CYBERITH SDK

Unreal Engine – Integration Guideline

Cyberith GmbH
Teslastraße 6
3100 St. Pölten
Austria
FN 410899p

For any questions, please contact
Cyberith Support
support@cyberith.com
+43 1 890 17 13

Table of Contents

Prerequisites 4

- Window 7 or newer 4
- Unreal Game Engine 4
- For projects containing C++ Code: Visual Studio 4

Getting Started with the Cyberith SDK in Unreal Engine 5

Step 1 - Setup Example Project 6

1. Create new empty project 6
2. Import the CybSDK Unreal Engine Plugin 9
3. Regenerate your Visual Studio Solution (C++ Projects ONLY) 11
4. Activate Plugin and Open Example Map 12
5. Start your Virtualizer Experience 18

Step 2: Setup Your Own Project – common to all options 19

1. Navigate to your project's root folder 19
2. Import the CybSDK Plugin by extracting the provided folder in your project's root folder 19
3. If your project is a C++ project: Regenerate your Visual Studio Project Files by right-clicking on the ".uproject" file of your own project (not required for Blueprint projects) 19
4. Open your own project in Unreal Engine 19

Step 2a: Setup Your Own Project – Option 1: Using the BP_VirtPlayerController (like in SampleA) 20

Step 2b: Setup Your Own Project – Option 2: Creating/Modifying your own C++ Player Controller (only for C++ projects) 22

Step 2c: Setup Your Own Project – Option 3: Creating/Modifying your own Blueprint Player Controller 23

- You can start by creating a Blueprint variable of the type "Virtualizer Device" 23
- Setup a node logic 24

No Virtualizer Hardware? – No Problem! 31

Project Settings Documentation 32

Virtualizer 32

Blueprint Settings Documentation 33

BP_VirtPawn 33

BP_VirtGameMode 34

BP_VirtPlayerController 34
BP_VirtHapticListenerComponent 35
BP_VirtHapticEmitterComponent 36
Specific Haptic Emitter Scripts 38
Compatibility with Standard Functions of Unreal Engine 39
Gamepad Axis Emulation 39
Force Feedback Input 40
Blueprints 41
UVirt 41
GetSDKVersion 41
GetPluginManagedDevice 41
Sample – Haptic Emitter 41
SDK Documentation 42
C++ SDK Documentation 42
FCybSDK_PluginModule 42
GetVirtualizerInputDevice 42
FVirtInputDevice 42
GetDevice 42
IsDecoupled 42
UVirtDevice 43
GetMovementVector 43
GetMovementDirectionVector 43
GetPlayerOrientationVector 43
GetPlayerOrientationQuaternion 43
UVirtHapticEmitterComponent 43
Play 43
Stop 43
C++ Example usage 44
Locomotion 44

Prerequisites

- Window 7 or newer

- Unreal Game Engine

Compatible: Unreal 4.20 or newer (incl. Unreal Engine 5)
https://www.unrealengine.com

- For projects containing C++ Code: Visual Studio

This is a requirement of Unreal Engine for using it with C++ code.
Blueprint-only projects work without Visual Studio as well.

Compatible: Visual Studio Community 2017 or newer (incl. 2019 & 2022)

https://www.visualstudio.com/en-us/downloads/download-visual-studio-vs.aspx

Select “Desktop development with C++” in the Visual Studio installer.

Getting Started with the Cyberith SDK in Unreal Engine

This instruction document is covering **both Unreal Engine 4 and Unreal Engine 5**. While menus may look different in different versions of Unreal Engine, **the concept and the required steps to implement the Virtualizer remain unchanged!**

To get started using the Virtualizer in Unreal Engine, we recommend you to first check the Example Maps we deliver with our Unreal Engine SDK Package.
To do so you can either use a **Blueprint** project or a **C++** project.

Once you have verified the functionality of the Virtualizer using our Example Map, you can integrate the Virtualizer in your own project.

Thus, we suggest to follow these steps:

Step 1: Setup Example Project
- Option A: Use a Blueprint Template
- Option B: Use a C++ Template

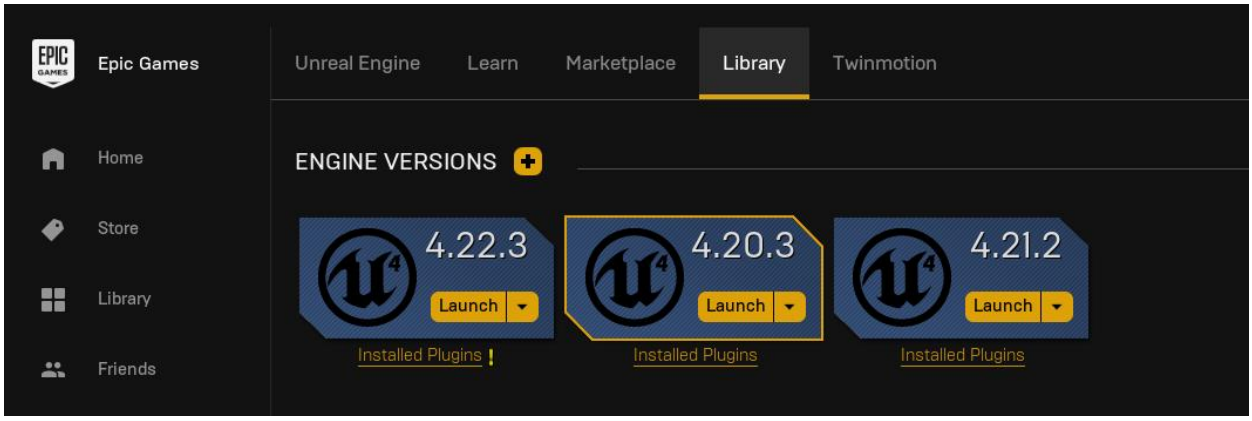
Step 2: Setup your Own Project
- Option 1: Using the provided BP_VirtPlayer Controller
- Option 2: Creating/Modifying your own C++ Player Controller
- Option 3: Creating/Modifying your own Blueprint Player Controller

Step 1 - Setup Example Project

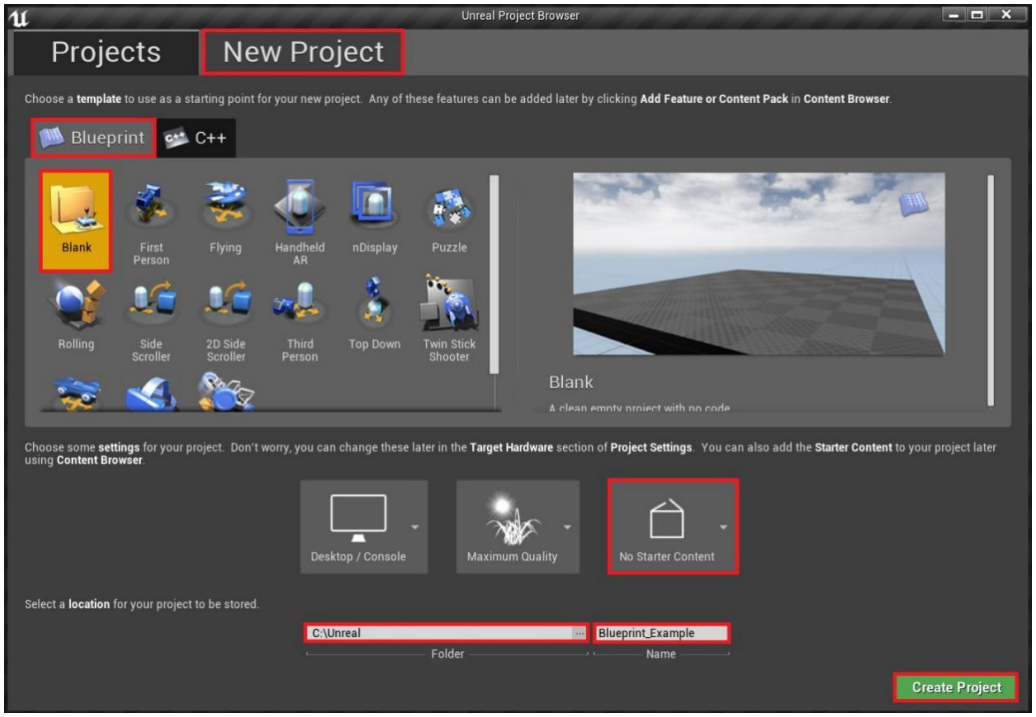
Setting up the Cyberith Virtualizer SDK and the Example Maps in an Unreal Engine (4 or 5) Blueprint or C++ project is done by following steps:

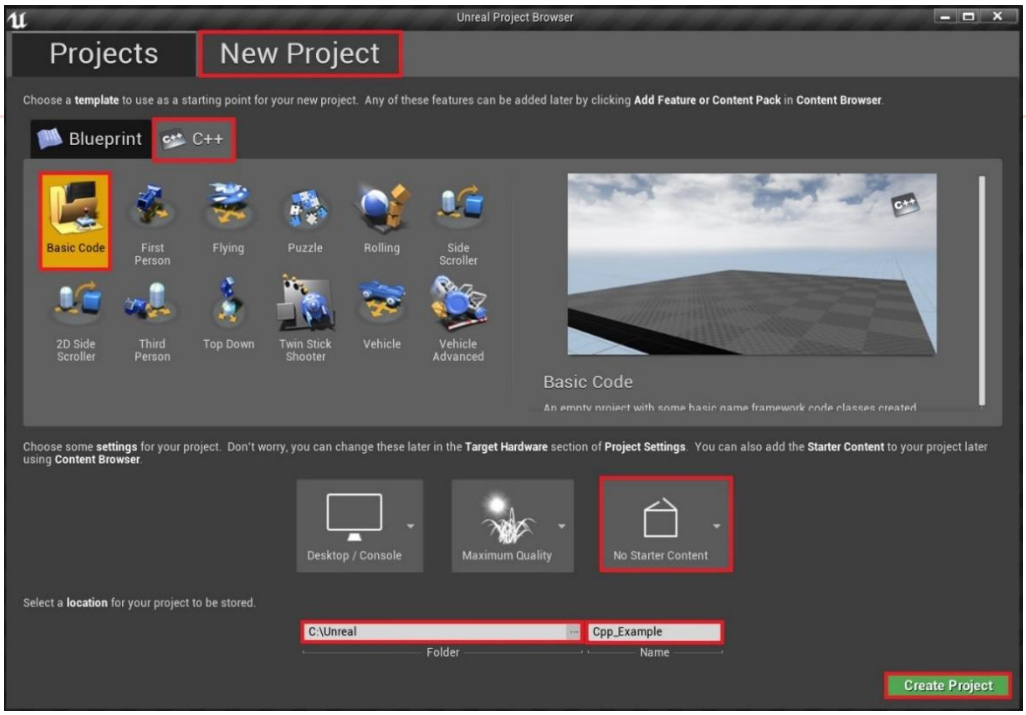
1. Create new empty project

Start the Epic Launcher, click on Library and launch the installed Engine version you want to work with.



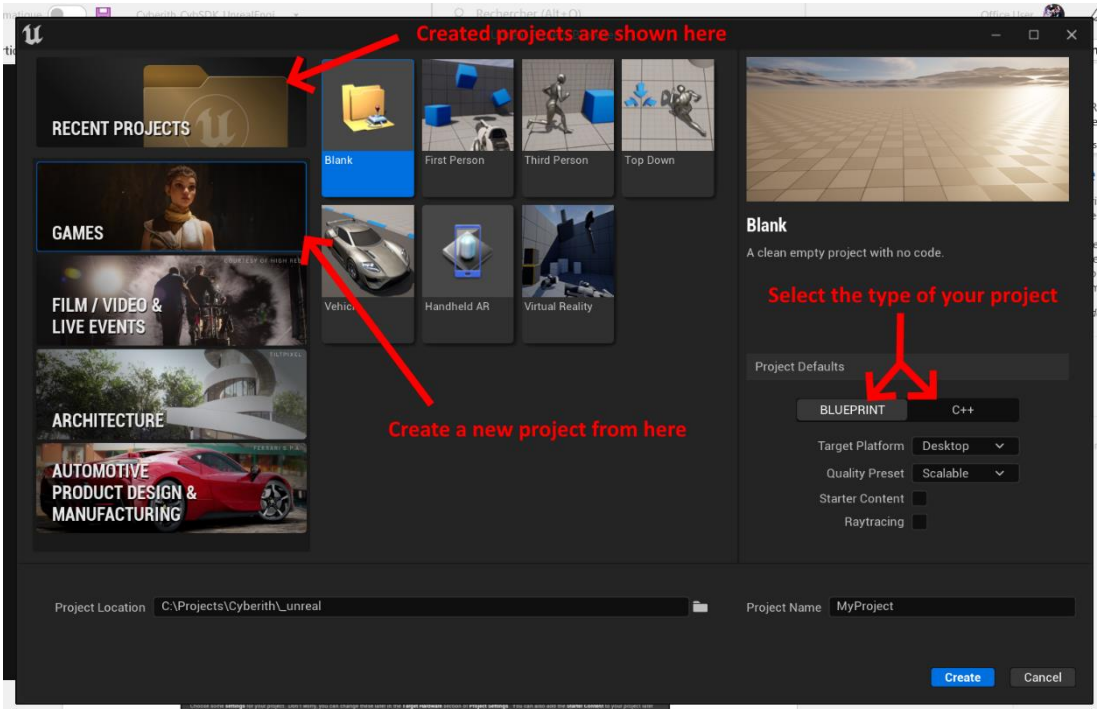
For **Unreal Engine 4**, create a New Project by selecting the Tab "New Project", then choose "**Blueprint**" (**Option A**) or "**C++**" (**Option B**) and "Blank". We suggest to select "No Starter Content" (it also works with Starter Content).



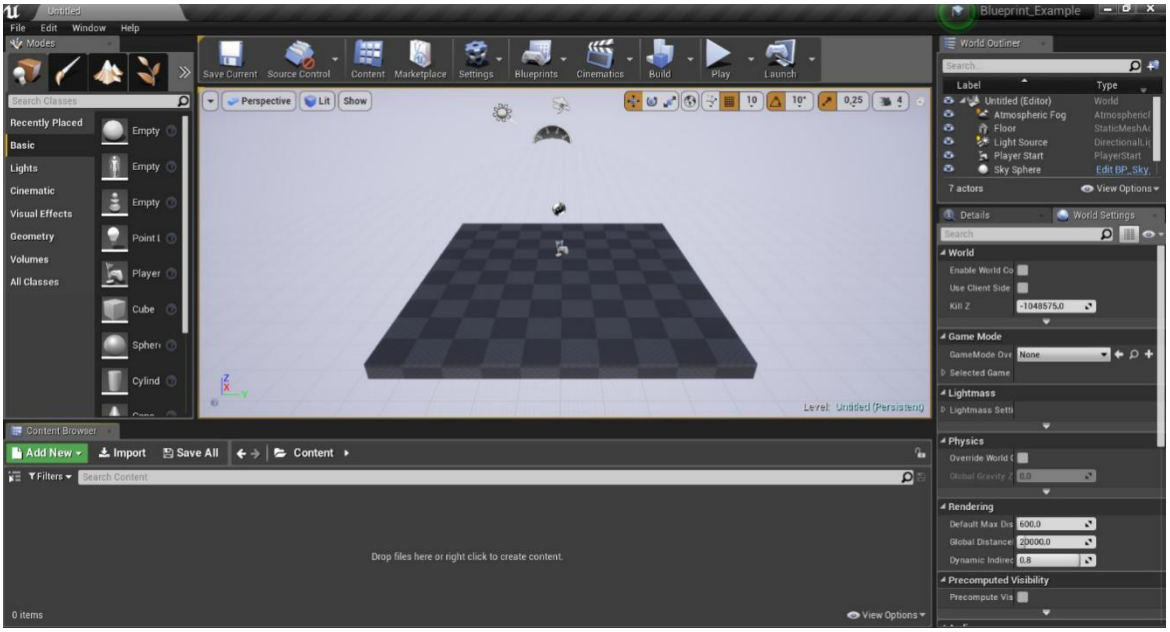


Choose a project folder and a project name before clicking “Create Project”. Make sure to remember the selected project folder and name (rootDirectory), as this will be required in the following steps. (The default installation path is “C:\Users*YourWindowsUserName*\Documents\UnrealProjects”.)

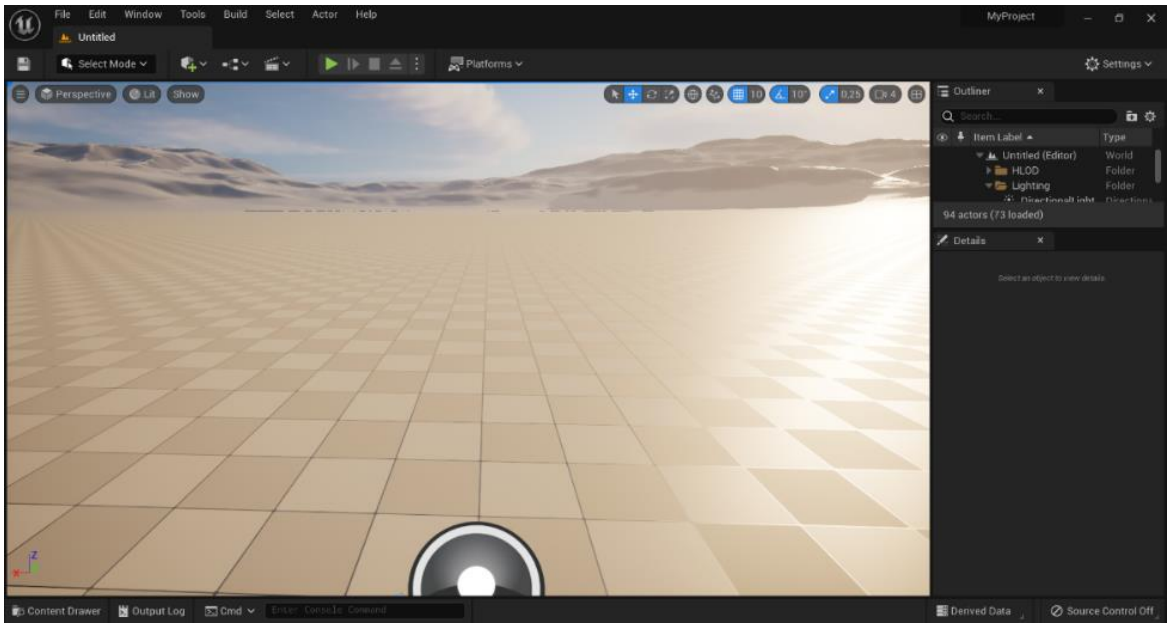
To create a new project in **Unreal Engine 5**, you need click “Games” and then select “Blueprint” on the right. Select the same settings as described above for UE 4.



