      #ifdef FIRSTPERSONVIEW
245          float4x4 _firstPersonProjectionMatrix;
246      #endif
247  // -----------------------------------------------
248      float4x4 glstate_matrix_projection;
249      float4x4 unity_MatrixV;
250      float4x4 unity_MatrixInvV;
251      float4x4 unity_MatrixVP;
252      float4 unity_StereoScaleOffset;
253      int unity_StereoEyeIndex;
254  #endif
255
256      fixed4 unity_ShadowColor;
257  CBUFFER_END
258
259
260  // -------------------------------------------------------------------------------
```

It is surrounded by a FIRSTPERSONVIEW keyword so we can enable/disable this when we want, and allows us to have 2 shaders in one, a normal one and a FPV one.

With this prepared, we now want to force the main defined variables that use the default camera's projection matrix to use the FPV matrix instead.

This is done at the beginning of the file, by making a special region for the FIRSTPERSONVIEW Keyword.

```
// ------------- FIRST PERSON VIEW 2 -------------
#ifdef FIRSTPERSONVIEW
#ifdef FPV_LIGHT // Used to enable/disable the projection matrix used during shadows calculations
    #define UNITY_MATRIX_P glstate_matrix_projection // FPV change
#else
    #define UNITY_MATRIX_P _firstPersonProjectionMatrix // FPV change
#endif
    #define UNITY_MATRIX_V unity_MatrixV
    #define UNITY_MATRIX_I_V unity_MatrixInvV
    #define UNITY_MATRIX_VP mul(UNITY_MATRIX_P, UNITY_MATRIX_V) // FPV change
    #define UNITY_MATRIX_M unity_ObjectToWorld

    #define UNITY_MATRIX_MVP mul(UNITY_MATRIX_VP, unity_ObjectToWorld) // FPV change
    #define UNITY_MATRIX_MV mul(unity_MatrixV, unity_ObjectToWorld)
    #define UNITY_MATRIX_T_MV transpose(UNITY_MATRIX_MV)
    #define UNITY_MATRIX_IT_MV transpose(mul(unity_WorldToObject, unity_MatrixInvV))
#else //Default Unity
    #define UNITY_MATRIX_P glstate_matrix_projection
    #define UNITY_MATRIX_V unity_MatrixV
    #define UNITY_MATRIX_I_V unity_MatrixInvV
    #define UNITY_MATRIX_VP unity_MatrixVP
    #define UNITY_MATRIX_M unity_ObjectToWorld

    #define UNITY_MATRIX_MVP mul(unity_MatrixVP, unity_ObjectToWorld)
    #define UNITY_MATRIX_MV mul(unity_MatrixV, unity_ObjectToWorld)
    #define UNITY_MATRIX_T_MV transpose(UNITY_MATRIX_MV)
    #define UNITY_MATRIX_IT_MV transpose(mul(unity_WorldToObject, unity_MatrixInvV))
#endif

#define UNITY_LIGHTMODEL_AMBIENT (glstate_lightmodel_ambient * 2)

// ------------------------------------------------------------------------
```

First, we keep the default Unity variables inside the #else section when the FIRSTPERSONVIEW keyword is not defined.

Inside the section that it is defined, we replace everything else that uses the default projection matrix and replace it with the FPV one.

Since this change also affects how lights calculate their shadow maps on FPV objects, we add another Keyword to the UNITY_MATRIX_P definition, so later on we can add CommandBuffers that will turn on/off this keyword and change what projection matrix it will use.

This fixes the issue with FPV objects not casting correct shadows.

This will be explained in a later section of the manual.

Depth Normals Texture

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

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 change just the sections of the shader that calculate the position of the vertices.

For this shader and for the Motion Vectors shader, we want to use the FPV projection matrix that has the FPV field of view, but we don't want to shorten the Z value like in the _firstPersonProjectionMatrix variable that we use to render the objects, so we use a new global shader variable for this, called _firstPersonProjectionMatrixCustom.

Inside the FPVCustom-DepthNormalsTexture.shader file, we add a new SubShader for the RenderType FPV.

All non-transparent FPV shaders must be of this RenderType (FPV) for this change to be applied. Transparent FPV shaders should not use this RenderType.

This new SubShader is a direct copy of the same SubShader of the RenderType Opaque.

The only changes applied to this new SubShader is to include the global _firstPersonProjectionMatrixCustom variable, and replace the method "UnityObjectToClipPos" with our own using the variable:

