

ULTIMATE FPS CAMERA



v1.4.0 Documentation

© VisionPunk, Minea Softworks. All rights reserved.

Table of contents

[Preface to v1.4](#)

[Introduction](#)

[Playing the demo](#)

[Demo controls](#)

[Getting started](#)

[Trying out the demo scenes](#)

[CleanScene](#)

[DemoScene1](#)

[DemoScene2](#)

[DemoScene3](#)

[Using the example player prefabs in your own scene](#)

[Important Concepts](#)

[FP-scripts](#)

[States](#)

[Player Events](#)

[What's with the "vp_" filename prefix?](#)

[Animation concepts](#)

[Springs](#)

[Bob](#)

[Procedural noise](#)

[Working in the editor](#)

[Adjusting values](#)

[Navigating for text assets](#)

[Keep an eye on the Console](#)

[FPController](#)

[Motor](#)

[Acceleration](#)

[Damping](#)

[Jump force](#)

[Air Speed](#)

[Slope Speed Up](#)

[Slope Speed Down](#)

[Physics](#)

[Force Damping](#)

- [Push Force](#)
 - [Gravity Modifier](#)
 - [Wall Bounce](#)
- [FPCamera](#)
 - [Mouse](#)
 - [Sensitivity](#)
 - [Smooth Steps](#)
 - [Smooth Weight](#)
 - [Acceleration](#)
 - [Acceleration Threshold](#)
 - [Rendering](#)
 - [Field of View](#)
 - [Zoom Damping](#)
 - [Position](#)
 - [Offset](#)
 - [Ground Limit](#)
 - [Spring Stiffness](#)
 - [Spring Damping](#)
 - [Spring2 Stiffness](#)
 - [Spring2 Damping](#)
 - [Kneeling](#)
 - [Rotation](#)
 - [Pitch Limit](#)
 - [Yaw Limit](#)
 - [Kneeling](#)
 - [Strafe Roll](#)
 - [Shake](#)
 - [Speed](#)
 - [Amplitude](#)
 - [Bob](#)
 - [Rate](#)
 - [Amplitude](#)
 - [Step Threshold](#)
 - [Input Velocity Scale](#)
 - [Max Input Velocity](#)
- [FPWeapon](#)
 - [Rendering](#)
 - [Field of View](#)
 - [Zoom Damping](#)
 - [Clipping Planes](#)
 - [Position](#)
 - [Offset](#)
 - [Exit Offset](#)
 - [Pivot](#)
 - [Pivot Stiffness](#)
 - [Pivot Damping](#)

[Show Pivot](#)
[Spring Stiffness](#)
[Spring Damping](#)
[Spring2 Stiffness](#)
[Spring2 Damping](#)
[Kneeling](#)
[Fall Retract](#)
[Walk Sliding](#)
[Input Velocity Scale](#)
[Max Input Velocity](#)

[Rotation](#)

[Offset](#)
[Exit Offset](#)
[Spring Stiffness](#)
[Spring Damping](#)
[Spring2 Stiffness](#)
[Spring2 Damping](#)
[Look Sway](#)
[Strafe Sway](#)
[Fall Sway](#)
[Slope Sway](#)
[Input Rotation Scale](#)
[Max Input Rotation](#)

[Shake](#)

[Speed](#)
[Amplitude](#)

[Bob](#)

[Rate](#)
[Amplitude](#)
[Max Input Velocity](#)

[FPSShooter](#)

[Projectile](#)

[Firing Rate](#)
[Prefab](#)
[Scale](#)
[Count](#)
[Spread](#)

[Motion](#)

[Position Recoil](#)
[Rotation Recoil](#)
[Position Reset](#)
[Rotation Reset](#)
[Position Pause](#)
[Rotation Pause](#)
[Dry Fire Recoil](#)

[Muzzle Flash](#)

- [Prefab](#)
 - [Position](#)
 - [Scale](#)
 - [Fade Speed](#)
 - [Show Muzzle Flash](#)
 - [Shell](#)
 - [Prefab](#)
 - [Scale](#)
 - [Eject Position](#)
 - [Eject Direction](#)
 - [Eject Velocity](#)
 - [Eject Spin](#)
 - [Eject Delay](#)
 - [Ammo](#)
 - [Max Count](#)
 - [Reload Time](#)
 - [Sound](#)
 - [Fire](#)
 - [Dry Fire](#)
 - [Reload](#)
 - [Fire Pitch](#)
- [Presets](#)
 - [Load & Save](#)
 - [Save Tweaks](#)
- [States](#)
 - [Creating a new state](#)
 - [Activating and deactivating states](#)
 - [State order](#)
 - [The Default state](#)
 - [Deleting a state](#)
 - [Removing a preset](#)
 - [Persist play mode changes](#)
- [Bullet](#)
 - [Creating a Bullet prefab from scratch](#)
 - [Parameters](#)
 - [Range](#)
 - [Force](#)
 - [Damage](#)
 - [Damage Method Name](#)
 - [SparkFactor](#)
 - [Impact Sounds](#)
 - [Sound Impact Pitch](#)
 - [Particle FX Prefabs](#)
 - [ImpactPrefab](#)
 - [DustPrefab](#)
 - [SparkPrefab](#)

DebrisPrefab

MuzzleFlash

*

Creating a MuzzleFlash prefab from scratch

Shell

Creating a Shell prefab from scratch

Parameters

Life Time

Persistence

Bounce Sounds

Damage Handler

Parameters

Health

DeathEffect

MinDeathDelay & MaxDeathDelay

Respawns

MinRespawnTime & MaxRespawnTime

RespawnCheckRadius

RespawnSound

Explosion

Spawning an Explosion from script

Parameters

Radius

Force

UpForce

Damage

CameraShake

DamageMethodName

SoundMinPitch & SoundMaxPitch

FXPrefabs

Image Effects

Important notes

Rendering Path considerations

Forward Rendering

Deferred Lighting

Use Player Settings

Compatibility notes

Crease

DepthOfField

EdgeDetectEffectNormals

GlobalFog

SSAOEffect

SunShafts

Manipulating the FPS components from script

Creating a new script

Moving the Controller

[Applying forces](#)

[Teleporting the player](#)

[A word on inventory placeholder features](#)

[Loading presets from script](#)

[Advanced: Event system](#)

[Introduction to the event system](#)

[Modularity](#)

[Performance](#)

[Event handler tutorial](#)

[Preparation](#)

[1. Declaring an event](#)

[2. Making the PlayerEventHandler available to the scripts](#)

[3. Registering a target script with the event handler](#)

[4. Adding an event listener](#)

[5. Sending an event](#)

[Event types](#)

[vp_Message](#)

[vp_Attempt](#)

[vp_Value](#)

[vp_Activity](#)

[Activity additional properties](#)

[Binding states to Activities](#)

[Activity example flows](#)

[What's the difference between an Activity and a State?](#)

[Support and additional information](#)

Preface to v1.4

Ultimate FPS Camera v1.4, which has been a long time in the making, is a more or less complete rewrite of what began as a simple prototype FPS camera system. It takes into account months of user feedback, feature requests and many, many reported bugs.

I want to direct a huge **THANK YOU** to all the awesome users that have provided fantastic feedback, testing and ideas. Ultimate FPS Camera has come a long, long way since the first prototype release (which didn't even have support for firing the weapon) and is now a quite extensive system with tons of features.

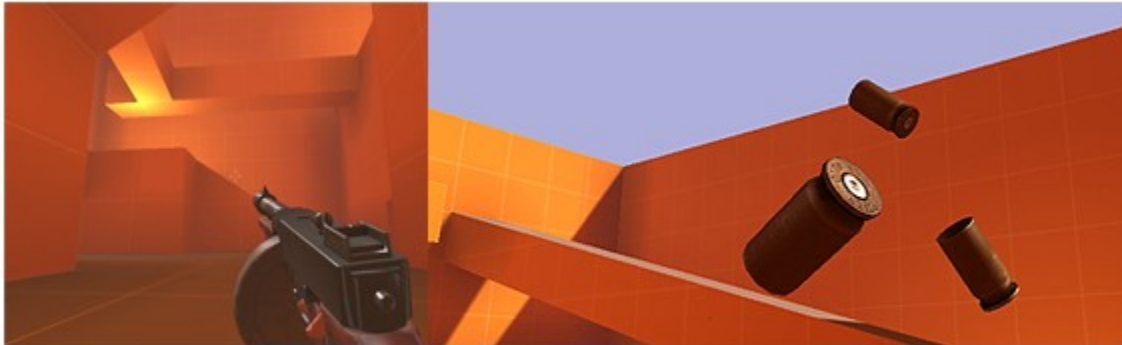
It couldn't have been done without you and I hope you'll have tons of fun with it!

Thanks

/Cal
@VisionPunk

Introduction

Congratulations on your purchase of Ultimate FPS Camera! A whole new world of amazing camera and weapon animation is now at your fingertips. Over 1800 hours of work has gone into the design and programming of these systems and I sincerely hope you'll find them useful!



The basic concept of Ultimate FPS Camera is feeding player movements (mouse input, walking, jumping) into the camera and weapon transforms using realtime spring physics, sinus bob and procedural noise shaking. The system uses hundreds of parameters to manipulate the camera and weapon model, allowing for a vast range of complex, realtime-generated behaviors.

Ultimate FPS Camera is not designed to fully replace regular animations, but combining it with traditional animation can result in super-lifelike motion rivaling the best AAA games out there! Imagine having an artillery shell detonating nearby, shaking your arms and the camera violently while you desperately attempt to reload a shotgun. Shell shocked, your vision, hands and movement pattern are disturbed for a while before returning to normal ...

In short, this system is specialized in handling camera and weapon motion resulting from player movement and firing, such as weapon swaying, recoil, fall impact and explosion knockback. It can be used as a cornerstone for a new FPS, but great effort has also been put into keeping it modular, so you should be able to pick it apart and use bits and pieces of it in your existing systems.

Hope you'll have as fun working and playing with this system as I've had building it! Feel free to show off your FPS work or participate in the official forum discussion at:

<http://forum.unity3d.com/threads/126886-Ultimate-FPS-Camera-RELEASED?goto=newpost>

/Cal
@VisionPunk

Playing the demo

The walkthrough demo is fairly self-explanatory. Use the big arrows at the top of the screen to navigate back and forth between screens. Press the buttons to try out various settings.



Demo controls

WASD

C

Space

Shift

R

Middle & Right Mouse Button

F

G

ESC

Enter

Move

Crouch

Jump

Hold to Run (in DemoScene3)

Reload (in DemoScene3)

Hold to Zoom in

Toggle Fullscreen

Toggle GUI

Quit app (if offline standalone player)

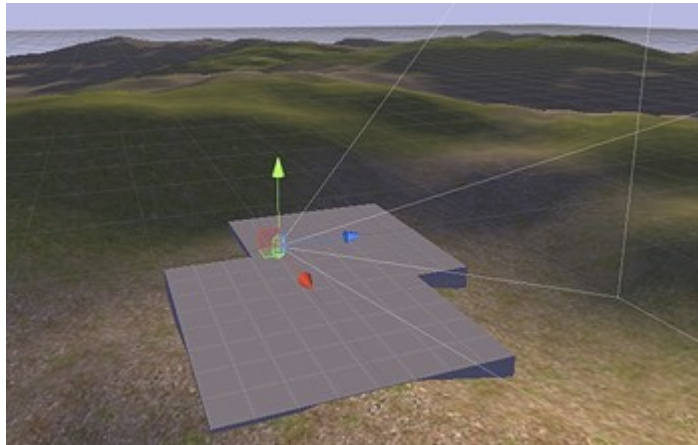
Toggle menu

Getting started

Trying out the demo scenes

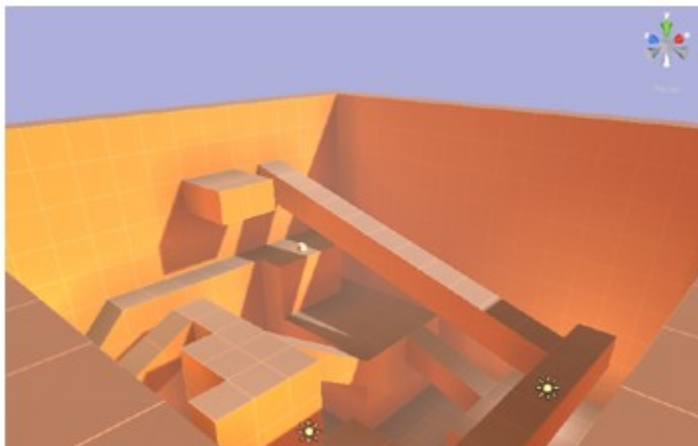
CleanScene

This scene (located in the "Demo/Levels" folder) is a simple terrain with a basic first person camera and controller, intended for prototyping and experimentation. To try some alternative FPS players, disable the *Camera&Controller* gameobject and drag the *SimplePlayer* or *AdvancedPlayer* from the "Content/Prefabs/Players" folder into the scene.