You'll have created an empty project now:



Screenshot taken in UE 4

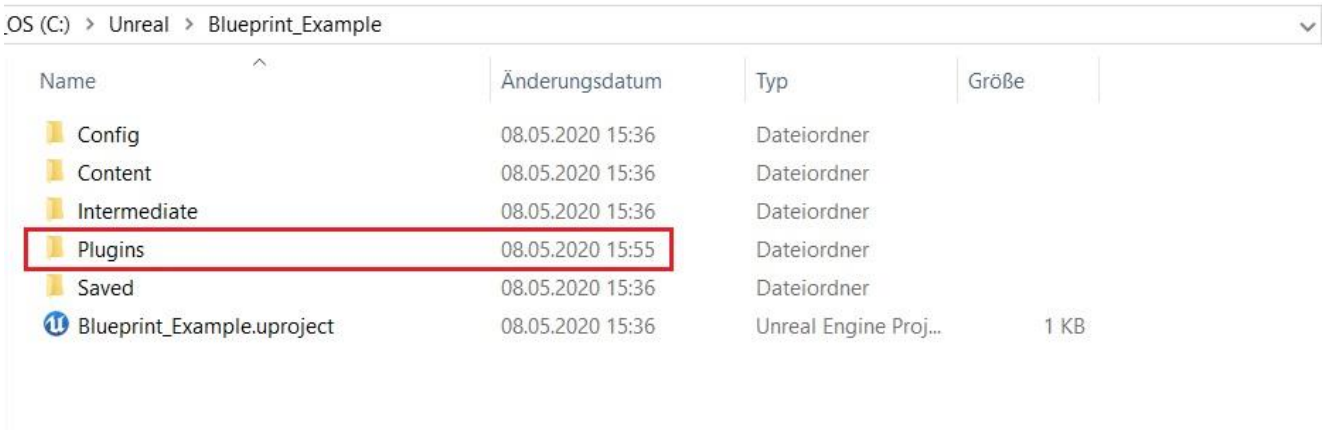


Screenshot taken in UE 5

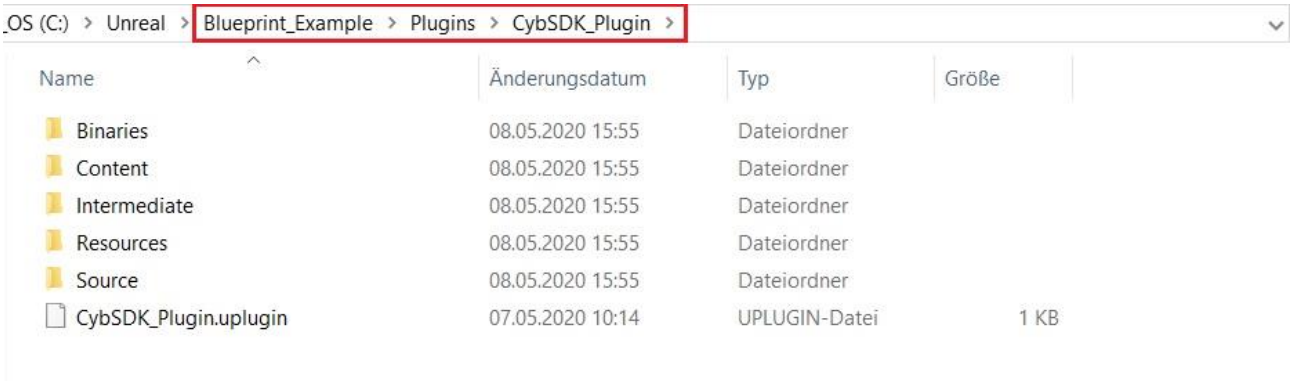
2. Import the CybSDK Unreal Engine Plugin

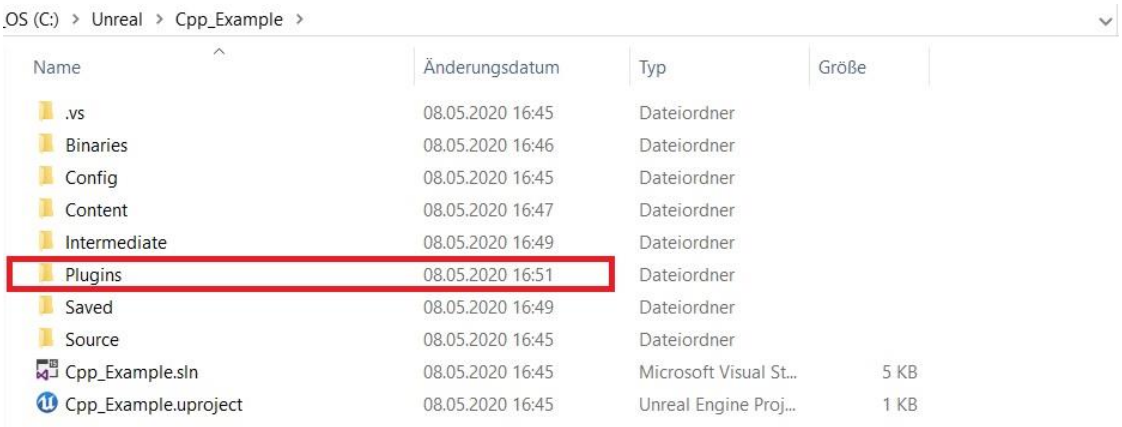
Cyberith Virtualizer SDK is distributed as a custom Unreal Engine Plugin. Follow these steps to implement it:

1. Close Unreal Engine.
2. Extract the CybSDK Plugin folder you downloaded from our developer webpage into the project:
    - Unzip the provided .zip folder
    - Copy the content of the extracted folder into the rootDirectory of your Unreal Engine project
    - The rootDirectory is the project folder + name created in step 1.
    - The exact location of the SDK path must be: *rootDirectory/*Plugins/CybSDK_Plugin
    - Example path: C:\Unreal\Blueprint_Example\Plugins\CybSDK_Plugin (Installation Folder: C:\Unreal; Project Name: "Blueprint_Example")

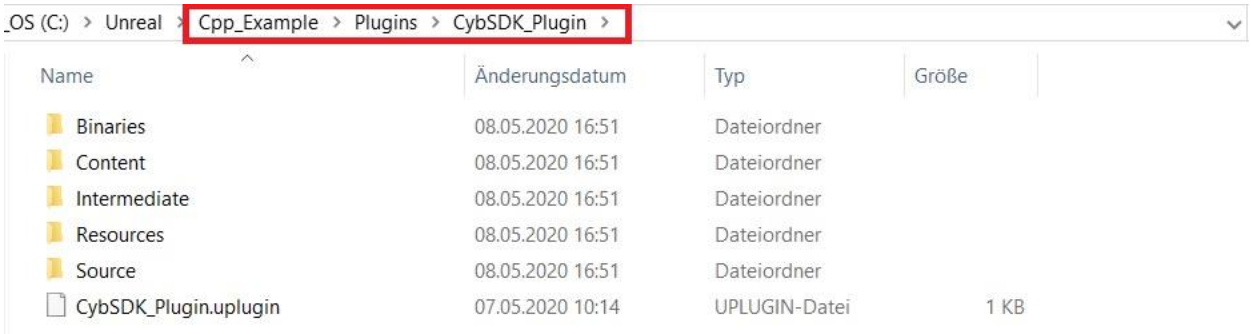


The above image shows a blank Blueprint project after the CybSDK Plugin was added.