```
40  SubShader {
41      Tags { "RenderType"="FPV" }
42      Pass {
43  CGPROGRAM
44
45  #pragma vertex vert
46  #pragma fragment frag
47  #include "UnityCG.cginc"
48  struct v2f {
49      float4 pos : SV_POSITION;
50      float4 nz : TEXCOORD0;
51      UNITY_VERTEX_OUTPUT_STEREO
52  };
53   float4x4 _firstPersonProjectionMatrixCustom;
54  v2f vert( appdata_base v ) {
55      v2f o;
56      UNITY_SETUP_INSTANCE_ID(v);
57      UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(o);
58      o.pos =  mul(mul(_firstPersonProjectionMatrixCustom, UNITY_MATRIX_V), mul(unity_ObjectToWorld, v.vertex));
59      o.nz.xyz = COMPUTE_VIEW_NORMAL;
60      o.nz.w = COMPUTE_DEPTH_01;
61      return o;
62  }
63  fixed4 frag(v2f i) : SV_Target {
64      return EncodeDepthNormal (i.nz.w, i.nz.xyz);
65  }
66  ENDCG
67      }
68  }
```

Motion Vectors (TBA)

Shaders

This whole process was done so 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 Shader

```
        [HideInInspector] _DstBlend ("__dst", Float) = 0.0
        [HideInInspector] _ZWrite ("__zw", Float) = 1.0
    }

    CGINCLUDE
        #pragma multi_compile __ FIRSTPERSONVIEW
        #pragma multi_compile __ FPV_LIGHT
        #define UNITY_SETUP_BRDF_INPUT SpecularSetup
    ENDCG

    SubShader
    {
        Tags { "RenderType"="FPV" "PerformanceChecks"="False" }
        LOD 300

        // ------------------------------------------------------------------
        //  Base forward pass (directional light, emission, lightmaps, ...)
```

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.

Surface Shader

```
Shader "FPV/Example/SurfaceShader" {
    Properties {
        _Color ("Color", Color) = (1,1,1,1)
        _MainTex ("Albedo (RGB)", 2D) = "white" {}
        _Glossiness ("Smoothness", Range(0,1)) = 0.5
        _Metallic ("Metallic", Range(0,1)) = 0.0
    }

    CGINCLUDE
        #pragma multi_compile __ FIRSTPERSONVIEW
        #pragma multi_compile __ FPV_LIGHT
    ENDCG