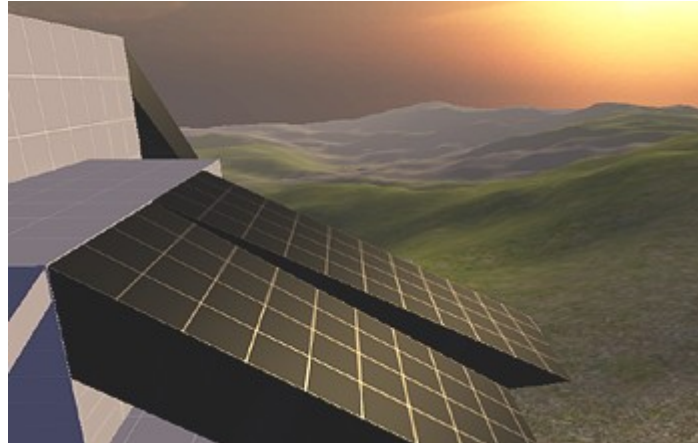
DemoScene1

This scene (located in the "Demo/Levels" folder) is designed as a test range using a Half Life "orange map" style. Its content and FPS player is tightly integrated into a walkthrough of the most important features.



DemoScene2

This scene is as test range for physics testing. It has an extra "slidy" player and several ramps tilted at various degrees for different test cases. To shut off the demo walkthrough, disable or delete the "Demo" gameobject in the scene Hierarchy. Upon pressing Play you can now run around freely.



DemoScene3

This scene is an outdoor environment with a platform and a bunch of explosive cubes and pickups of various kinds. It is a demonstration of how to put some of the example gameplay scripts in use. To shut off the demo walkthrough, disable or delete the "Demo" gameobject in the scene Hierarchy. Upon pressing Play you will be able to explore freely with an advanced FPS player.



Using the example player prefabs in your own scene

1. Make sure your scene has some kind of floor in it, for example a huge box with collision. An easy way of achieving this is going to the Unity main menu, choosing "GameObject -> Create Other -> Cube" and setting the position of the cube transform to "0, 0, 0" and its scale to "1000, 0, 1000".
2. In the Project view, browse to the "Content/Prefabs/Players" folder and drag one of the example player prefabs into your scene hierarchy. Make sure it is positioned slightly above the floor.

NOTE: If you have a fresh scene with an auto created "Main Camera" object in it, delete this object (or atleast its Audio Listener) or you will get Unity warnings in the console.

Important Concepts

If you're like me and skip through manuals, here's what you *really* need to know: Ultimate FPS Camera is all about **FP-scripts**, **Component States** and **Player Events**.

FP-scripts

An FP script is a *Unity component* designed to be part of a local, First Person player. These all have names beginning with "vp_FP" and include the *Player*, the *Controller*, the *Camera*, its *Weapons* and their *Shooters*.

States

A state is a *range of settings* for a specific component during a specific activity. For example: the camera "Crouch" state has a value in it defining a lower Y-position for the camera. States are saved in small scripts called *Presets*.

Player Events

Player events are how FP-scripts *communicate*. For example: when the character controller detects a ground impact from a long fall, it sends the force of impact to the *Player Event Handler*, which in turn broadcasts it to the camera and weapon components (making them shake violently).

***TIP:** If you are planning to do more than just a little scripting with this system, it is a good idea to first have a glance at the Event System chapter of this manual, and everything might become easier for you.*

What's with the "vp_" filename prefix?

The "vp_" prefix serves to identify all scripts in the package as belonging to the same ecosystem. The main purpose of this is preventing filename collision with scripts from other asset publishers or Unity core classes.

Animation concepts

Springs

The main workhorse of Ultimate FPS Camera is a simple spring class that operates on transform position, rotation or scale. The spring has a **Rest State** (also known as the "static equilibrium") where it "wants to be". If you move it away from the target value using external force, it will immediately try to go back to its target position or angle. The spring **Stiffness** - or mechanical strength - determines how loosely or rigidly the spring behaves. Finally, the **Damping** makes spring velocity wear off as it approaches its rest state. Springs can be used to simulate everything from jello (a loose spring operating on object scale), an arrow hitting a wall (a very stiff spring rotating the arrow with the pivot at the head) or an underwater camera (a loose spring making the camera bounce softly as your feet hit the ocean floor).

Bob

Bob is the sinusoidal motion of an object or the camera. View bob has been around in first person shooters for ages as a means of swinging the camera up and down when walking. Modern shooters don't have a very pronounced view bob, but bob is still very useful for animating the weapon model (and camera if used in moderation). If applying bob along both the x and y vector you can get interesting results such as the feeling of being a dinosaur or a huge ogre.

Procedural noise

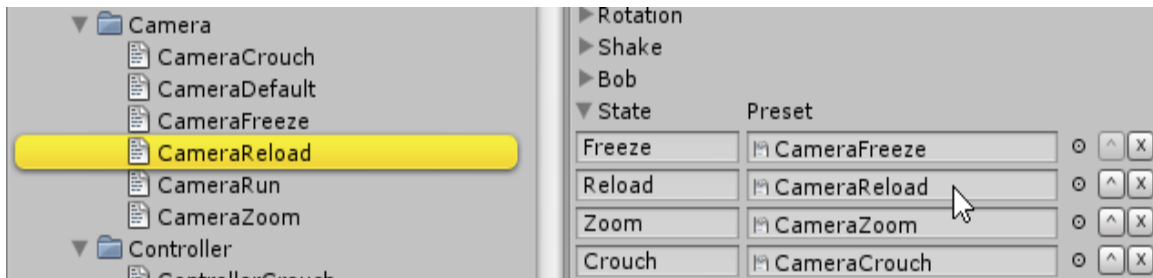
Perlin noise has been used in computer graphics and special effects for years as a means of generating smoke, terrains and camera shakes (to name a few). Ultimate FPS Camera uses a standard perlin noise function for applying random turbulence to the weapon and camera. For example: a slight breathing camera motion as you zoom in with a sniper rifle, gentle idle movements for a hand holding a sub machine gun, or the heavily disturbed camera movements of a character that is drunk or poisoned.

Working in the editor

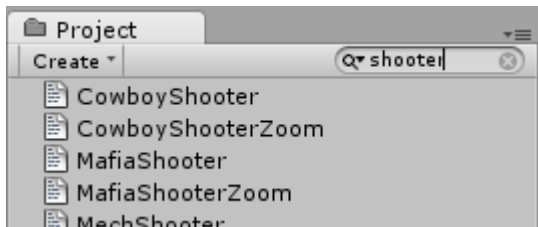
Adjusting values

Sliders are used to modify most variables. Text edit fields are used for variables with no limited range. Note that you can click and drag the text in front of a text edit field (i.e. the "X") to adjust the value with the mouse. Pressing and holding "Alt" while doing this gives you finer control. This is very useful when adjusting particular values, for example weapon position offset

Navigating for text assets



The Unity Editor has a neat highlight feature which simplifies navigating for assets. To locate a text asset you see in a component State list, simply click on it and the text asset will be highlighted in the Project view. This gives you quick access to close by text presets compatible with the current component. You can also click directly on a text file in the Project view to inspect its values.



Another great way to quickly locate text assets is to write part of their names in the Project view search box (one of the best Unity features imho).

Keep an eye on the Console

Ultimate FPS Camera will try and communicate any errors or problems it encounters via the Unity console. It's always a good idea to be aware of any red (Error) or yellow (Warning) messages it displays. Click on the icon to see the full list of errors. Sometimes the latest error is not the culprit and you need to look further up the list.



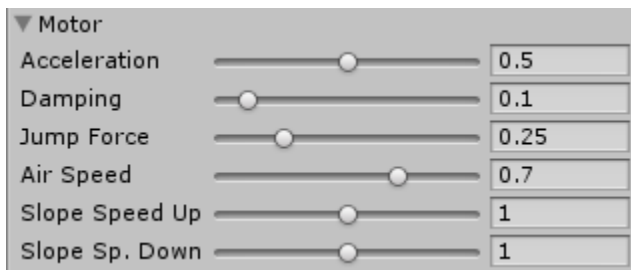
FPController

vp_FPController is an extended character controller designed to work perfectly with the vp_FPCamera class. It features acceleration, sliding, air control, wall bouncing, external forces and the ability to push around rigidbodies. Move it from script by calling its "MoveLeft", "MoveRight", "MoveForward" or "MoveBack" methods every Update.

The controller cannot rotate by itself. To rotate it via mouselook, add an FPCamera to its hierarchy as described in the above chapter "Getting Started -> Creating an FPPlayer from scratch".

Motor

The Motor foldout contains all basic movement parameters of the FPController.



Acceleration

The amount of world velocity added to the controller every frame when moving.

Damping

How quickly velocity should deteriorate when movement stops. Low values will get you a "slippery floor" or icy feeling. High values will slow down the controller.

TIP: If you're not a big fan of "slidy" controls, set both Acceleration and Damping to 1 and reduce damping until you're happy with the walking speed. This will get you a controller that stops abruptly.

TIP: If you want to tweak the controller for a specific real world max speed, you may output the player's speed in meters per second using this line of code:

```
Debug.Log (Controller.Velocity.magnitude);
```

Jump force

The amount of up-force that is added to the character controller in a single frame when the player jumps. The height of the jump is slightly influenced by the gravity modifier.

Air Speed

A factor determining how fast the controller can move when it has no contact with the ground (jumping, falling or flying). A value of 0 means the player can not move sideways at all when flying (like a real world human). A value of 1 means the player can move as quickly in the air as on the ground. This is often used in games to increase the feeling of control, especially if platform jumping is a gameplay element.

Slope Speed Up

This value increases or decreases controller velocity on upward slopes. At 1.0, velocity on upward slopes will be kept roughly the same as on flat ground. Values lower or higher than 1 will make the controller slow down / speed up on upward slopes, respectively.

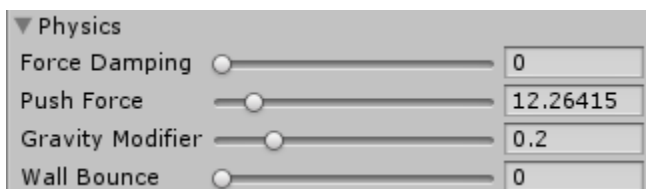
***TIP:** The slope speed feature can be especially useful on outdoor terrain; try setting "Slope Speed Up" to 0.5 and "Slope Speed Down" to 1.5. This will slow down the controller moving uphill, and give it a speed boost running downhill.*

Slope Speed Down

This value increases or decreases controller velocity on downward slopes. At 1.0, velocity on downward slopes will be kept roughly the same as on flat ground. Values lower or higher than 1 will make the controller slow down / speed up on downward slopes, respectively.

Physics

The Physics foldout contains parameters related to interaction with the game world.



Force Damping

This value scales how receptive the controller is to scripted external forces such as explosion knockback. A value of 1 means it is not affected at all, like an extremely heavy object. A value of 0 means the controller will be affected enormously, as would a very light object.

Push Force

This value determines how "strong" the controller is when pushing around physical objects (rigidbodies). Rigidbodies with larger mass will be harder to move.

Gravity Modifier

The gravity modifier regulates how much the controller will be affected by gravity. Higher values give a heavier feel. The "Physics.Gravity" value of the Unity Physics Manager affects the object aswell.

Wall Bounce

This experimental feature makes the character controller bounce when it runs into walls, giving a slightly more organic feel on wall collision.

FPCamera

The `vp_FPCamera` class governs all behaviour of the first person camera which is a child to the main `FPPlayer` gameobject. It features mouse smoothing, view bob, camera shakes, smooth zooming, kneeling and reacting to external forces among many other things.

NOTE: The `vp_FPCamera` class is designed to be fairly independent from the `vp_FPController` class. It should work with any FPS walker based on Unity's standard `CharacterController`.

Mouse



Sensitivity

Sets mouse sensitivity in the X (yaw) and Y (pitch) axes.

Smooth Steps

Mouse smoothing interpolates mouse input over several frames to reduce jerky player input. "Smooth Steps" determines how many of the most recent frames of mouse input to average. A typical value is 10.

Smooth Weight

"Smooth Weight" determines the influence of recent mouse input as it ages. Reducing the weight modifier to 0 will provide the user with raw, unfiltered feedback. Increasing it to 1.0 will cause the result to be a simple average of all the recently sampled smooth steps (and will feel very laggy). A typical value is 0.5.