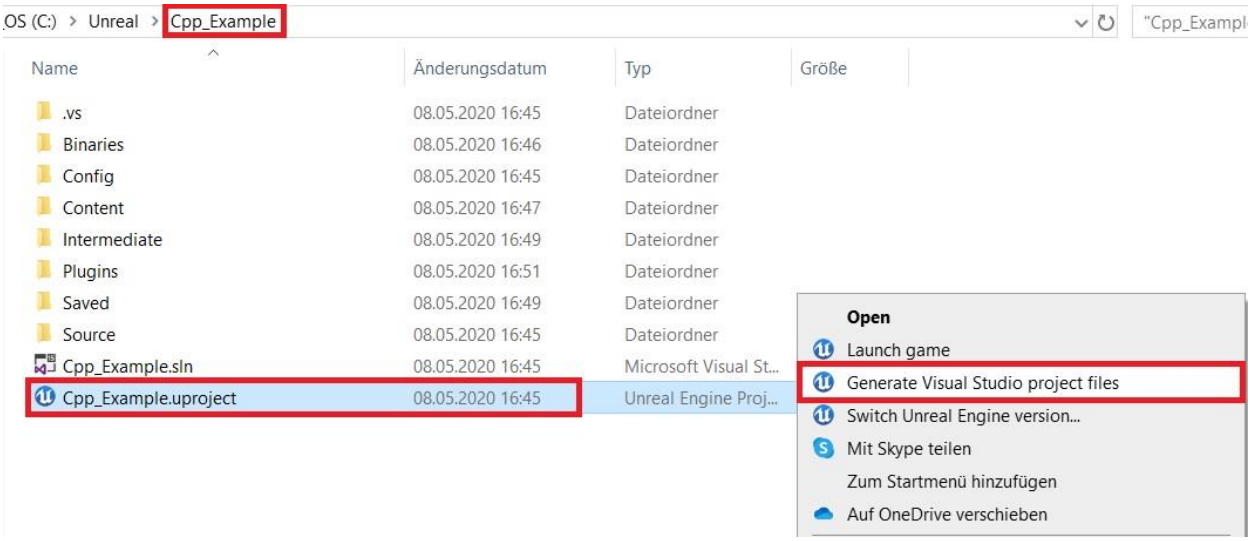
The above image shows a blank C++ project after the CybSDK Plugin was added. Note the ".sln" Visual Studio Solution file.



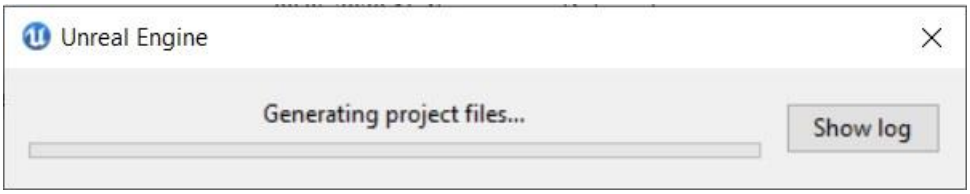
3. Regenerate your Visual Studio Solution (C++ Projects ONLY)

This step is only required for C++ Projects (OptionB as selected in step 1, as laid out above). ***Skip this step, if you are using a Blueprint project*** (OptionA).

Right-click the .uproject file in your rootDirectory. In the pop-up menu, click “Generate Visual Studio project files”.



This step is required for Visual Studio and Unreal Engine to recognize the newly added C++ code.
After your project files have been generated (windows below), there is no specific confirmation of success.

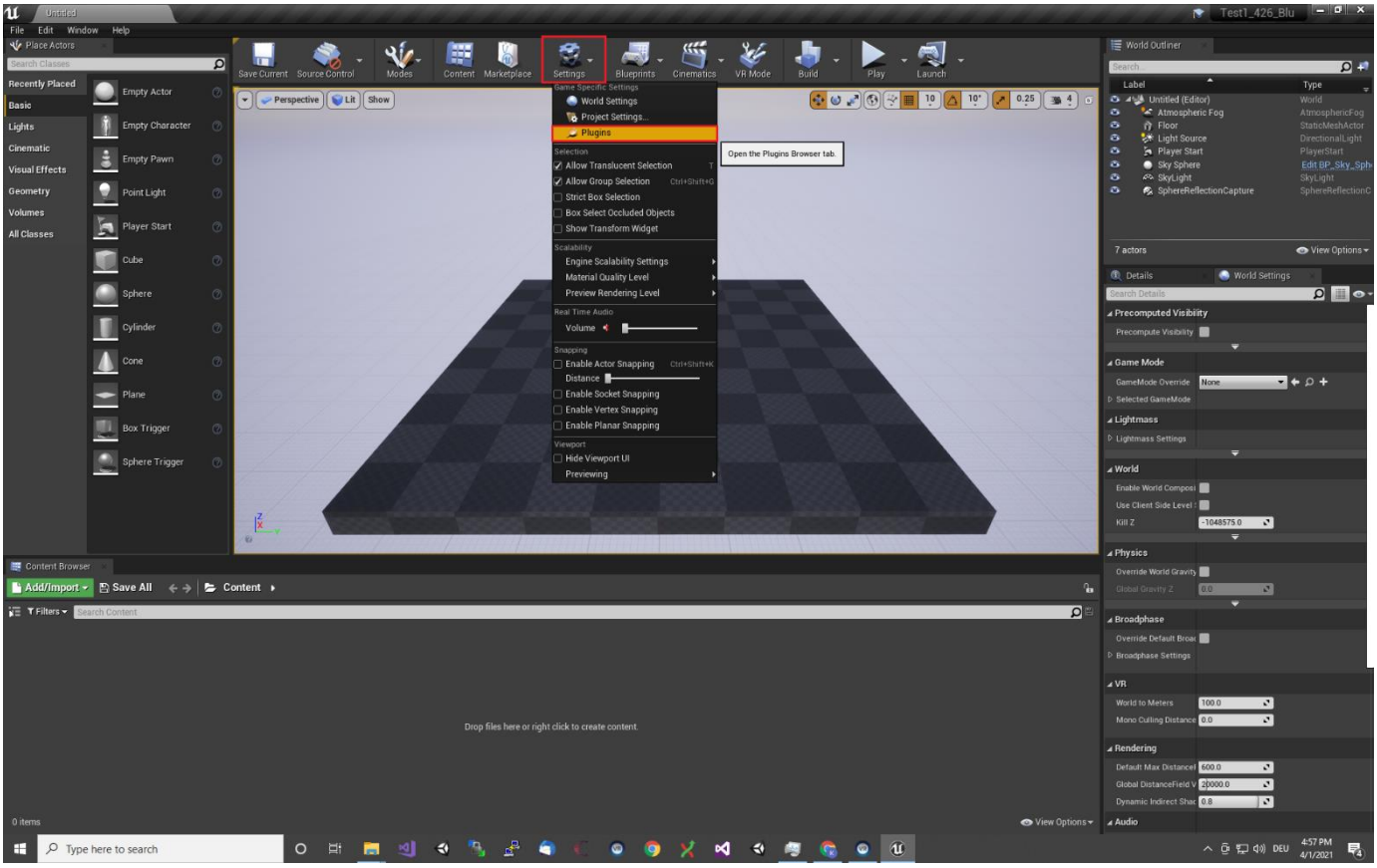


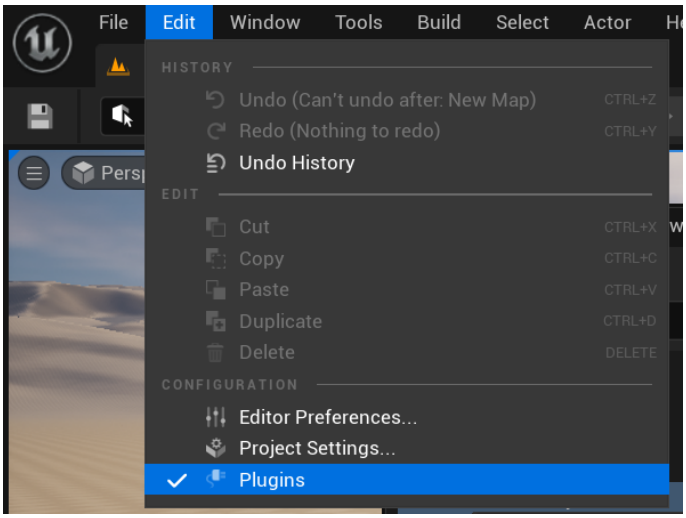
As long as there is no Error Massage, everything is normal.

4. Activate Plugin and Open Example Map

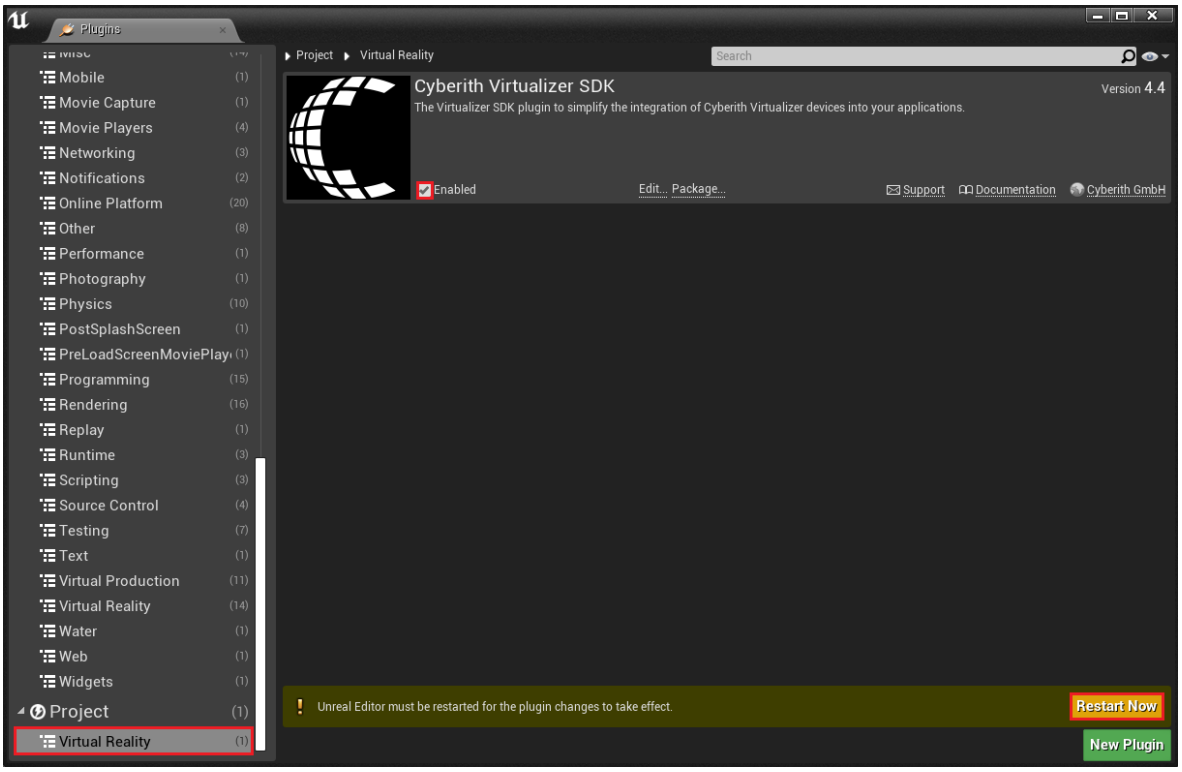
The CybSDK comes with a prebuilt example scene to demonstrate the packages capabilities:

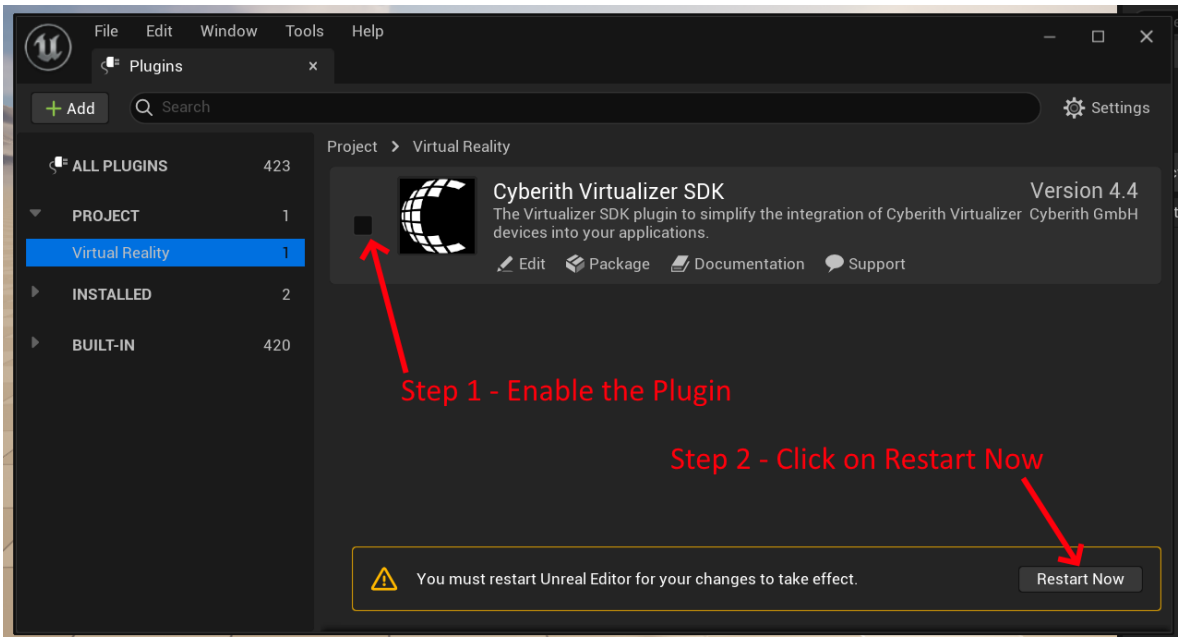
- Reopen your project. Then, click on “Settings” and “Plugins” to open the Plugins window in UE 4. In UE 5, you need to click “Edit” and “Plugins” to do the same.



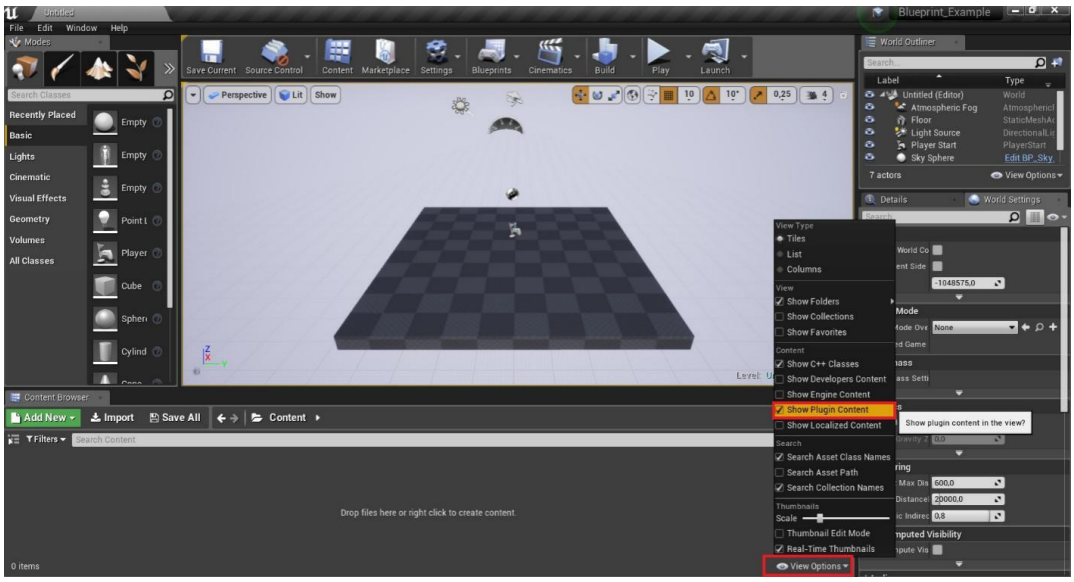


- In the Plugins window, search for the “Project” Plugins. In the section “Virtual Reality”, you will find the Cyberith Virtualizer SDK.
  Check the checkbox to enable it. Then, a pop-up massage will appear telling you that a restart is required. Click “Restart now”.

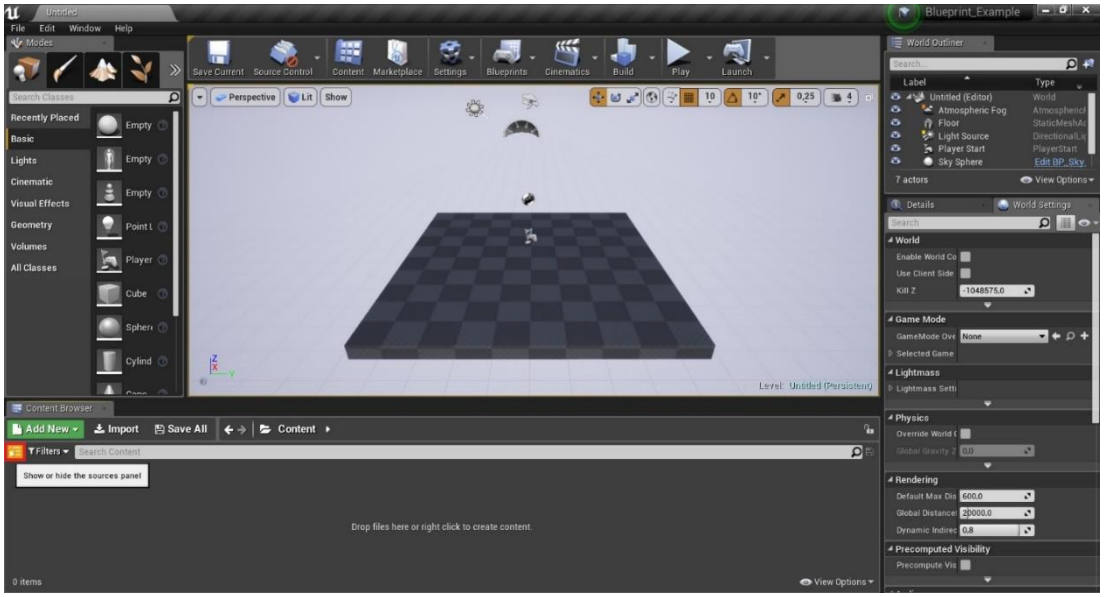




- After the restart, close the Plugins window and add your Plugin content to the content browser: In UE 4:
    - Click on View Options in the bottom of the Unreal Editor
    - Make sure the checkbox “Show Plugin Content” is checked