    SubShader {
        Tags { "RenderType"="FPV" }
        LOD 200

        CGPROGRAM
```

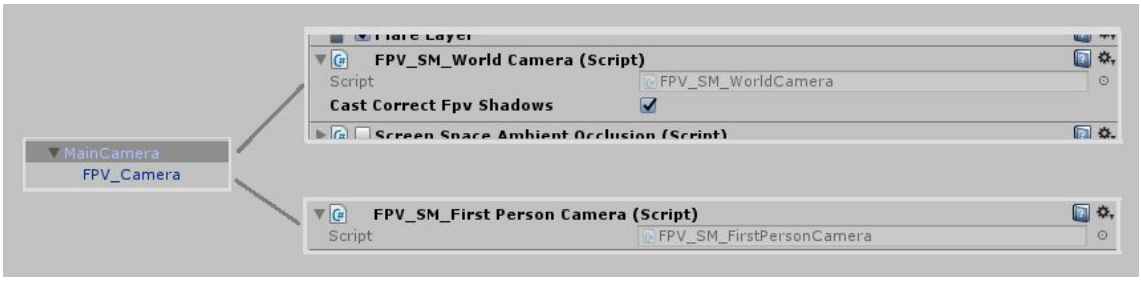
Particle Alpha

```
Shader "FPV/Example 2017_1/Particle Alpha Blended" {
Properties {
    _TintColor ("Tint Color", Color) = (0.5,0.5,0.5,0.5)
    _MainTex ("Particle Texture", 2D) = "white" {}
    _InvFade ("Soft Particles Factor", Range(0.01,3.0)) = 1.0
}

CGINCLUDE
    #pragma multi_compile __ FIRSTPERSONVIEW
    #pragma multi_compile __ FPV_LIGHT
ENDCG

Category {
    Tags { "Queue"="Transparent" "IgnoreProjector"="True" "RenderType"="Transparent" "PreviewType"="Plane" }
    Blend SrcAlpha OneMinusSrcAlpha
    ColorMask RGB
    Cull Off Lighting Off ZWrite Off
```

Camera Setup

First Person View 2 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_SM_WorldCamera, that is responsible for updating the global shader variables, like both projection matrices, 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_SM_FirstPersonCamera 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_SM_WorldCamera

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

```
38      void OnPreCull() {
39          //Before rendering the scene, update the global first person projection on the shaders.
40          Matrix4x4 fpvProjection = FPV.firstPersonCamera.GetProjectionMatrix();
41
42          Matrix4x4 fpvProjectionCustom = fpvProjection;
43
44
45          if (d3d) {
46              fpvProjection.m11 *= -1; // Invert Y value of the MVP vertex position
47              fpvProjectionCustom.m11 *= -1; // Invert Y value of the MVP vertex position
48          }
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              fpvProjectionCustom.m22 *= 0.3f;
55          }
56          else { //Z-Buffer not inverted
57              fpvProjection.m22 *= 0.7f; // Shorten the Z value of the MVP vertex position
58              fpvProjectionCustom.m22 *= 0.7f;
59          }
60
61          //Update the global shader variable
62          Shader.SetGlobalMatrix("_firstPersonProjectionMatrix", fpvProjection);
63          Shader.SetGlobalMatrix("_firstPersonProjectionMatrixCustom", fpvProjectionCustom);
64      }
```

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

```
66      private void PrepareLightingCommandBuffer() {
67          _commandBufferBefore = new CommandBuffer();
68          _commandBufferAfter = new CommandBuffer();
69
70          _commandBufferBefore.EnableShaderKeyword("FPV_LIGHT");
71          _commandBufferAfter.DisableShaderKeyword("FPV_LIGHT");
72
73          _camera.AddCommandBuffer(CameraEvent.BeforeLighting, _commandBufferBefore);
74          _camera.AddCommandBuffer(CameraEvent.AfterLighting, _commandBufferAfter);
75
76      }
```

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.

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_SM_Object

The simplest way to prepare the object will be to attach the FPV_SM_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_SM_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_SM_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_SM_Renderer if the renderer has just 1 material, or the FPV_SM_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_SM_Renderer

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

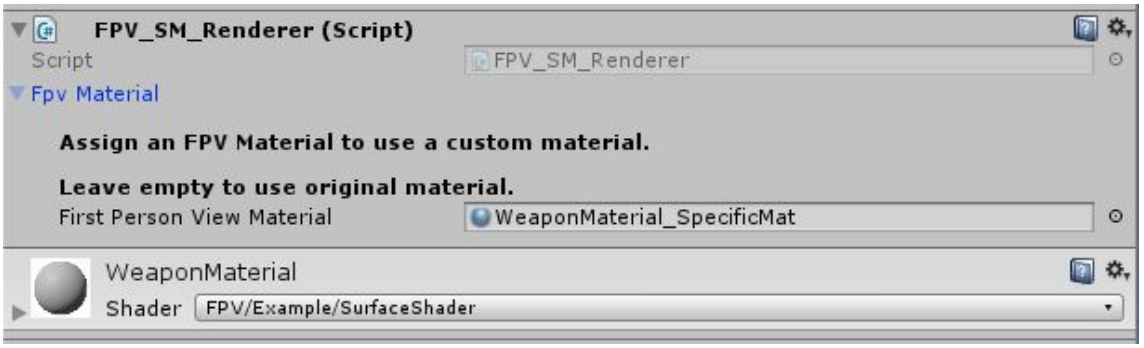
By using a FPV shader, the shader can be in two modes: Normal or FIRSTPERSONVIEW.

When initializing this component, if the variable “fpvMateria”l 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.

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



FPV_SM_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.

Static FPV Class

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