Acceleration

Mouse acceleration increases the mouse sensitivity above certain speeds. Along with low mouse sensitivity this allows high precision without loss of turn speed. It gives you optimal precision at slow mouse movements (for example when

sniping), but will also allow you to turn 180 degrees quickly to aim at someone behind you.

Acceleration Threshold

Describes the speed above which mouse acceleration will be activated. If you move your hand above this set speed, mouse acceleration will kick in.

Rendering



Field of View

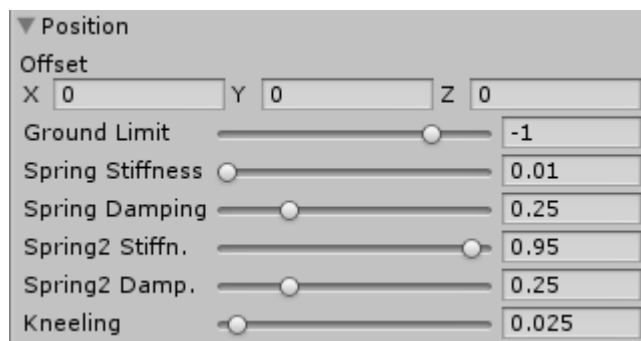
Sets the target FOV of the first person camera.

Zoom Damping

Determines how fast the camera will interpolate to the target value when FOV is changed from script. Higher values result in slower zoom.

Position

The camera position is hooked to two springs, allowing for a host of interesting effects. The first, "regular" spring is used for reacting to player movements that need a soft spring setting. The second spring is intended for external forces that require stronger spring settings, such as the camera shaking violently from an artillery impact.



Offset

Camera position offset relative to the player. For example: to create a crouch animation, you can smoothly lower the camera with the Y-offset. To peek around

corners without moving the character, you can move the camera sideways using x-offset.

This position is where the camera "wants to be". If you move it away from the target value using external force, it will immediately try to go back to the target position.

Ground Limit

A vertical limit intended to prevent the camera from intersecting the ground. The value is in negative world units relative to the camera position. It determines how far below the normal head position the camera is allowed to go when kneeling or reacting to external forces or camera bob.

Spring Stiffness

Camera spring stiffness - or mechanical strength - determines how loosely or rigidly the camera spring behaves. A low value will result in a soft, swaying motion, and higher values will result in stronger spring movements.

Spring Damping

Camera spring damping makes spring velocity wear off as it approaches its rest state. Low values will result in a very loose, swaying motion, while higher values can result in either motion that quickly grinds to a halt, or a stiff shaking motion (much depending on the spring stiffness setting).

Spring2 Stiffness

Stiffness for the secondary camera spring. This spring is intended for external forces. It does not automatically do anything. Instead, activate it using the "AddForce2" script function of vp_FPCamera. See the DemoScript for example usage.

Spring2 Damping

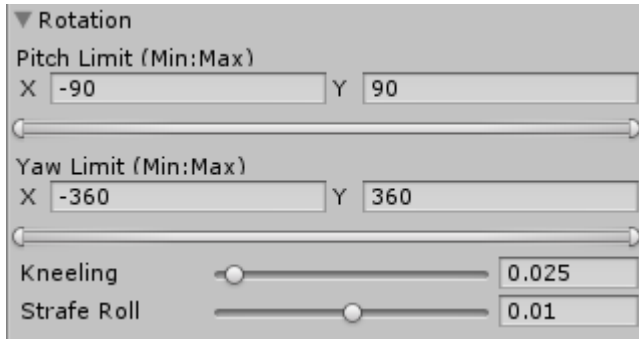
Damping for the secondary camera spring. This spring is intended for external forces. It does not automatically do anything. Instead, activate it using the "AddForce2" script function of vp_FPCamera. See the DemoScript for example usage.

Kneeling

Determines how much the camera will be pushed down when the player falls onto a surface.

Rotation

The camera is mainly rotated by mouse input and camera shakes. It also has a rotation spring, mostly used for rotating around the Z vector (rolling) by means of external forces.



Pitch Limit

Limits vertical mouse rotation of the camera.

Yaw Limit

Limits horizontal mouse rotation of the camera. Useful mainly for creating things like gun turrets. It could also be useful for players mounted on vehicles, let's say a player shooting out of a car side window.

Kneeling

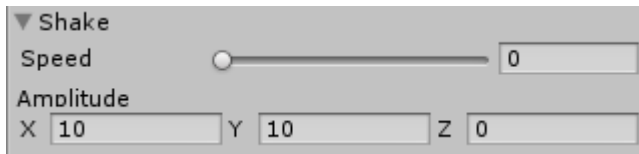
Determines how much the camera will be rotated when the player falls onto a surface. There is a 50% chance the camera will rotate left versus right upon impact.

Strafe Roll

This variable rotates the camera depending on sideways local velocity of the character controller, resulting in the camera leaning into or away from its sideways movement direction. It's useful when moving and crouching. Larger positive values (2.0 and above) will result in a motorcycle or airplane type behaviour.

Shake

Ultimate FPS Camera uses a standard perlin noise function for applying random turbulence to the weapon and camera. For example: a slight breathing camera motion as you zoom in with a sniper rifle, gentle idle movements for a hand holding a sub machine gun, or the heavily disturbed camera movements of a character that is drunk or poisoned.



Speed

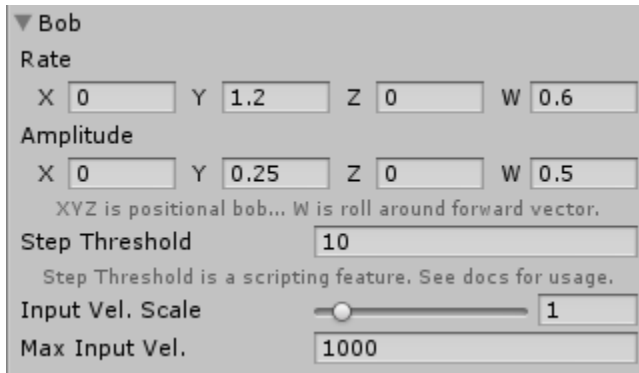
Determines the shaking speed of the camera.

Amplitude

The strength of the angular camera shake around the X, Y and Z vectors. X and Y shakes are applied to the actual controls of the character model while Z is only applied to the camera. If you increase the shakes, you will essentially get a drunken / sick / drugged movement experience. This can also be used for i.e. sniper breathing since it will affect aiming.

Bob

View bob is the sinusoidal relative motion of the camera, hooked to character controller velocity. If applying bob along both the x and y vector you can get interesting results such as the feeling of being a dinosaur or a huge ogre.



Rate

Speed of the camera bob. X, Y and Z is positional motion. W is roll around the forward vector (tilting the head from left to right). A typical value for Y is 0.9.

Amplitude

The strength of the camera bob. Determines how far the camera swings in each respective direction. For camera bob, X, Y and Z is positional motion. W is roll around the forward vector (tilting the head from left to right). A typical value for Y is 0.1.

Step Threshold

A feature for syncing bob motion with a script callback. The bob step callback is triggered when the vertical camera bob reaches its bottom value (provided that the speed is higher than the bob step threshold). This can be used for footstep sounds and various other behaviours. This feature is not automatically used by the system. You need to provide a callback through script. See the scripting chapter below for more info on this.

Input Velocity Scale

This tweak feature is useful if the bob motion goes out of hand after changing player velocity. Just use this slider to scale back the bob without having to adjust lots of values.

Max Input Velocity

A cap on the velocity value being fed into the bob function, preventing the camera from flipping out when the character travels at excessive speeds (such as when affected by a jump pad or a speed boost).

TIP: *Current speed used by the camera bob can be displayed through script using:*

```
Debug.Log (Controller.Velocity.sqrMagnitude);
```

FPWeapon

The `vp_FPWeapon` component handles all procedural motion properties of the weapon object. Weapons are implemented by adding child gameobjects to the `FPCamera` transform and dragging `vp_FPWeapon` components onto each gameobject. Remember that weapons need a `WeaponCamera` in order to render, see the above chapter "Getting Started -> Creating an `FPPlayer` from scratch".

When the app starts, any such gameobjects are put in a list and may be toggled on or off from script. The weapons are arranged in alphabetical order, so you have great control over the order of your weapons, for example by putting numbers at the beginning of their names in the Hierarchy.

In order to prevent them from sticking through walls and other geometry, weapons are rendered by their own camera which is created at runtime.

NOTE: The `m_AvailableWeapons` list in the `vp_FPPlayer` script is intended to contain all the weapons the player will ever be able to carry, and the game code should allow or prevent the player from activating them depending on the inventory state. You should not control weapon availability by adding or removing weapon gameobjects to the `FPCamera` hierarchy dynamically.

NOTE: The `vp_FPWeapon` class is not responsible for shooting logic, only for motion resulting from player movement or external forces. This is because an `FPWeapon` is not necessarily a firearm. It could just as easily be a knife, an axe, a crowbar or a couple of fists. All firearm specific features exist in a separate class called `vp_FPShooter`.

Rendering



Field of View

Field of View for the weapon camera. This setting gives you great control of the appearance of the weapon. Tweak this along with weapon position and rotation offset.

Zoom Damping

Determines how fast the weapon will interpolate to the target value when FOV is changed from script. Higher values result in slower zoom.

Clipping Planes

Determines the near and far clipping planes of the weapon camera. Very useful in situations where you want to use a specific FOV and positional / rotational offset for the weapon, but parts of the weapon mesh intersect the view too close to the camera. In these cases, simply increase the near clipping plane. This is best done at runtime by clicking and dragging on "Clipping plane: X" in the Inspector while holding the "Alt" key.

Position

The weapon is hooked to its own position and rotation springs. Just like the camera it supports bob, shakes and external forces. It applies player local velocity to its position when walking and falling. The weapon has special position and rotation springs for additional forces such as recoil. The pivot point of the weapon is also hooked to a spring, and manipulating this at runtime can be quite useful.

▼ Position

Offset

X 0.15 Y -0.15 Z -0.15

Exit Offset

X 0 Y -1 Z 0

Pivot

X 0 Y 0 Z 0

Pivot Stiffness 0.01

Pivot Damping 0.25

Show Pivot ☐

Pivot can be shown when the game is playing.

Spring Stiffness 0.01

Spring Damping 0.25

Spring2 Stiffn. 0.95

Spring2 Damp. 0.25

Spring2 is intended for recoil. See the docs for usage.

Kneeling 0.1

Fall Retract 1

Walk Sliding

X 0.5 Y 0.75 Z 0.5

Input Vel. Scale 1

Max Input Vel. 25

Offset

Weapon position offset relative to the weapon object position. This position is where the weapon "wants to be". If you move it away from the target value using external force, it will immediately try to go back to the target position.

Exit Offset

This position is used by the parent FPCamera for switching weapons. It will move the current weapon model to its Exit Offset, switch weapons and move the new weapon into view, starting at its Exit Offset.

Pivot

This setting manipulates the pivot point of the weapon model. A value of "0, 0, 0" will leave the object's pivot at the point defined in the 3d object file.

***TIP:** The relation between pivot and weapon model position is affected by rotations. This may lead to unexpected behaviour when moving the pivot in the editor. Before editing weapon pivot it is best to turn off weapon shake and set weapon rotation offset to (0, 0, 0).*

Pivot Stiffness

Pivot spring stiffness - or mechanical strength - determines how loosely or rigidly the pivot spring behaves. A low value will result in a soft, swaying motion, and higher values will result in stronger spring movements.

In some scripting situations you may need to move the weapon offset and pivot offset at the same time. In these cases you may want to make sure that the two springs share the same Stiffness and Damping settings, and are moved approximately the same distance, or one spring will finish before the other. On the other hand, this results in a more complex animation and may sometimes be desirable. If you run into problems related to pivot switching in script, try calling the "SnapPivot" method of vp_FPWeapon.cs.

Pivot Damping

Pivot spring damping makes spring velocity wear off as it approaches its rest state. Low values will result in a very loose, swaying motion, while higher values can result in either motion that quickly grinds to a halt, or a stiff shaking motion (much depending on the pivot stiffness setting).

Show Pivot

Toggles pivot point visualization on or off for editing purposes. The pivot point looks like a blue transparent ball. It can only be visualized when the game is playing.

***TIP:** Set Pivot Z to about -0.5 to bring it into view.*

Spring Stiffness

Weapon spring stiffness - or mechanical strength - determines how loosely or rigidly the weapon spring behaves. A low value will result in a soft, swaying motion, and higher values will result in stronger spring movements.