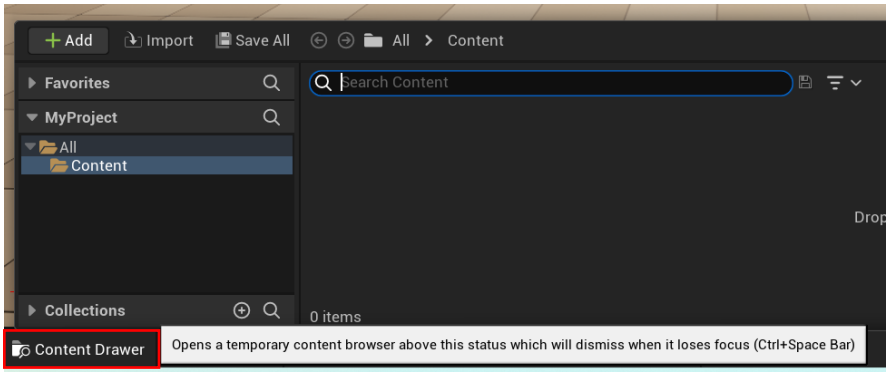


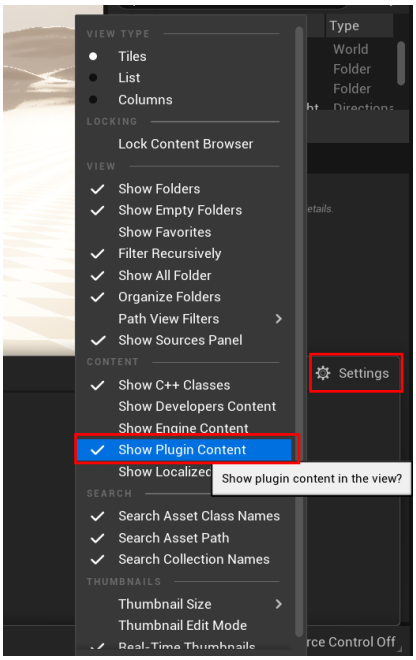
    - Then, click the highlighted symbol in the bottom left of your Unreal Engine 4 Editor



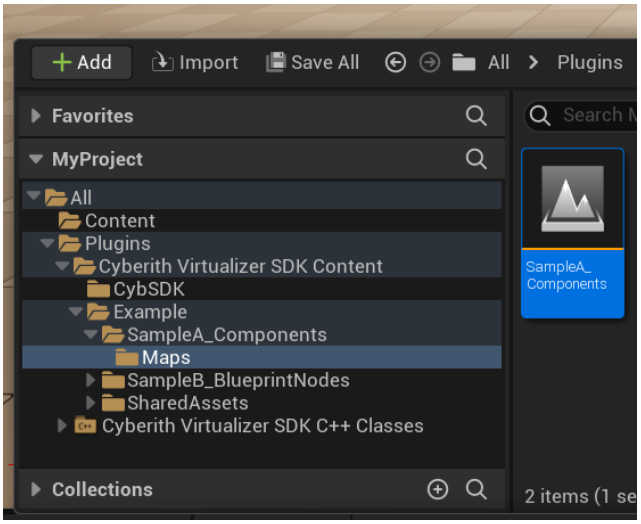
In UE 5:

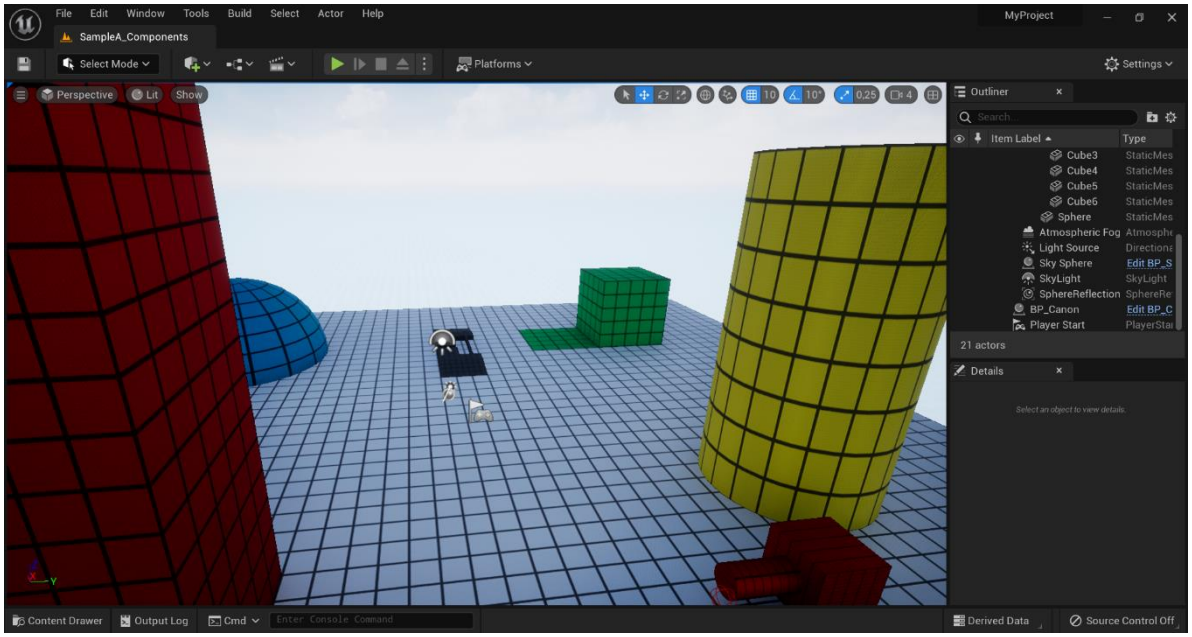
- Open your “Content Drawler”, by clicking on the according button in the bottom left
- Click “Settings” in the top right of the “Content Drawler” window
- Make sure the checkbox “Show Plugin Content” is checked





- Open “Cyberith Virtualizer SDK Content”, click “Example”, choose “**Sample A**_Components”, then “Maps” and open the map “SampleA_Components”. This map is pre-configured, you should be able to start it and get a first impression of the Cyberith Virtualizer SDK.



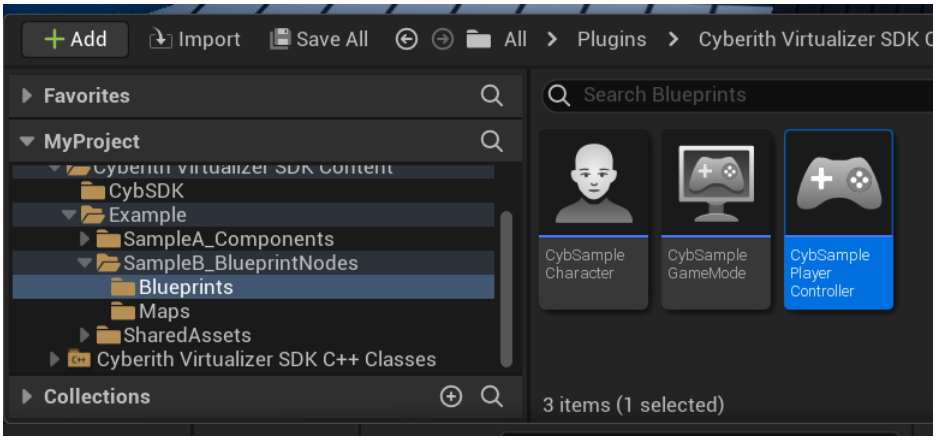


Sample A highlights our custom PlayerController called “BP_VirtPlayerController” which should handle everything on its own.

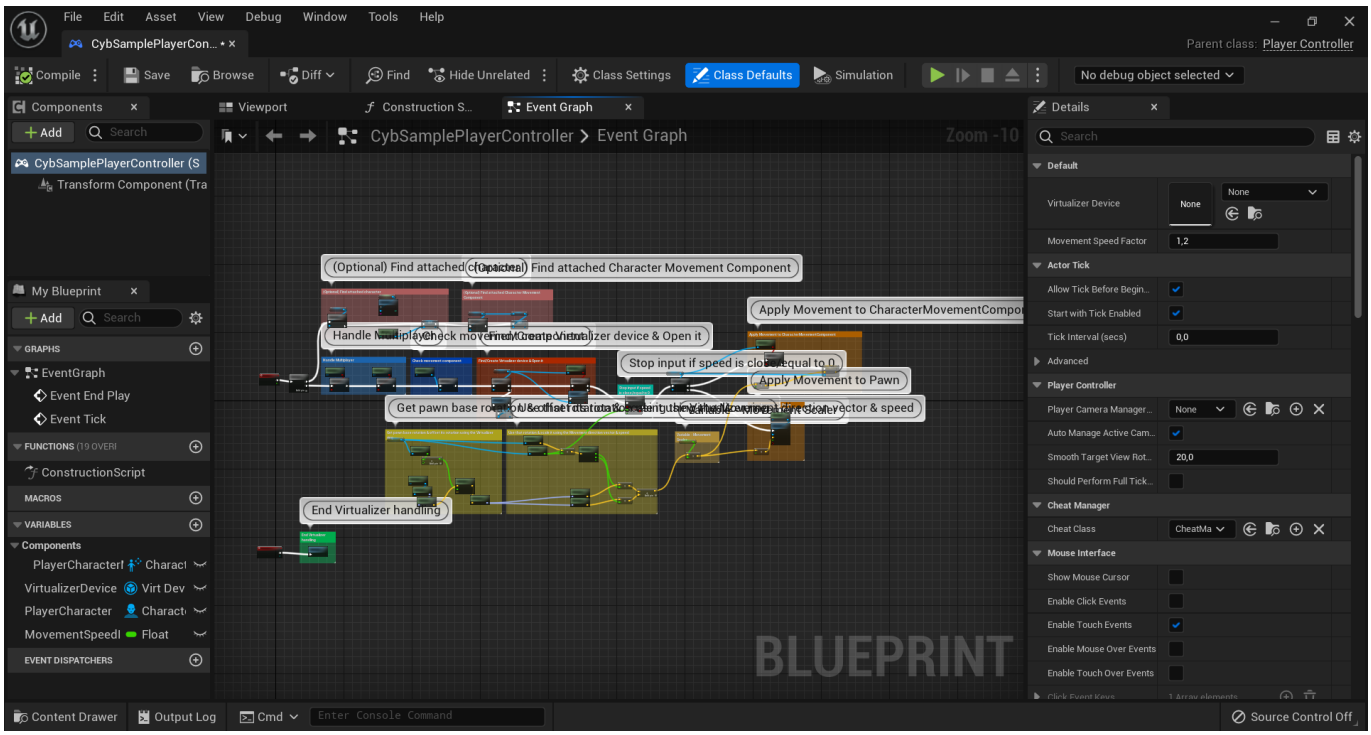
The Player Controller in Sample A should serve as an easy start for having VR movement implemented right out of the box.