```
1   using UnityEngine;
2
3   namespace FirstPersonView
4   {
5       /// <summary>
6       /// Static Class for the First Person View.
7       /// </summary>
8       public static class FPV
9       {
10          /// <summary>
11          /// Reference to the Main Camera that renders the environment
12          /// </summary>
13          public static IFPV_Camera mainCamera { get; set; }
14
15          /// <summary>
16          /// Reference to the First Person Camera.
17          /// </summary>
18          public static IFPV_Camera firstPersonCamera { get; set; }
```

Three important methods are provided in this static class:

```
26          /// <summary>
27          /// Convert a First Person View point to World View point.
28          /// </summary>
29          /// <param name="point"></param>
30          /// <returns></returns>
31          public static Vector3 FPVPointToWorldPoint(Vector3 point)
32          {
33              point = firstPersonCamera.GetCamera().WorldToScreenPoint(point);
34              return mainCamera.GetCamera().ScreenToWorldPoint(point);
35          }
```

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.

```
37          /// <summary>
38          /// Transform a point and a direction from First Person View to World View
39          /// </summary>
40          /// <param name="point"></param>
41          /// <param name="direction"></param>
42          /// <param name="resPoint"></param>
43          /// <param name="resDirection"></param>
44          public static void FPVToWorld(Vector3 point, Vector3 direction, out Vector3 resPoint, out Vector3 resDirection)
45          {
46              resPoint = FPVPointToWorldPoint(point);
47
48              Vector3 pointForward = point + direction;
49              pointForward = FPVPointToWorldPoint(pointForward);
50              resDirection = pointForward - resPoint;
51          }
```

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.

```
53          /// <summary>
54          /// Transform a point and a direction based on a Transform from First Person View to World View
55          /// </summary>
56          /// <param name="trans"></param>
57          /// <param name="resPoint"></param>
58          /// <param name="resDirection"></param>
59          public static void FPVToWorld(Transform trans, out Vector3 resPoint, out Vector3 resDirection)
60          {
61              FPVToWorld(trans.position, trans.forward, out resPoint, out resDirection);
62          }
63
```

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 example, let's take a look at the ExampleWeapon prefab.

Example Weapon

The main gameobject contains the FPV_SM_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_SM_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.

```
32      public void SetWeaponOnPlayer(Transform player)
33      {
34          transform.SetParent(player);
35          transform.localPosition = Vector3.zero;
36          transform.localRotation = Quaternion.identity;
37
38          wRigidbody.isKinematic = true;
39          wCollider.enabled = false;
40
41          isOnPlayer = true;
42
43          fpvObject.SetAsFirstPersonObject();
44      }
```

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

```
46      public void SetWeaponOnWorld()
47      {
48          transform.parent = null;
49
50          wRigidbody.isKinematic = false;
51          wCollider.enabled = true;
52
53          isOnPlayer = false;
54
55          fpvObject.RemoveAsFirstPersonObject();
56      }
```

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.