Spring Damping

Weapon spring damping makes spring velocity wear off as it approaches its rest state. Low values will result in a very loose, swaying motion, while higher values can result in either motion that quickly grinds to a halt, or a stiff shaking motion (much depending on the weapon stiffness setting).

Spring2 Stiffness

Positional stiffness for the second weapon spring. This spring is intended for additional forces such as recoil. It is not used internally by `vp_FPWeapon`, but the `vp_FPSShooter` component uses it for recoil. It can also be activated using the "AddForce2" script function of `vp_FPWeapon`.

Spring2 Damping

Positional damping for the second weapon spring. This spring is intended for additional forces such as recoil. It is not used internally by `vp_FPWeapon`, but the `vp_FPSShooter` component uses it for recoil. It can also be activated using the "AddForce2" script function of `vp_FPWeapon`.

Kneeling

Determines how much the weapon will be pushed down when the player falls onto a surface.

Fall Retract

Makes the weapon pull backward while falling.

Walk Sliding

Walk sliding moves the weapon in different directions depending on character controller movement direction.

- X slides the weapon sideways when strafing
- Y slides the weapon down when strafing (it can only move down)
- Z slides the weapon forward or backward when walking

Input Velocity Scale

This tweak feature is useful if the spring motion goes out of hand after changing player velocity. Just use this slider to scale back the spring motion without having to adjust lots of values.

Max Input Velocity

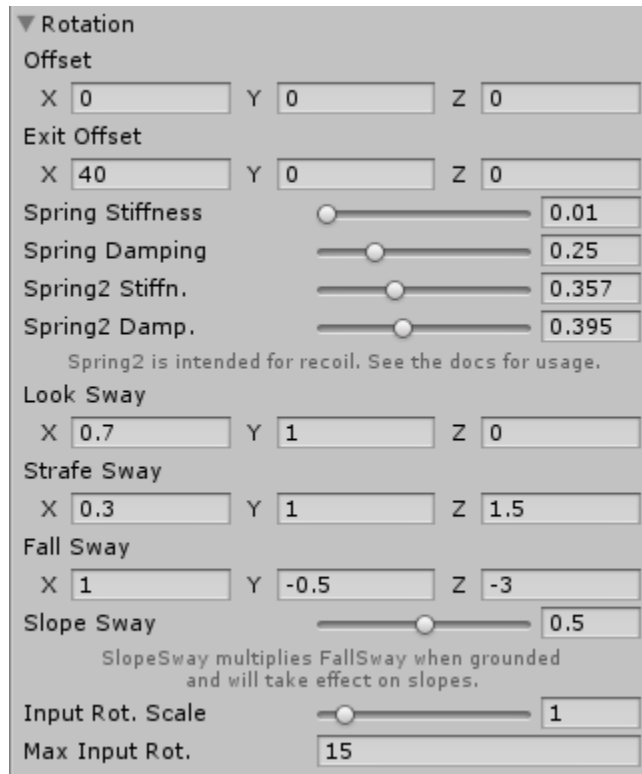
A cap on the velocity value being fed into the weapon swaying method, preventing the weapon from flipping out when the character travels at excessive speeds (such as when affected by a jump pad or a speed boost). This also affects fall sway.

TIP: Current speed used by weapon swaying can be displayed through script using:

```
Debug.Log (Controller.Velocity.magnitude);
```

Rotation

Weapon rotation is affected via springs by mouse movements and character controller motion. It can also be affected by external forces such as falling impact, and additional forces such as recoil.



The screenshot shows a 'Rotation' settings panel with various fields and sliders. The 'Offset' section has X, Y, and Z input fields, all set to 0. The 'Exit Offset' section has X, Y, and Z input fields, with X set to 40 and Y and Z set to 0. The 'Spring Stiffness' section has a slider and a numeric field set to 0.01. The 'Spring Damping' section has a slider and a numeric field set to 0.25. The 'Spring2 Stiffn.' section has a slider and a numeric field set to 0.357. The 'Spring2 Damp.' section has a slider and a numeric field set to 0.395. A note below these states: 'Spring2 is intended for recoil. See the docs for usage.' The 'Look Sway' section has X, Y, and Z input fields, with X set to 0.7, Y set to 1, and Z set to 0. The 'Strafe Sway' section has X, Y, and Z input fields, with X set to 0.3, Y set to 1, and Z set to 1.5. The 'Fall Sway' section has X, Y, and Z input fields, with X set to 1, Y set to -0.5, and Z set to -3. The 'Slope Sway' section has a slider and a numeric field set to 0.5. A note below this states: 'SlopeSway multiplies FallSway when grounded and will take effect on slopes.' The 'Input Rot. Scale' section has a slider and a numeric field set to 1. The 'Max Input Rot.' section has a numeric field set to 15.

Property	X	Y	Z	Value
Offset	0	0	0	
Exit Offset	40	0	0	
Spring Stiffness				0.01
Spring Damping				0.25
Spring2 Stiffn.				0.357
Spring2 Damp.				0.395
Look Sway	0.7	1	0	
Strafe Sway	0.3	1	1.5	
Fall Sway	1	-0.5	-3	
Slope Sway				0.5
Input Rot. Scale				1
Max Input Rot.				15

Offset

Weapon rotation offset. This angle is where the weapon "wants to be". If you turn it away from the target value using external force, it will immediately try to swing back to the target angle.

Exit Offset

This angle is used by the parent FPCamera for switching weapons. It will move the current weapon model to its Exit Offset, switch weapons and move the new weapon into view, starting at its Exit Offset.

Spring Stiffness

Weapon rotation spring stiffness determines how loosely or rigidly the weapon rotation spring behaves. A low value will result in a soft, swaying motion, and higher values will result in stronger spring movements.

Spring Damping

Weapon rotation spring damping makes spring velocity wear off as it approaches its rest state. Low values will result in a very loose, swaying motion, while higher values can result in either motion that quickly grinds to a halt, or a stiff shaking motion (much depending on the weapon stiffness setting).

Spring2 Stiffness

Angular stiffness for the second weapon spring. This spring is intended for additional forces such as recoil. It is not used internally by `vp_FPWeapon`, but the `vp_FPSShooter` component uses it for recoil. It can also be activated using the "AddForce2" script function of `vp_FPWeapon`.

Spring2 Damping

Angular damping for the second weapon spring. This spring is intended for additional forces such as recoil. It is not used internally by `vp_FPWeapon`, but the `vp_FPSShooter` component uses it for recoil. It can also be activated using the "AddForce2" script function of `vp_FPWeapon`.

Look Sway

This setting determines how much the weapon sways (rotates) in reaction to mouse movements. Horizontal and vertical mouse movements will sway the weapon spring around the Y and X vectors, respectively. Rotation around the Z vector is hooked to horizontal mouse movement, which is very useful for swaying long melee weapons such as swords or clubs.

Strafe Sway

Rotation strafe sway rotates the weapon in different directions depending on character controller movement direction.

- X rotates the weapon up when strafing (it can only rotate up)
- Y rotates the weapon sideways when strafing
- Z twist the weapon around the forward vector when strafing

Fall Sway

This setting rotates the weapon in response to vertical motion (e.g. falling or walking in stairs). Rotations will have opposing direction when falling versus rising. However, the weapon will only rotate around the Z vector while moving downwards / falling.

Slope Sway

This parameter reduces the effect of weapon Fall Sway when moving on the ground. At a value of 1.0, the weapon behaves as if the player was falling. A value of 0.0 will disable Fall Sway altogether when the controller is grounded.

Input Rotation Scale

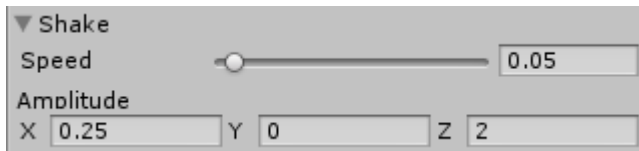
A tweak feature that can be used to temporarily alter the impact of mouse input motion on the weapon rotation spring, for example in a special player state.

Max Input Rotation

A cap on the mouse input motion being fed into the weapon swaying method, preventing the weapon from flipping out when extreme mouse sensitivities are being used.

Shake

This is procedural weapon rotation shaking, intended as a purely aesthetic motion to breathe life into the weapon. Useful for idle animations.



Speed

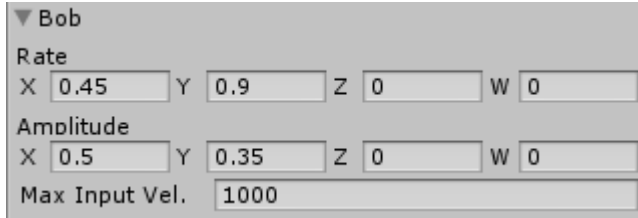
Determines the shaking speed of the weapon.

Amplitude

The strength of the angular weapon shake around the X, Y and Z vectors.

Bob

Weapon bob is the sinusoidal motion of the weapon, hooked to character controller velocity.



▼ Bob

Rate

X 0.45 Y 0.9 Z 0 W 0

Amplitude

X 0.5 Y 0.35 Z 0 W 0

Max Input Vel. 1000

Rate

Speed of the weapon bob. X & Z rate should be (Y/2) for a classic weapon bob. To invert the curve, make X and Y negative. For weapon bob, X, Y and Z is angular bob. W is the position along the forward vector (pushing back and forth). A typical value for Y is 0.9.

Amplitude

The strength of the weapon bob. Determines how far the weapon swings in each respective direction. For weapon bob, X, Y and Z is positional motion. W is roll around the forward vector (tilting the head from left to right). A typical value for Y is 0.1.

Max Input Velocity

A cap on the velocity value being fed into the bob function, preventing the weapon from flipping out when the character travels at excessive speeds (such as when affected by a jump pad or a speed boost).

TIP: Player velocity can be output through script using:

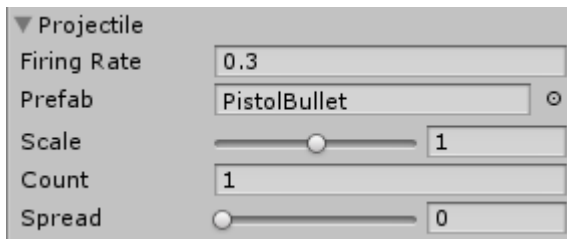
```
Debug.Log (Controller.velocity.magnitude);
```

FPShooter

The vp_FPShooter class adds firearm properties to a weapon. The component should be added to a weapon GameObject that already has a vp_FPWeapon component added to it. The script will manipulate the weapon component for recoil and manages firing rate, accuracy, sounds, muzzle flashes and spawning of projectiles and shell casings. It also has basic ammo and reloading features.

Projectile

The Projectile foldout handles the initial behavior of any projectile objects spawned by the weapon.



Firing Rate

The firing rate of the weapon. If the user fires continuously, shots will be fired using this as the minimum time interval in seconds.

Prefab

This should be a gameobject with a projectile logic script added to it, such as vp_Bullet. But remember that a projectile does not *have to* use a vp_Bullet script. You could put any gameobject here and a copy of it will be spawned when the weapon is fired. Feel free to write your own crazy projectile components! For more information about creating a bullet prefab, see the "Bullet" chapter below.

NOTE: *Gameobjects can not be saved using the preset system, so they need to be hooked manually each time you create a new vp_FPShooter component.*

Scale

This parameter will scale each projectile object by the set amount. If using a vp_Bullet projectile, this will be the scale of the resulting bullet hole object.

Count

The amount of projectiles to be fired simultaneously. Each projectile will get its own unique spread. However only one shell will be ejected and the muzzleflash will display as if one projectile was fired.

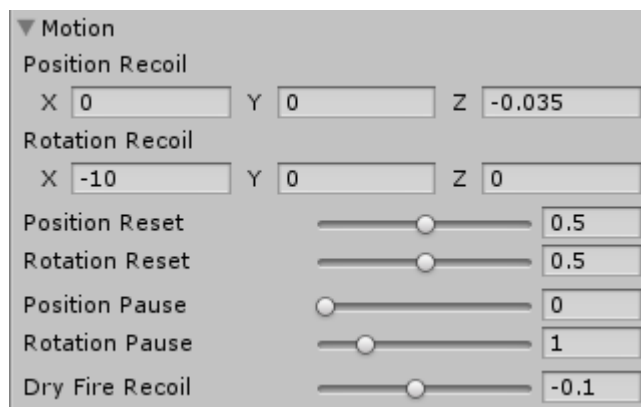
Spread

The conical deviation of the projectile. This parameter randomly alters the direction of the projectile by "Spread" degrees. A value of 1 means the projectile will deviate 1 degree within a cone from the player to the aim point. A value of 360 means the projectile will be emitted from the player in a completely random direction (essentially within a sphere). This is good for scripting shotgun type weaponry or manipulating accuracy at runtime.