- Alternatively, the other option is “SampleB_BlueprintNodes”. It showcases the movement behaviour demonstrated with the blueprint system (visual programming) of Unreal Engine.

You can find the corresponding map in the “Maps” subfolder, similarly to Sample A.
However, in SampleB, you can see how we did set up movement controlled by the Virtualizer using Blueprint by checking the **CybSamplePlayerController** blueprint.



Double-click on CybSamplePlayerController to visualize its visual programming logic.



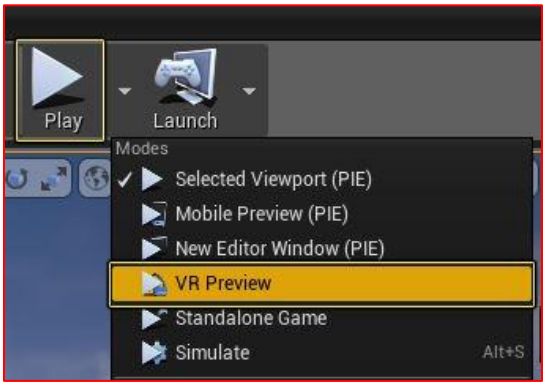
SampleB should help you in case you need to implement the Virtualizer into your own custom Player Controller via Blueprint. In case there is no such need, we strongly recommend to use the Player Controller provided by us, like in SampleA.

5. Start your Virtualizer Experience

Check if your Virtualizer's USB is plugged.

If it is plugged, you can run the Example Maps (SampleA and also SampleB) and test their functionality with your Virtualizer.

Please select the **VR Preview** launch option in the drop-down-menu and have fun.



Please check two points:

- The Locomotion: Walk on your Virtualizer to see your character moving accordingly. This is implemented in both SampleA & SampleB.
- The Haptic (for SampleA): Walk to the green elevator. When the elevator moves up or down, you should feel Vibrations coming from the Virtualizer baseplate. This is implemented in SampleA only!

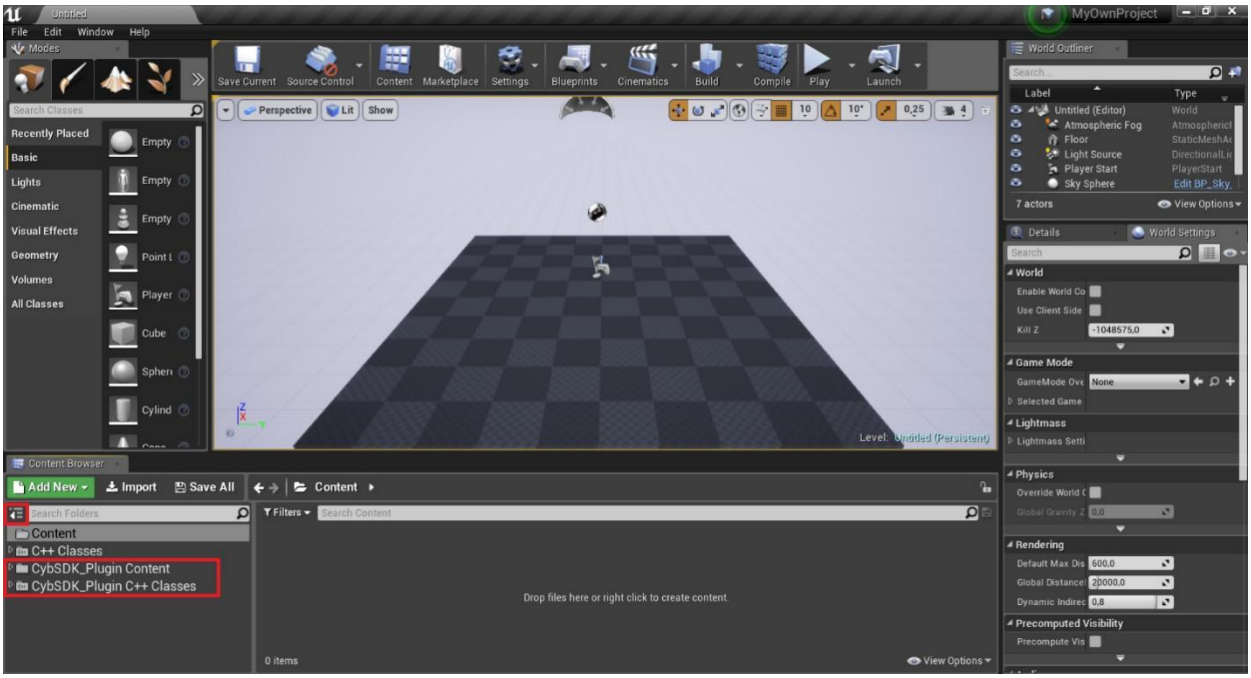
If you run into any problems with the steps above feel free to contact us: support@cyberith.com

Step 2: Setup Your Own Project – common to all options

Setting up the CybSDK Plugin with your own project, works equivalently to setting it up with the example maps as laid out in Step 1:

1. Navigate to your project's root folder
2. Import the CybSDK Plugin by extracting the provided folder in your project's root folder
3. If your project is a C++ project:
   Regenerate your Visual Studio Project Files by right-clicking on the ".uproject" file of your own project (not required for Blueprint projects)
4. Open your own project in Unreal Engine

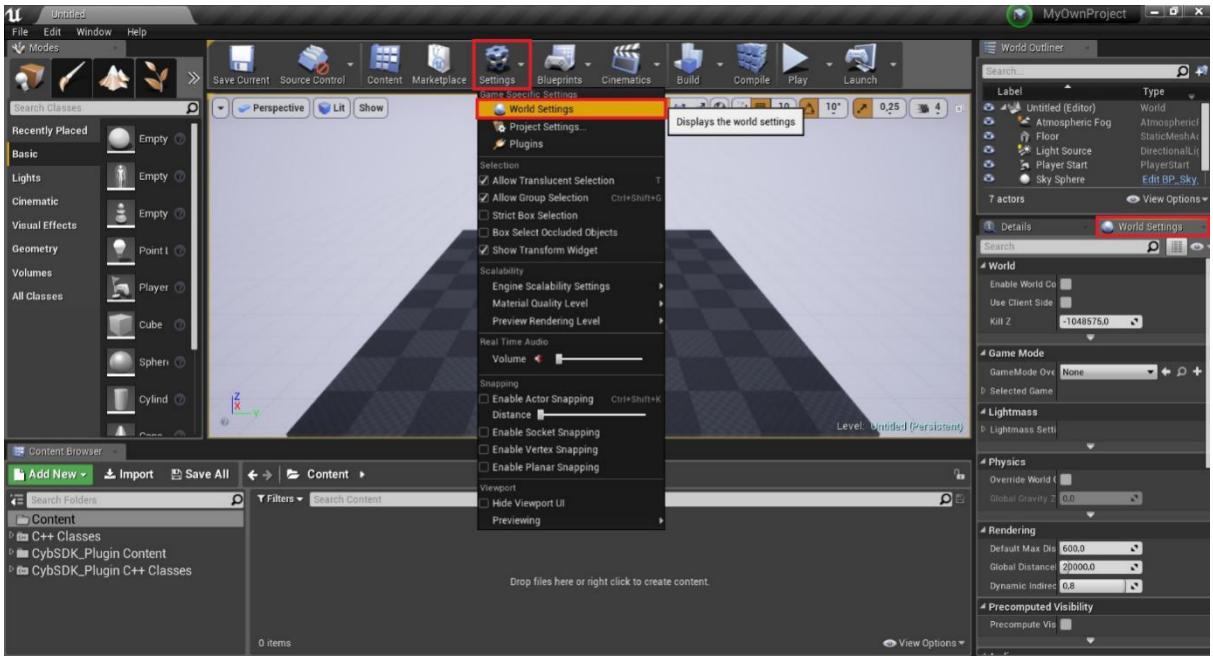
After repeating the instructions of Step 1 for your own project, the CybSDK should be visible in your own project.