```
76      private void CreateWeapon()
77      {
78          if (weapon != null) return;
79
80          GameObject newWeapon = GameObject.Instantiate(weaponPrefab);
81
82          weapon = newWeapon.GetComponent<FPV_Shader_Example_Weapon>();
83          weapon.Setup();
84          SetWeaponOnPlayer();
85      }
```

```
110     private void SetWeaponOnPlayer()
111     {
112         weapon.SetWeaponOnPlayer(weaponPlacement);
113     }
```

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

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

```
 98      private void SwitchWeapon()
 99      {
100          if (weapon.IsOnPlayer())
101          {
102              SetWeaponOnWorld();
103          }
104          else
105          {
106              SetWeaponOnPlayer();
107          }
108      }
109
110      private void SetWeaponOnPlayer()
111      {
112          weapon.SetWeaponOnPlayer(weaponPlacement);
113      }
114
115      private void SetWeaponOnWorld()
116      {
117          weapon.SetWeaponOnWorld();
118      }
```

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.

```
36          private void Aim()
37          {
38              if (weapon == null) return;
39
40              bool aiming = Input.GetMouseButton(1);
41
42              weapon.Aim(aiming, fpvCamera.transform);
43
44              float futureFoV = aiming ? 20 : 70;
45
46              fpvCamera.fieldOfView = Mathf.Lerp(fpvCamera.fieldOfView, futureFoV, Time.deltaTime * 10);
47          }
48
49          private void ChangeFOV()
50          {
51              //World Camera FOV
52              float fOVChange = 0;
53              if (Input.GetKey(KeyCode.Comma))
54              {
55                  fOVChange = -Time.deltaTime * 10;
56              }
57              else if (Input.GetKey(KeyCode.Period))
58              {
59                  fOVChange = Time.deltaTime * 10;
60              }
61              worldCamera.fieldOfView = Mathf.Clamp(worldCamera.fieldOfView + fOVChange, 50, 120);
62
63              //First Person Camera FOV
64              fOVChange = 0;
65              if (Input.GetKey(KeyCode.N))
66              {
67                  fOVChange = -Time.deltaTime * 10;
68              }
69              else if (Input.GetKey(KeyCode.M))
70              {
71                  fOVChange = Time.deltaTime * 10;
72              }
73              fpvCamera.fieldOfView = Mathf.Clamp(fpvCamera.fieldOfView + fOVChange, 4, 70);
74          }
```

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.

```
66          private void PrepareLightingCommandBuffer() {
67              _commandBufferBefore = new CommandBuffer();
68              _commandBufferAfter = new CommandBuffer();
69
70              _commandBufferBefore.EnableShaderKeyword("FPV_LIGHT");
71              _commandBufferAfter.DisableShaderKeyword("FPV_LIGHT");
72
73              _camera.AddCommandBuffer(CameraEvent.BeforeLighting, _commandBufferBefore);
74              _camera.AddCommandBuffer(CameraEvent.AfterLighting, _commandBufferAfter);
75
76          }
```

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_SM_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_SM_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.

```
4   public class FPV_SM_LightCommandBuffer : MonoBehaviour
5   {
6       [Header("Only use this script when using Forward Rendering")]
7       [SerializeField]
8       private Light _light;
9
10      private CommandBuffer _commandBufferBefore;
11      private CommandBuffer _commandBufferAfter;
12
13      void Start() {
14          _commandBufferBefore = new CommandBuffer();
15          _commandBufferAfter = new CommandBuffer();
16
17          _commandBufferBefore.EnableShaderKeyword("FPV_LIGHT");
18          _commandBufferAfter.DisableShaderKeyword("FPV_LIGHT");
19
20          _light.AddCommandBuffer(LightEvent.BeforeShadowMapPass, _commandBufferBefore);
21          _light.AddCommandBuffer(LightEvent.AfterShadowMapPass, _commandBufferAfter);
22      }
23  }
```

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