***TIP:** To emulate the famous Classic DOOM shotgun, set "Firing Rate" to 1, "Count" to 7, "Spread" to 6 and "Shell -> Eject Delay" to 0.5. Oh yeah..*

Motion

The Motion foldout contains parameters that manipulate the position and rotation of the weapon when discharged.



The screenshot shows a 'Motion' foldout menu with the following settings:

Category	X	Y	Z
Position Recoil	0	0	-0.035
Rotation Recoil	-10	0	0

Below these are sliders for:

- Position Reset: 0.5
- Rotation Reset: 0.5
- Position Pause: 0
- Rotation Pause: 1
- Dry Fire Recoil: -0.1

Position Recoil

A force added to the secondary position spring upon firing the weapon. Setting Z to a negative number will make the weapon increasingly "kick like a mule". Keep in mind that achieving a good positional recoil depends in large part on tweaking "Position->Spring2" of your vp_FPWeapon component. Check out the demo weapon presets for some example Spring2 Stiffness and Damping settings.

Rotation Recoil

A force added to the secondary rotation spring upon firing the weapon. Setting X to a negative number will make the weapon twist upward like a pistol. Keep in mind that achieving a good angular recoil depends in large part on tweaking "Rotation->Spring2" of your vp_FPWeapon component. Check out the demo weapon presets for some example Spring2 Stiffness and Damping settings.

Position Reset

Upon firing, the primary position spring will snap back to its rest state by this factor.

Rotation Reset

Upon firing, the primary position spring will snap back to its rest state by this factor. This can be used to make a weapon always fire in the forward direction regardless of current weapon angles.

Position Pause

Upon firing, any forces acting on the primary position spring will freeze and fade back in over this interval in seconds.

Rotation Pause

Upon firing, any forces acting on the primary rotation spring will freeze and fade back in over this interval in seconds. This is typically useful if the weapon has a pronounced Fall Sway and the player fires it in mid-air. Without a Rotation Pause, the weapon may fire upwards or sideways while falling.

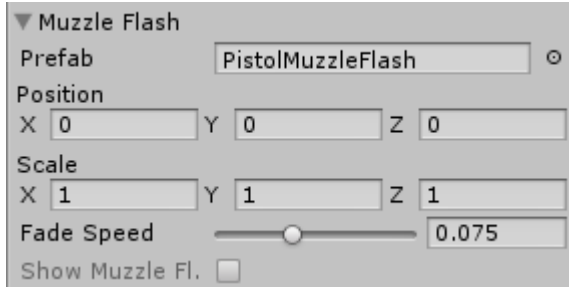
Dry Fire Recoil

This parameter multiplies the recoil value when the weapon is out of ammo. This can be used to simulate pulling the trigger with no discharge.

***TIP:** make 'MotionDryFireRecoil' about -0.1 for a subtle out-of-ammo effect.*

Muzzle Flash

The Muzzle Flash foldout handles logic for displaying the weapon's muzzle flash (if any) and animating its rotation and opacity.



Prefab

This should be a mesh with a "Particles/Additive" shader and a `vp_MuzzleFlash` script added to it. For more information about creating a muzzle flash prefab, see the "MuzzleFlash" chapter below.

NOTE: Gameobjects can not be saved using the preset system, so they need to be hooked manually each time you create a new `vp_FPSHooter` component.

Position

Muzzle flash position offset relative to the `FPCamera`. Set Position Z to a value larger than 0.5 to bring it into view.

Scale

This parameter will scale the muzzle flash object by the set amount.

Fade Speed

This amount of alpha will be deducted from the muzzle flash shader 60 times per second. When the weapon is discharged, the muzzle flash will be set to full alpha. It will then immediately start fading out at "Fade Speed". Note that the default (full) alpha for the "Particles/Additive" shader is **0.5**.

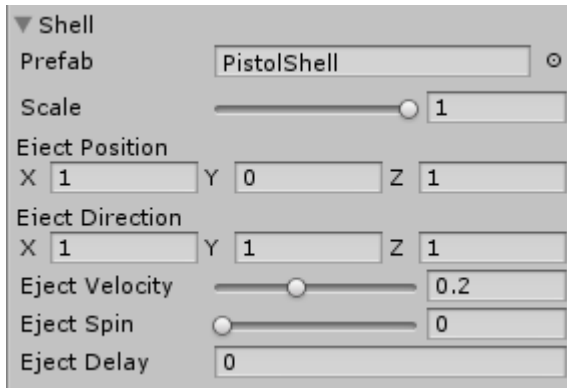
Show Muzzle Flash

Toggles muzzle flash visualization on or off for editing purposes. The muzzle flash can only be visualized when the game is playing.

TIP: Set Position Z to a value larger than 0.5 to bring it into view.

Shell

The parameters under this foldout govern how shell casings are spawned and set in motion by the weapon.



Prefab

A gameobject that will be ejected from the weapon upon firing. This should be a mesh with a `vp_Shell` script added to it. For more information about creating a shell prefab, see the "Shell" chapter below.

NOTE: Gameobjects can not be saved using the preset system, so they need to be hooked manually each time you create a new `vp_FPSHooter` component.

Scale

This parameter will scale each ejected shell object by the set amount.

Eject Position

Shell eject position offset relative to the `FPCamera`. Set Position Z to atleast 0.5 to bring it into view.

Eject Direction

The vector in relation to the `FPCamera`, along which shells will be ejected from the Eject Position. To send a shell flying forward, upward and to the right, set Eject Direction to 1, 1, 1.

Eject Velocity

The amount of force (positional speed) added to the shell's rigidbody object when instantiated. This is what sends the shell flying.

Eject Spin

When a shell is instantiated it receives a completely random torque (rotation speed, or spin). This parameter scales the spin. A value of 0 means the shell won't spin at all. A value of 1 means it will spin like crazy.

Eject Delay

Time to wait before ejecting the shell after firing. This is very useful for i.e. shotguns and grenade launchers. For example, you could use a traditional animation for a pump action shotgun, and sync "Eject Delay" to the animation.

Ammo

▼ Ammo	
Max Count	<input type="text" value="20"/>
Reload Time	<input type="text" value="1.5"/>

Max Count

This parameter holds the maximum amount of bullets that this weapon can hold (i.e. one full clip / mag).

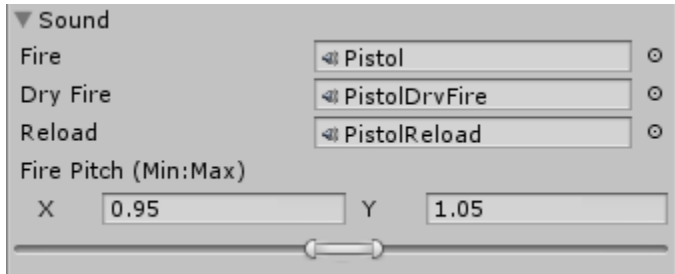
NOTE: This (along with the FPPlayer's "AvailableWeapons" list) is intended as placeholder inventory support. Some programmers will want to alter this functionality when hooking up their own inventory code. Implementations may differ quite a lot between games and may depend on whether it's an online or offline game, your choice of network solution and so on.

Reload Time

This delay in seconds determines for how long to prevent firing upon reload.

Sound

The Sound foldout has links to any sound content to be played by the weapon.



Fire

The standard firing sound of the weapon.

NOTE: Gameobjects can not be saved using the preset system, so they need to be hooked manually each time you create a new vp_FPSooter component.

Dry Fire

This sound is played if the player attempts to fire when the weapon is out of ammo.

Reload

The standard reload sound of the weapon.

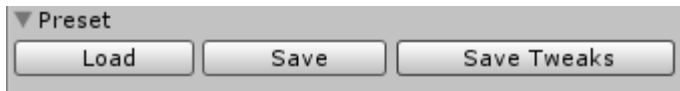
Fire Pitch

This parameter optionally pitches the sound of each shot fired slightly different to get a more organic firing sound. Variations in pitch should be kept minimal. A typical Min-Max range would be 0.95-1.05. This parameter allows you to use the same audio file with different pitch for different weapons.

Presets

One of the most powerful features in this package is the "ComponentPreset" class, which allows you to save component snapshots to text files. Presets can be loaded via script at runtime and used to quickly manipulate all the parameters of a component as needed by the gameplay. You'll find the "Preset" foldout at the bottom of all components in the inspector.

Load & Save



These buttons will open a file dialog allowing you to load or save presets to disk. Saving a preset will result in all the current parameter values of the component being dumped to disk. Loading a preset will alter the current component values with data from the preset. Preset text files are meant to be assigned to player states under the State foldout (see the "States" chapter below).

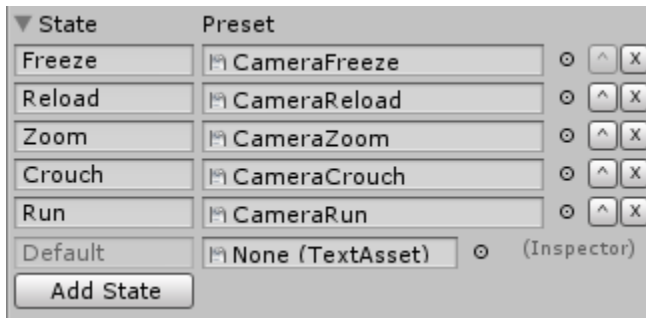
NOTE: Assets (such as 3d objects and sounds) can not be loaded or saved via text presets. This is because the Unity Editor AssetDatabase is no longer accessible once the game has been built. Inspector slots for sound, 3d meshes and textures always have to be set manually in the Inspector or via scripting methods.

Save Tweaks

This feature will save **partial presets**, that is: create or update a preset with only the values modified since the last time you pressed Play or changed the states at runtime. The purpose of this will usually be to create presets for use with the State Manager.

For example: the player may pick up a powerup that should only affect running speed. To create a preset for this, you could select the FPController component in the Inspector, start the game, alter the "MotorAcceleration" parameter and press "Save Tweaks". This would result in a FPController preset with only the acceleration parameter in it. See the "States" chapter below to learn how to assign such a partial preset to a player state and bind it to a button.

States




Any fast paced FPS will have the player switching between different modifier keys quickly and repeatedly (e.g. crouching, zooming, running). This calls for swapping lots of presets at runtime, usually requiring lots of scripting, for all the special cases and key combos. Ultimate FPS Camera's *State Manager* simplifies this by moving much of the work into the Editor and solving runtime preset blending automagically. It will combine multiple presets smoothly, depending on a current combination of *States*.

NOTE: States are completely optional to use. If you don't need them, don't add any states and your component will simply use the current Inspector settings as its Default and only state.

States can be used not only for input, but for messing with the player in many different ways. For instance, you could use timers to activate a "ShellShock" state shaking the camera for a limited amount of time, or you could use triggers to prevent the player from enabling a certain state in a certain area.

Creating a new state

1. Select an FPS component in the hierarchy (for example the FP_Camera).
2. Expand the "State" foldout.
3. Press the "Add State" button . A new empty state will appear.
4. Name the state by clicking and typing a name into the "New State" box. Remember that names are case sensitive.
5. Drag a preset text asset from the Project View into the "None (TextAsset)" box (the default text presets are located in the "Presets" folder in the Ultimate FPS Camera root).
6. Done. You now have a new state with a preset assigned to it. It won't do anything yet, though. The state needs to be activated via script.

Activating and deactivating states

Here's an example of toggling a state on and off via script.

- In the "Update" method of the "vp_FPPlayer" script, add the following lines, replacing "MyState" with the name of your state.

```
if (Input.GetKeyDown(KeyCode.H))  
    SetState("MyState", true);  
if (Input.GetKeyUp(KeyCode.H))  
    SetState("MyState", false);
```

When the user presses "H" this code will broadcast the state to all FPS components available in the player gameobject, working its way down the hierarchy.


- You can also toggle states on or off at runtime by clicking on their names. This is sometimes useful when creating partial presets ("Save Tweaks").




NOTE: There are many other ways of using states. Study the `vp_FPPlayer` script to get more acquainted with some different methods of working with states and timers.

State order


The state system works basically like the layers of a paint program. The state (layer) at the bottom represents the current inspector values of a component. The state above it represents a preset that will override all - or just some - of the underlying one's values, and so on up the list. If a state is disabled, the remaining presets in the list will be smoothly recombined (let's say you release the Crouch key while still holding the Zoom key).

To bump up a state, press the little arrow button  at the right and the state will jump up one step. Any currently active states in the top will override any currently active states below them.


The Default state

The bottommost state is called the 'Default' state. It represents the standard mode of the component and cannot be removed or renamed. It should usually be left empty: "None (TextAsset)", in which case it uses the current Inspector settings for the component. If you do choose to assign a text preset to the Default state, it will lock down the component, preventing editing. If so the component can be modified at runtime only, and changes will revert when the app stops (presets can still be saved though). To re-enable editing, the 'Default' state must be unlocked by pressing the small 'Unlock' button .

Deleting a state

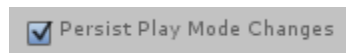
To delete a state simply press the small "x" button  at the far right. Beware though; deleting a state can't be undone.

Removing a preset

To remove a preset from a state, press the small circle icon  to open a "Select TextAsset" window. At the top of the list, select "None".

Persist play mode changes

The Default state can optionally be "persisted", meaning the variables you change during play will not be reset when the application stops. Whenever this checkbox is unchecked and the app is stopped, Unity will reset the Default state to what it was before you pressed play (the default Unity behaviour). If it's checked, Unity will stay away from your variables. Please note that only the currently selected component will be persisted even if the checkbox is checked on several components.



Bullet

Ultimate FPS camera comes with `vp_Bullet`, a generic class for *hitscan projectiles*. The `vp_Bullet` script should be attached to a gameobject with a mesh to be used as the impact decal (bullet hole). When a gameobject with a `vp_Bullet` component is instantiated, it immediately raycasts straight ahead for a certain range, and moves itself to the surface of the first object hit. It also spawns a number of particle effects and plays a random impact sound.

NOTE: "Hitscan" means that the game uses immediate raycasting rather than Newtonian physics to move the bullet. This is the way in which bullets work in the majority of first person shooter games.

Creating a Bullet prefab from scratch

1. Create a **bullet hole texture** with and alpha channel, to be used with a transparent material / shader.

TIP: The provided demo prefab uses the "Transparent/Bumped Diffuse" shader. The diffuse channel (A) contains the hole and a soft gradient. It's alpha channel (B) emphasizes the hole and has a very slight gradient too. The normal map (C) does the real work and simulates a hole or dent in a hard surface. This type of bullet hole is very versatile since it will adapt to the color of any surface it is put on top of. It will also work great with lighting.



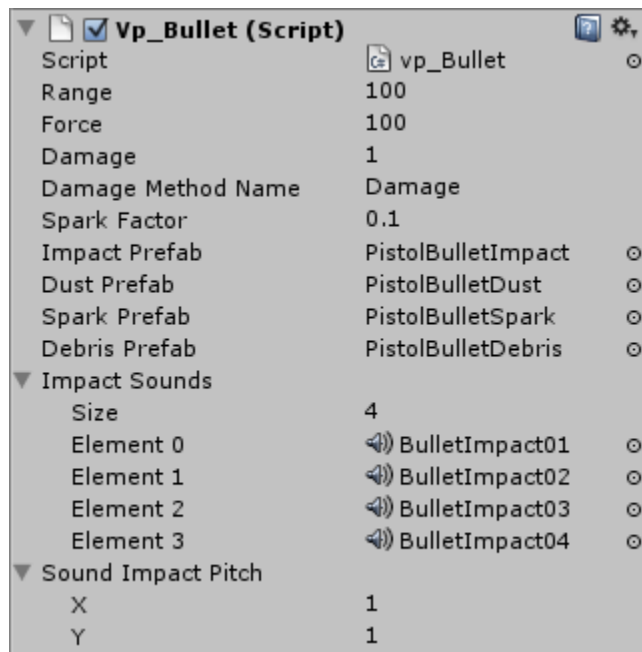
2. Create a **decal mesh** for the bullet hole in your favorite 3d package, facing along the Z vector. It is recommended to make the mesh 1x1 unit large. Also, a good thing is to displace it very slightly in the Z (forward) direction. This will put its geometry slightly in front of any surface hit and will reduce or prevent issues with z-fighting.

NOTE: If you are not doing anything special with the mesh, it is recommended to just use the provided decal mesh.

3. In Unity, create an empty gameobject and name it something.

4. Drag your decal mesh onto the gameobject (make sure its import scale and orientation is correct).
5. Drag the vp_Bullet script onto the gameobject.
6. Adjust the parameters of the vp_Bullet script (see info on the parameters below).
7. When happy, drag your gameobject from the Unity Hierarchy to the Project view in order to create a prefab. Next, this prefab can be dragged onto the "Projectile -> Prefab" field of a vp_FPShooter component.

Parameters



Range

The max travel distance of this type of bullet in meters.

Force

Force applied to any rigidbody hit by the bullet (note that if your FPSooter spawns multiple projectiles, each bullet will add its own force).

Damage

The amount of damage (in the form of a floating point number) that should be transmitted to each target hit by this bullet.

Damage Method Name

The name of the public damage method you are using in your other gameobject scripts. By default this is "Damage".

***TIP:** Ultimate FPS Camera features an example damage script. See the chapter about "DamageHandler" below.*

***TIP:** By having multiple damage methods on the targets, this can be used to apply unique types of damage per projectile type/prefab, for example "MagicDamage", "FreezingDamage", "PoisonDamage", "ElectricDamage" and so on.*

SparkFactor

Chance of bullet impact spawning the bullet's "Spark" particle FX prefab.

Impact Sounds

This is the bullet's list of impact sounds. When a bullet hits something, one of these will be randomly selected and played. There is no limit to the amount of sounds. In order to add sounds to an empty list, change the "Size " parameter to the amount of sounds you intend to add, and this number of fields will appear.

***TIP:** Try and alternate the impact sounds a little bit. Using a few different ricochets can be very effective.*

Sound Impact Pitch

This parameter optionally pitches the sound of each impact slightly different. Variations in pitch should be kept minimal. "X" is "Min" and "Y" is "Max". A typical Min-Max range would be 0.95-1.05.

Particle FX Prefabs

The bullet will optionally spawn a number of gameobjects at the point and moment of impact. Technically these prefabs could be anything, but their intended uses are as follows:

***TIP:** The example particle effects included in Ultimate FPS Camera are actually pretty basic. Study their prefabs to learn how they are put together.*

ImpactPrefab

A flash or burst illustrating the shock of impact.

DustPrefab

Evaporating dust or moisture from the hit material.

SparkPrefab

A quick spark, as if hitting stone or metal.

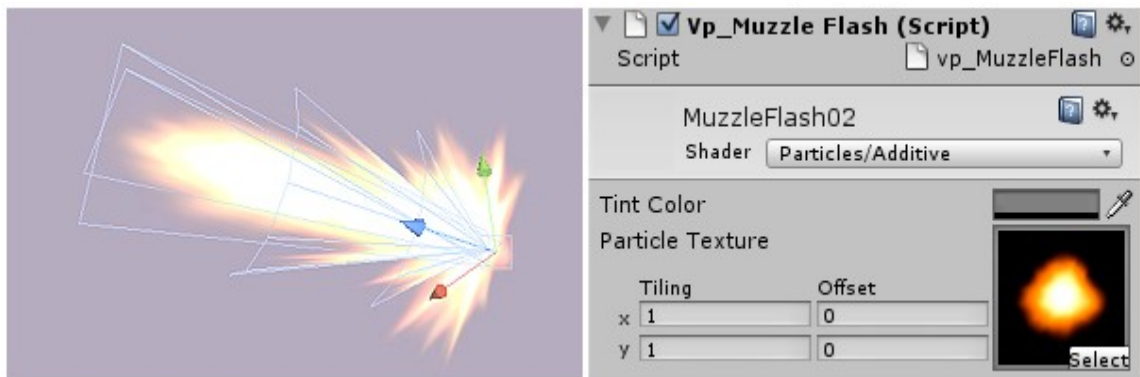
DebrisPrefab

Pieces of material thrust out of the bullet hole and / or falling to the ground.

MuzzleFlash

vp_MuzzleFlash handles logic for showing an additive, randomly rotated muzzle flash in front of the weapon. The script should be attached to a gameobject with a mesh to be used as the muzzle flash, using the "Particles/Additive" shader.

This gameobject will always be enabled but most often invisible. When a shot is fired, it snaps to a new random rotation and its alpha is set to full. It then immediately fades out.



Creating a MuzzleFlash prefab from scratch

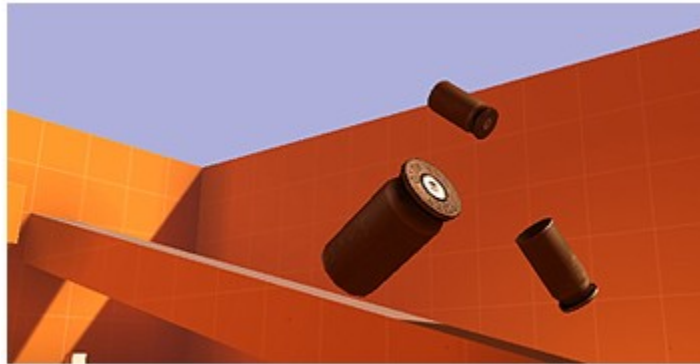
1. Create a **muzzle flash texture** with an alpha channel, to be used with the "Particles/Additive" shader.
2. Create a **mesh** for the muzzle flash in your favorite 3d package, facing along the Z vector.
3. In Unity, create an empty gameobject and name it something.
4. Drag your mesh onto the gameobject. Make sure its import scale and orientation is correct. Also, make sure that the muzzle flash never gets a collider component, or it will mess with your character controller's motion.
5. Drag the vp_MuzzleFlash script onto the gameobject.
6. Set the shader of your object to "Particles/Additive".

NOTE: Sometimes the muzzle flash may display briefly (for one frame) right when the game is started. In order to prevent this, set the initial alpha of its material to 0.

7. When happy, drag your gameobject from the Unity Hierarchy to the Project view in order to create a prefab. Next, this prefab can be dragged onto the "MuzzleFlash -> Prefab" field of a vp_FPSHooter component.

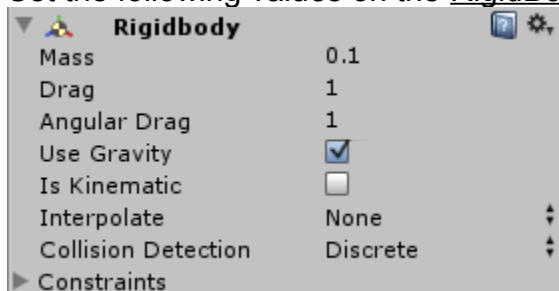
Shell

vp_Shell is a script for managing the behavior of shell casings ejected from weapons. It uses managed rigidbody physics to behave in a manner typical of bouncing shell casings. It also plays random bounce sounds. The script should be added to a gameobject with a mesh to be used as the shell casing. The vp_Shooter component which spawned the shell is responsible for setting its initial velocity and random spin.



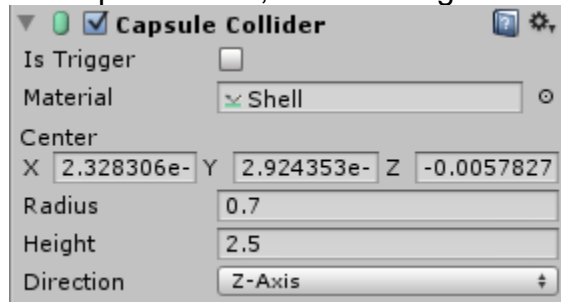
Creating a Shell prefab from scratch

1. Create a **textured shell mesh** in your favorite 3d package, facing along the Z vector. It is recommended to make the shell 1x1 unit large, and using the "Scale" parameter of vp_FPSooter to scale it (at a later stage).
2. In Unity, create an empty gameobject and name it something.
3. Drag your custom mesh onto the gameobject. Make sure its import scale and orientation is correct.
4. Drag the vp_Shell script onto the gameobject.
5. Set the following values on the RigidBody of the gameobject:



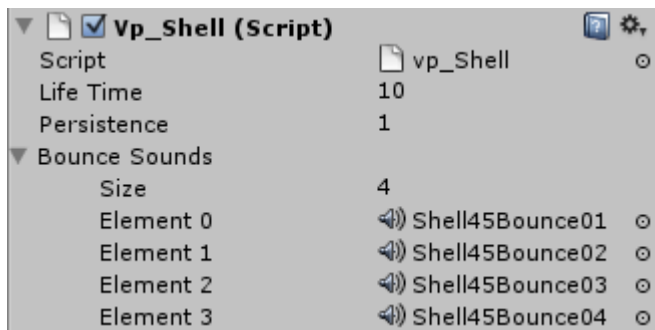
6. Adjust the Capsule Collider of the gameobject. Begin by dragging the **"Weapons/Shell/Shell.physicMaterial"** to the "Material" slot. For the rest

of the parameters, the following tested values can be a good start:



7. Adjust the parameters of the vp_Shell component (see info on the parameters below).
8. When happy, drag your gameobject from the Unity Hierarchy to the Project view in order to create a prefab. Next, this prefab can be dragged onto the "Shell -> Prefab" field of a vp_FPSHooter component.

Parameters



Life Time

Time to live in seconds for this type of shell. When the time has passed, the shell is quickly and discretely shrunk and destroyed.

Persistence

This is the chance of a shell *not* being removed after settling on the ground. An optional optimization feature for weapons that emit large amounts of shells. A value of **0** means that *all shells will be removed* after bouncing once or twice. A value of **0.5** means that roughly *half of the shells will be removed* early. A value of **1** means that *all shells will stick around* for the remainder of their lifetime.

Bounce Sounds

This is the shell's list of bounce sounds. When a shell hits the ground sufficiently hard, one of these will be randomly selected and played. There is no limit to the amount of sounds. In order to add sounds to an empty list, change the "Size" parameter to the amount of sounds you intend to add, and new slots will appear.

Damage Handler

vp_DamageHandler is a helper class to give your scene objects the capability of taking damage, dying and respawning. To use it, just drag it to a gameobject inside your level and tweak its parameters. You should then be able to shoot and kill it.

When the DamageHandler's "Health" goes below zero, it will call its "Die" method to remove the object, restore health to the original value and respawn after some time in the original location.

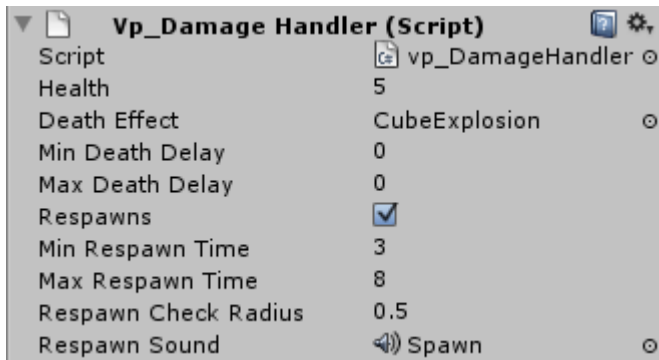
Any other script can send damage to the DamageHandler using the following code:

```
hitObject.SendMessage("Damage", 1.0f, SendMessageOptions.DontRequireReceiver);
```

Where "hitObject" is the DamageHandler's gameobject and "1.0f" is the amount of damage to do.

NOTE: This script won't work on an FPPlayer because of the player's complex gameobject hierarchy. For a simple example on how to handle player damage, have a look at the example "Damage" method in vp_FPPlayer.

Parameters



Health

Initial health of the object instance, to be reset on respawn.

DeathEffect

Gameobject to spawn when the object dies. The included examples all use an explosion, but this could also be a gameobject hierarchy with rigidbody rubble, such as the shards of a vase or wood debris from a crate.

MinDeathDelay & MaxDeathDelay

Random timespan in seconds to delay death. Good for designing cool serial explosions!

Respawns

Whether to respawn the object or just delete it upon death.

MinRespawnTime & MaxRespawnTime

Random timespan in seconds to delay respawn. This can be used to make respawning less predictable.

RespawnCheckRadius

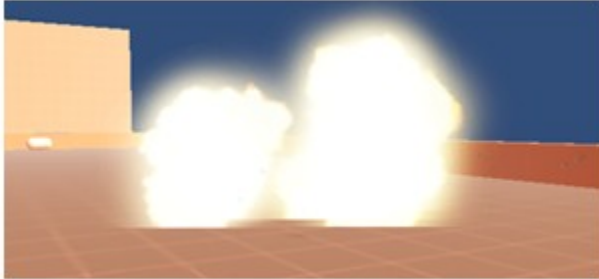
Radius around the object that must be cleared of other objects before respawn can take place. If the spawn area is occupied the DamageHandler will reschedule respawn using a new RespawnTime.

RespawnSound

Sound to play upon respawn.

Explosion

vp_Explosion is a simple death effect for exploding objects. It will apply damage and a physical force to all rigidbodies and FPPlayers within range. It will also play a sound and spawn a list of FX gameobjects. These would typically be particles, but could also contain things like image effects. The explosion gameobject destroys itself as soon as the sound has stopped playing.



Spawning an Explosion from script

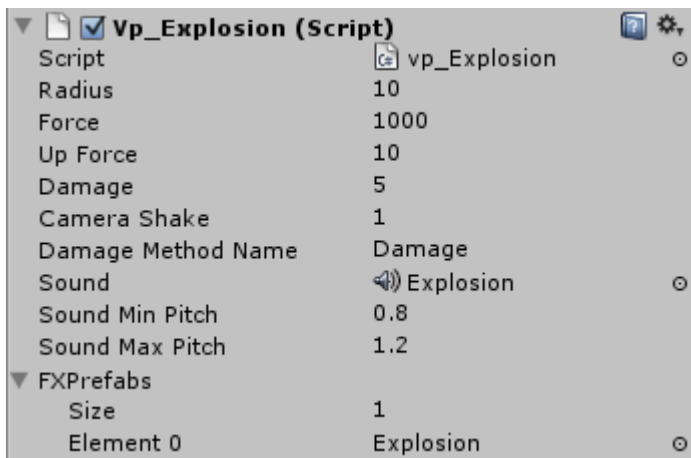
The included explosion prefab is intended as an example death effect for the vp_DamageHandler, but you can also spawn an explosion from script very easily with the following lines of code:

```
Object.Instantiate(ExplosionPrefab, position, Quaternion.identity);
```

Where "position" is the desired vector3 location of your blast, and "ExplosionPrefab" is a GameObject declared as public variable at the top of your script and assigned in the Editor:

```
public GameObject ExplosionPrefab = null;
```

Parameters



Radius

Any objects within this radius in meters will be affected by the explosion.

Force

Amount of motion force to apply to affected objects. Force will wear off with distance.

UpForce

How much to push affected objects up into the air. This will only take effect on objects that are currently grounded.

Damage

Amount of damage to apply to objects via their "Damage" method. Damage will wear off with distance.

CameraShake

How much of a shockwave impulse to apply to the camera. The impulse will wear off with distance.

DamageMethodName

User defined name of the damage method on the affected object.

***TIP:** By using different damage method names for different prefabs (and creating corresponding methods in your target object scripts) multiple damage types can be supported. For instance: MagicDamage, FreezingDamage, PlasmaDamage etc.*

SoundMinPitch & SoundMaxPitch

Random pitch range for the explosion sound. This will add variation and a little realism. Variations in pitch should be kept minimal. A typical Min-Max range would be 0.95-1.05.

FXPrefabs

This is a list of special effects objects to spawn when the explosion goes off. This would typically be particle effects but may also contain rigidbody rubble or wreckages. There is no limit to the amount of objects. In order to add objects to an empty list, change the "Size " parameter to the amount of objects you intend to add, and the corresponding number of fields will appear.

Image Effects



Image Effects (Unity Pro only) are an awesome addition to any FPS and will give your game that next-gen look. Ultimate FPS Camera has been tested with all image effects available in the Pro Standard Assets package (see below for compatibility notes). For more information about how to get started with image effects, check out the following links:

<http://docs.unity3d.com/Documentation/Manual/HOWTO-InstallStandardAssets.html>

<http://docs.unity3d.com/Documentation/Components/comp-ImageEffects.html>

Important notes

- Image effects should be assigned to one of the two cameras available under your FPPlayer gameobject hierarchy - usually the WeaponCamera gameobject.
- Image effects assigned to the WeaponCamera will affect the scene AND the weapon.
- Image effects assigned to the FPCamera will only affect the scene, NOT the weapon.
- Image effects should never be assigned to the FPPlayer object! This will auto-create an unneeded Unity Camera component and mess up rendering. If this happens by accident, make sure to remove both the image effect and the auto-generated Camera component from your FPPlayer gameobject.



Rendering Path considerations

The *Rendering Path* of the two cameras will influence the behavior of image effects.

Forward Rendering

When *Forward Rendering* is enabled, image effects will work on either the FPCamera (maincam) or the WeaponCamera, as described above.

Deferred Lighting

When *Deferred Lighting* is enabled, image effects will only work if assigned to the WeaponCamera. The effects will render on both the scene and the weapon.

IMPORTANT: When Deferred Lighting is enabled, the FPCamera (maincam) and the WeaponCamera must use the same HDR setting (on / off), otherwise there will be serious rendering artifacts.

Use Player Settings

If a camera's rendering path is set to *Use Player Settings*, you may alter it by going to Unity's top menu and "Edit -> Project Settings -> Player -> Per Platform Settings -> Other Settings -> Rendering Path".

Compatibility notes

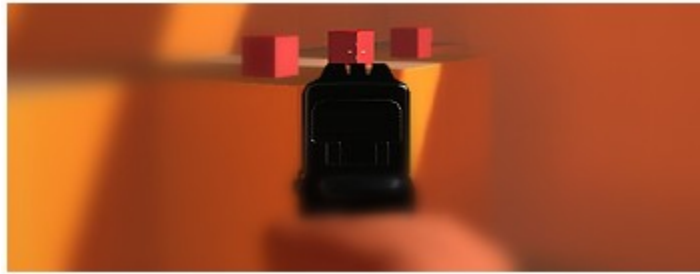
Crease

When attached to the WeaponCam this effect will only work on the weapon, not the scene.

DepthOfField

These effects will work best if attached to the FPCamera, though they will only affect the scene and not the weapon. If you wish to blur the weapon and close

surroundings but keep the target sharp (for example when sniping) consider instead using the "Vignette and Chromatic Aberration" effect.



Example of the *Vignette and Chromatic Aberration* effect using Blur

EdgeDetectEffectNormals

This image effect does not seem to work with UFPS. If you can find out why, please email me at info@visionpunk.com =)

GlobalFog

This effect will not work with Deferred Lighting, and must be assigned to the FPCamera. Hence it will only affect the scene and not the weapon.

SSAOEffect

This effect will not work with Deferred Lighting, and must be assigned to the FPCamera. Hence it will only affect the scene and not the weapon.

SunShafts

If attached to the WeaponCamera, the shafts will only be cast off of the weapon geometry, not the scene geometry (looks cool though).

Manipulating the FPS components from script

Creating a new script

If you're building an FPPlayer from scratch, you need to add a MonoBehaviour script to it in the Inspector. Next, you should create values in the script to easily access the camera and controller. Add the following variables at the top of your script:

```
vp_FPCamera m_Camera;  
vp_FPController m_Controller;
```

In the "Start" method of your MonoBehaviour, add the following lines:

```
m_Camera = gameObject.GetComponentInChildren<vp_FPCamera>();  
m_Controller = gameObject.GetComponent<vp_FPController>();
```

You can now access the components using the m_Camera and m_Controller variables.

Moving the Controller

To move around your controller using the keyboard, put the following snippet in the Update function of a MonoBehaviour attached to the vp_FPController:

```
if (Input.GetKey(KeyCode.W)) { m_Controller.MoveForward(); }  
if (Input.GetKey(KeyCode.S)) { m_Controller.MoveBack(); }  
if (Input.GetKey(KeyCode.A)) { m_Controller.MoveLeft(); }  
if (Input.GetKey(KeyCode.D)) { m_Controller.MoveRight(); }
```

The controller is rotated by mouselook by adding an FPCamera to its hierarchy as described in the above chapter "Getting Started -> Creating an FPPlayer from scratch".

Applying forces

The "external forces" feature set of vp_FPCamera is very powerful and can be used for lots of different scenarios. Forces are applied using the various "AddForce" methods, along with the "DoBomb", "DoStomp" and "DoEarthQuake" methods. See the comments in the vp_Camera class for more info.

Teleporting the player

The recommended way of forcing position and angle on an FPPlayer is this:

1. Temporarily add the following line somewhere in order to print the camera eulerangles to the console:

```
Debug.Log(m_Camera.transform.eulerAngles);
```

2. Start the app and position your player in the position & look angle you want.
3. Take a note of the player transform's position (from the Inspector) and its eulerangles (from the console).
4. In your teleport script, use the controller's "SetPosition" and camera's "SetAngle" method to teleport to that location and angle.

For example:

```
if (Input.GetKeyDown(KeyCode.T))  
{  
    m_Controller.SetPosition(new Vector3(-26.283f, 113.062f, -0.104f));  
    m_Camera.SetAngle(202.0f, 357);  
}
```

A word on inventory placeholder features

Ultimate FPS Camera - although feature rich - does not aspire to be a fully fledged game project. This means some typical game features are merely implemented as placeholders. Two such features are the "m_AmmoCount" parameter in vp_FPShooter, along with vp_FPPlayer's "AvailableWeapons" list. You will likely want to alter this functionality when hooking up your own inventory code (inventory implementation may differ quite a lot between games and may depend on whether it's an online or offline game, your choice of network solution and so on).

Use the 'SetWeaponAvailable' method to give or take weapons from the player, and check 'WeaponAvailable' to see if the available weapons list contains a certain weapon (if not, the script won't allow the player to wield it). All weapons are available by default. All weapons can be taken away by doing:

```
m_AvailableWeapons.Clear();
```

NOTE: The *m_AvailableWeapons* list is intended to contain all the weapons the player will ever be able to carry, and the game code should allow or prevent the player from activating them depending on the inventory state. It is not recommended to control weapon availability by adding or removing weapon gameobjects to the FP_Camera hierarchy dynamically.

Loading presets from script

Loading presets onto a component should normally be done in the Editor using the Preset and State foldouts, but can also be done via script using the "Load" helper methods like so:

```
m_Camera.Load(MyCameraPreset);  
m_Controller.Load(MyControllerPreset);
```

Where "MyCamera/ControllerPreset" should be TextAssets declared as public variables at the top of your script and assigned in the Editor:

```
public TextAsset MyCameraPreset = null;  
public TextAsset MyControllerPreset= null;
```

For more info on various ways of loading and assigning presets, see the comments in `vp_ComponentPreset.cs`.

Advanced: Event system

Introduction to the event system

The FP components of UFPS 1.4 communicate via a script called the *PlayerEventHandler*, which is a hub for all player scripts. This is not the *brain* of the player hierarchy (it has little or no game logic). Instead, it can be thought of as the *nervous system* of the player.

All scripts that make up the player hierarchy can talk to the event handler - and it can talk to them - but the event handler knows absolutely nothing about what's going on in the connected scripts, and the scripts know little or nothing about each other.

So what's the point of all this secrecy?

Modularity

Objects being unknown to each other makes for a *modular* design which is more resistant to brutal changes and is easier to maintain long term. In previous versions of Ultimate FPS Camera, taking out the weapon switching logic or the camera system would be an arduous process, requiring many surgical procedures before the game would even compile.

In v1.4 a big step has been taken towards complete future modularity (although lots of work remains). In this first version of the event system, difficulty has increased (a lot) for some scripting tasks, while other tasks have become easier.

As an example, DemoScene3, uses the Inventory system example script. This means the player has limited ammo and weapons. Now, if you disable or even delete the SimpleInventory component at runtime, the game will keep running - the player will just suddenly have unlimited ammo and weapons because it no longer understands the concept of "items being limited". The hooks for the inventory system still exist in the player event handler - *but no inventory is plugged in anymore*.

This makes it easy for you to write your own custom inventory system. As long as it talks properly to the event handler you can code it pretty much how you like, ideally without changing a single line of code in any other FP component (such as the weapon or shooter).

Performance

The event system is designed, above all, for speed. It utilizes core C# engine features for very fast runtime execution, and typically does most of its work on

startup, caching information about classes once for fast lookup should it ever need the info again.

Ofcourse, being the first release, this system is not yet battle tested. Use cases will be collected over the coming weeks and months. I have no doubt users will push the event system to its limits and reveal weaknesses that need be dealt with, simplified or optimized. See the release notes for current "Known Issues" pertaining to the event system.

Event handler tutorial

Let's say we want to create an event that prints a string to the console. We don't want the string to be printed by the calling object - instead it should be sent to a target "printer object" - via the event handler.

Preparation

1. Open the "CleanScene" example scene.
2. Right click in the Project view -> "Create" -> "C# Script" -> name it "Source".
3. Create a second script, this time naming it "Target".
4. Drag both scripts onto the "Camera&Controller" gameobject.

Now, to work with the event system you will typically engage in five different kinds of tasks:

1. Declaring an event

Open the vp_PlayerEventHandler.cs script and add a new event declaration:

```
public vp_Message<string> PrintSomething;
```

2. Making the PlayerEventHandler available to the scripts

Since our scripts are MonoBehaviours, we need to give them access to the event handler. This is done by adding the below code to both scripts. This will fetch the player event handler object located on the root transform on startup.

```
vp_FPPlayerEventHandler m_Player;

void Awake()
{
    m_Player = transform.GetComponent<vp_FPPlayerEventHandler>();
}
```


3. Registering a target script with the event handler

Our "Target" script will have some code to be run, and will need to register properly with the event handler (this is not needed for the sender). The following code will register and unregister the MonoBehaviour with the event handler whenever the component is enabled or disabled, whenever its gameobject is activated or deactivated, and whenever the gameobject is destroyed. Paste the below code into the "Target" script.

```
protected virtual void OnEnable()
{
    if (m_Player != null)
        m_Player.Register(this);
}

protected virtual void OnDisable()
{
    if (m_Player != null)
        m_Player.Unregister(this);
}
```

4. Adding an event listener

Now it's time to add the actual implementation of the printing method. Add the following code to the "Target" script. This is what will be executed when the event is sent.

```
void OnMessage_PrintSomething(string s)
{
    Print(s);
}
```

5. Sending an event

In the "Update" method of the "Source" script, add the following code:

```
if (Input.GetKeyUp(KeyCode.Return))
{
    m_Player.PrintSomething.Send("Hello World!");
}
```

Now, start the game and press Enter. This should result in the string "Hello World" being printed to the console.

NOTES:

- a. Notice that the above code doesn't do just "PrintSomething("Hello World")". Instead it uses the **".Send"** before the parenthesis. This is because

*"PrintSomething" is not a method - it is an event of type "vp_Message", which is always invoked by its "Send" method. **Remember this because it can be confusing at times, especially since different types of events have different invocation syntax.***

- b. In case our scripts had been derived from vp_Component, steps 2-3 above would not have been necessary, since the base class would have taken care of them.*

Event types

There are four general types of event with very varying scripting difficulty level. All types support a single generic parameter (or none). Some event types support generic return values. Some support an unlimited amount of registered methods. Others don't.

TIPS:

- a. To learn more about the various event types, open the vp_PlayerEventHandler script and mouse-over on the event types (e.g. vp_Message) for usage info.*
- b. To find the places where an event is **SENT**, you can do 'Find All References' on the event in your IDE. if this is not available, you can search the project for the event name preceded by '.' For example: **.Reload***
- c. To find the methods that **LISTEN** to an event, search the project for its name preceded by '_'. For example: **_Reload***

vp_Message

Listener prefix: **OnMessage_**

Invoker: **Send**

Multiple listeners per event: **Yes**

Return value: **any type (optional)**

The standard event type. Takes 0-1 generic arguments and optionally a generic return value. Messages are used for exposing script methods, allowing all scripts that are registered to the event handler to call them. The method name in the target script must have the prefix 'OnMessage_'. Call 'Send' on the event to invoke the method.

An unlimited number of callback methods with the 'OnMessage_' prefix can be added to the same event. For example: there might be 100 methods in 100 different scripts all added to the 'BombShake' event. Now, if a single script calls 'Player.BombShake.Send(1.0f)' every one of those methods will trigger with the argument '1.0f'. a method in a registered script will automatically be added to an event if its name has the prefix 'OnMessage_', followed by the event name. For example, search the project source code for 'OnMessage_BombShake'.

vp_Attempt

Listener prefix: **OnAttempt_**

Invoker: **Try**
Multiple listeners per event: **No**
Return value: **bool**

An *attempt* is a simpler form of a message, which supports 0-1 arguments of any type and always returns a bool. It is called using the word 'Try' instead of 'Send'. Its intended usage is for situations when gameplay code needs to ask permission, typically for player actions that may succeed or fail. This event type only supports a single listener (only one method in one script can be hooked up to an attempt).

For example: 'Player.SetWeaponByName.Try("Axe")' can be called to try and switch weapons. If the registered method doesn't think the player has an axe, it should return false. Otherwise, it should switch to the axe and return true. A method in a registered script will automatically be added to an event if its name has the prefix 'OnAttempt_', followed by the event name.

vp_Value

Listener prefix: **OnValue_**
Invokers: **Get, Set**
Multiple listeners per event: **No**
Return value: **any type (for 'Get')**

This event type exposes a *C# property* of a single registered script. The generic invocation fields, 'Get' and 'Set', expose their property counterparts. The property name in the target script must have the prefix 'OnValue_'. Use 'Get' and 'Set' to retrieve or assign the property via the event.

Values work basically just like the C# properties they expose. Either the "get" or the "set" accessor can be left out. A property without a set accessor is considered read-only. A property without a get accessor is considered write-only. A property with both accessors is read-write. A listener for a vp_Value looks a little bit different from a vp_Message or a vp_Attempt:

```
Vector2 InputMoveVector;  
Vector2 OnValue_InputMoveVector  
{  
    get  
    {  
        return m_MoveVector;  
    }  
    set  
    {  
        m_MoveVector = value;  
    }  
}
```

vp_Activity

Listener prefixes: **CanStart_, CanStop_, OnStart_, OnStop_**

Invokers: **TryStart, TryStop, Active, Start, Stop**
Multiple listeners per event: **Yes**
Return value: **bool (for 'TryStart' & 'TryStop')**

vp_Activity is a very advanced event type encapsulating a set of common activity related game logics such as event duration, trigger intervals, scheduled disabling, and most importantly conditional enabling and disabling of activities. It is quite hard to grasp actually, but it is the main workhorse in UFPS input logic.

An activity may be **active** or **inactive**, and may trigger callbacks when it starts or stops. The activity is typically started or stopped by calling its 'TryStart' and 'TryStop' methods, in which case a list of conditions will determine whether the call succeeds. The activity may also be forced on or off using the 'Active' property or by calling the 'Start' and 'Stop' methods. An unlimited number of conditions and callback methods can be added to the start and stop delegates.

The method names in the target script are categorized into 'conditions' and 'callbacks'. When a 'TryStart' method is called, an unlimited amount of registered conditions for the event may trigger. All scripts that have registered a condition for the activity now have a say in whether the activity should be able to start. If a single script returns false, the activity will fail to start and return false, otherwise true. The same goes for the TryStop command.

The event optionally supports a single generic argument which can be accessed over the event duration via the 'Argument' property. To learn more about activities in action, it is best to study the events declared in the vp_FPPlayerEventHandler.

Activity additional properties

MinPause

Prevents player from restarting an activity too soon after stopping

MinDuration

Prevents player from aborting an activity too soon after starting

AutoDuration

Automatically stops an activity after a set timespan

Argument

Returns the user-passed argument of this activity,

Active

When set, starts or stops an activity and fires its start or stop callbacks (if any).

Start

Starts an activity and fires its start callbacks (if any).

Stop

Stops an activity and fires its stop callbacks (if any).

Disallow

Disallows an activity for 'duration' seconds.

Binding states to Activities

The player event handler also has a feature which can *bind activities to component states*. This is done using the method "BindStateToActivity", which can activate or deactivate a state when an activity with the same name is toggled. This is used quite heavily for the example "AdvancedPlayer" prefab, in conjunction with several component *state blocking rules*.

Activity example flows

Crouch

1. the player presses the crouch button and climbs under a desk. the Crouch activity's TryStart method returns true and the Crouch activity is set to ON.
2. the Crouch activity triggers crouching states on the camera and controller. the camera's Crouch state makes it go lower. the controller's Crouch state makes it slow down.
3. the player tries to stand up again, but the TryStop method of the Crouch activity returns false (we hit our head on the underside of the desk) and the character keeps crouching even though the crouch button has been released.

Jump

1. on startup the controller adds a condition to the Jump activity, stating that this activity may only start if the player is standing on the ground (according to the controller)
2. the controller also adds a callback to the Jump activity, giving the player an upward force when the activity starts
3. the input object will later try to start the activity, by invoking the 'Jump.TryStart' method when the jump button is pressed

Attack

1. on startup the inventory adds a condition to the Attack activity, stating that this activity may only start if there is a wielded weapon with enough ammo (according to the inventory)
2. on startup the weaponhandler also adds a condition to the Attack activity, stating that this activity may only start if the player is not currently reloading
3. etc.

What's the difference between an Activity and a State?

- **Activities** determine **what the player is doing** - such as running or swimming.
- **States** describe **what components are like** - such as being fast, elevated or zoomed in.
- Also, **states are not events**. They are more like component snapshots, save files.
- You can think of an **activity** as a **verb** - it explains what the player is currently up to.
- You can think of a state as an **adjective** - it describes the attributes of a component.

Support and additional information

Again, if you have any further questions or just want to show off your FPS work, don't hesitate to participate in the Unity Community forum discussion:

<http://forum.unity3d.com/threads/126886-Ultimate-FPS-Camera-RELEASED?goto=newpost>

... or fire off an email to:

info@visionpunk.com

Thanks for buying this asset and Good Luck! =)

/Cal