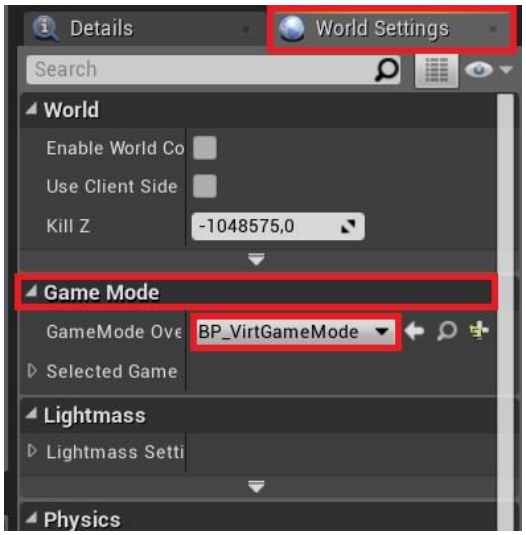
Step 2a: Setup Your Own Project – Option 1: Using the BP_VirtPlayerController (like in SampleA)

To activate the Virtualizer in your own project using the BP_VirtPlayerController (like in SampleA), one more step is required:

- Open the World Settings:
  - In UE 4 you need to click on *Settings/World Settings*
  - In UE 5 you need to click on *Window/World Settings*



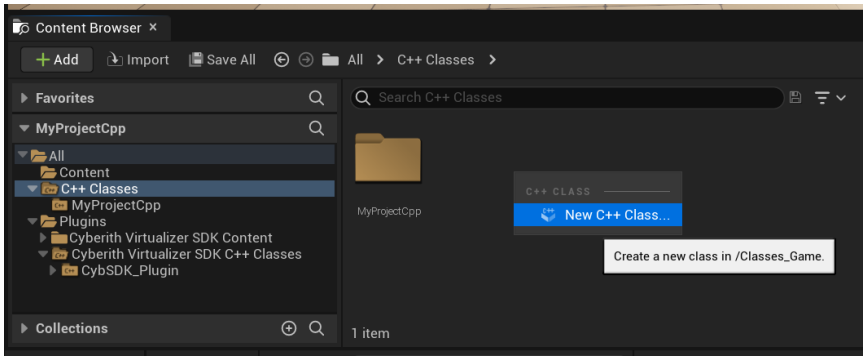
- In the World Settings menu, select the game mode “BP_VirtGameMode” from the drop-down menu “GameMode Overide” in the section “GameMode”:

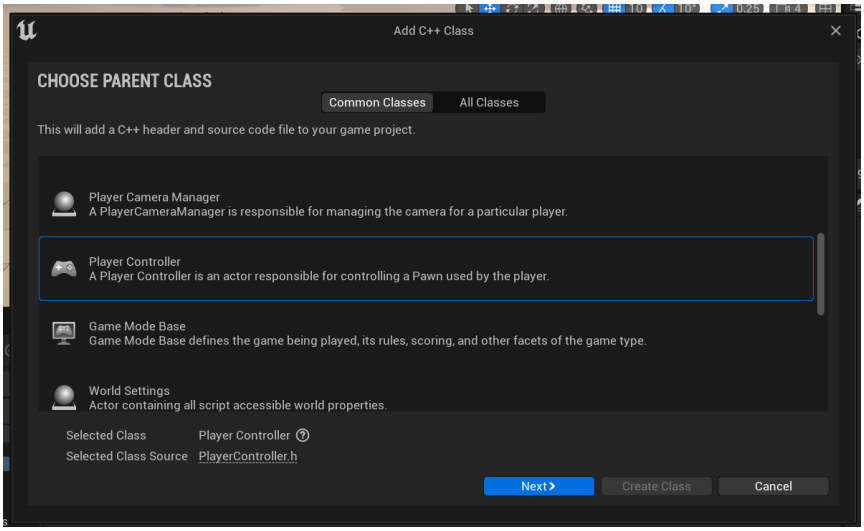


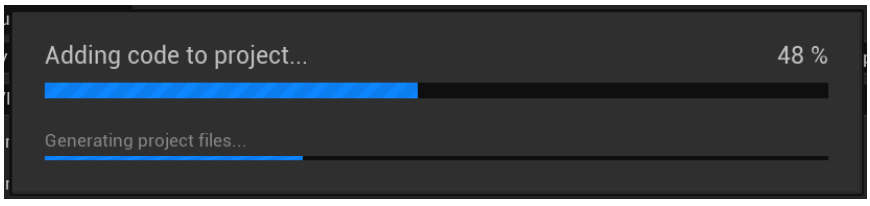
- Now, make sure your Virtualizer is plugged
- Press “Play” and check if you can walk in your VR project using the Virtualizer
- Please read the documentation below for further information on the Virtualizer’s locomotion functions and the Haptic Feedback

Step 2b: Setup Your Own Project – Option 2: Creating/Modifying your own C++ Player Controller (only for C++ projects)

- Start by creating a new C++ class of the type “Player Controller”:







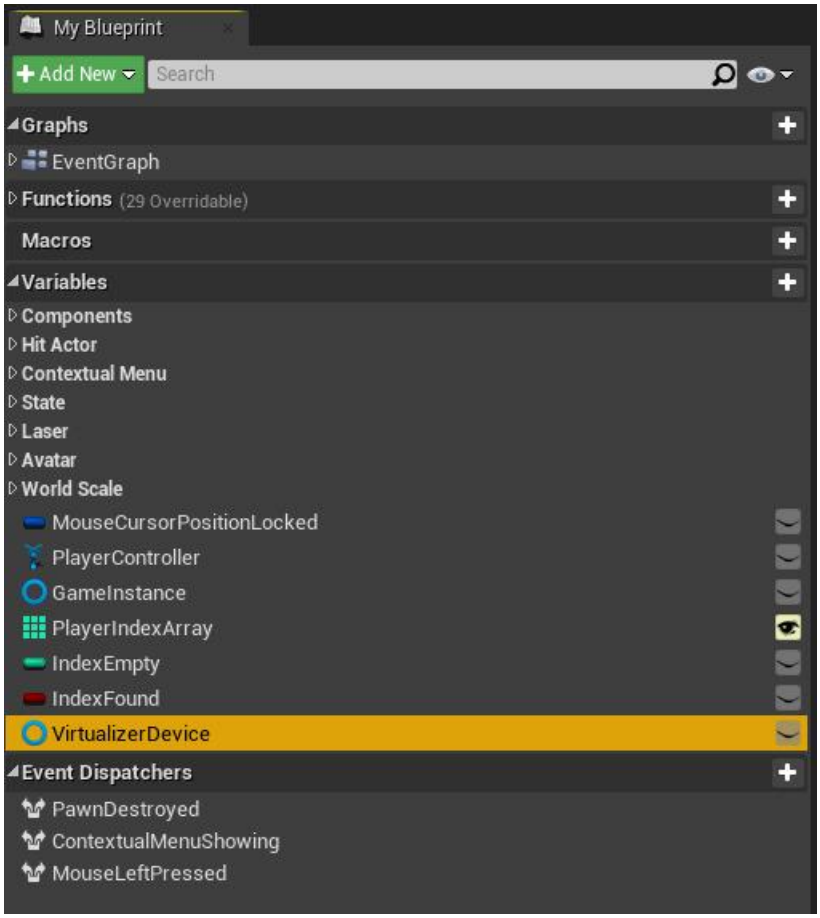
- For exemplary C++ code, please view the “Example usage” of Locomotion in the last chapter of this document. > follow link to chapter (click here)
- For further details, please check out the Doxygen Documentation for C++ available in our Developer Center at “developer.cyberith.com” (in the Section “Downloads / SDK”).

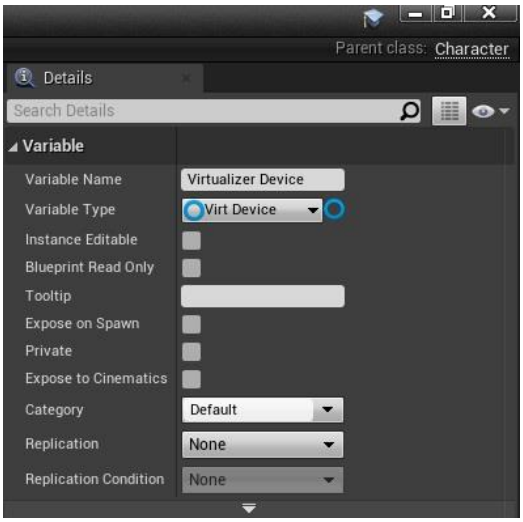
Step 2c: Setup Your Own Project – Option 3: Creating/Modifying your own Blueprint Player Controller

In the case that you don't want to use our standard character controller (and don't want to create custom C++ code), for example if you already use your own character controller or other logic limitations would block you, we also offer an alternative way to use the Virtualizer's data via Blueprints.

- You can start by creating a Blueprint variable of the type "Virtualizer Device"

Go to your desired component/actor blueprint.
On the right of the "Variables" dropdown view, you will see a small "**+**" where you will be able to type in the search view "Virt Device". Click on it to add the variable.

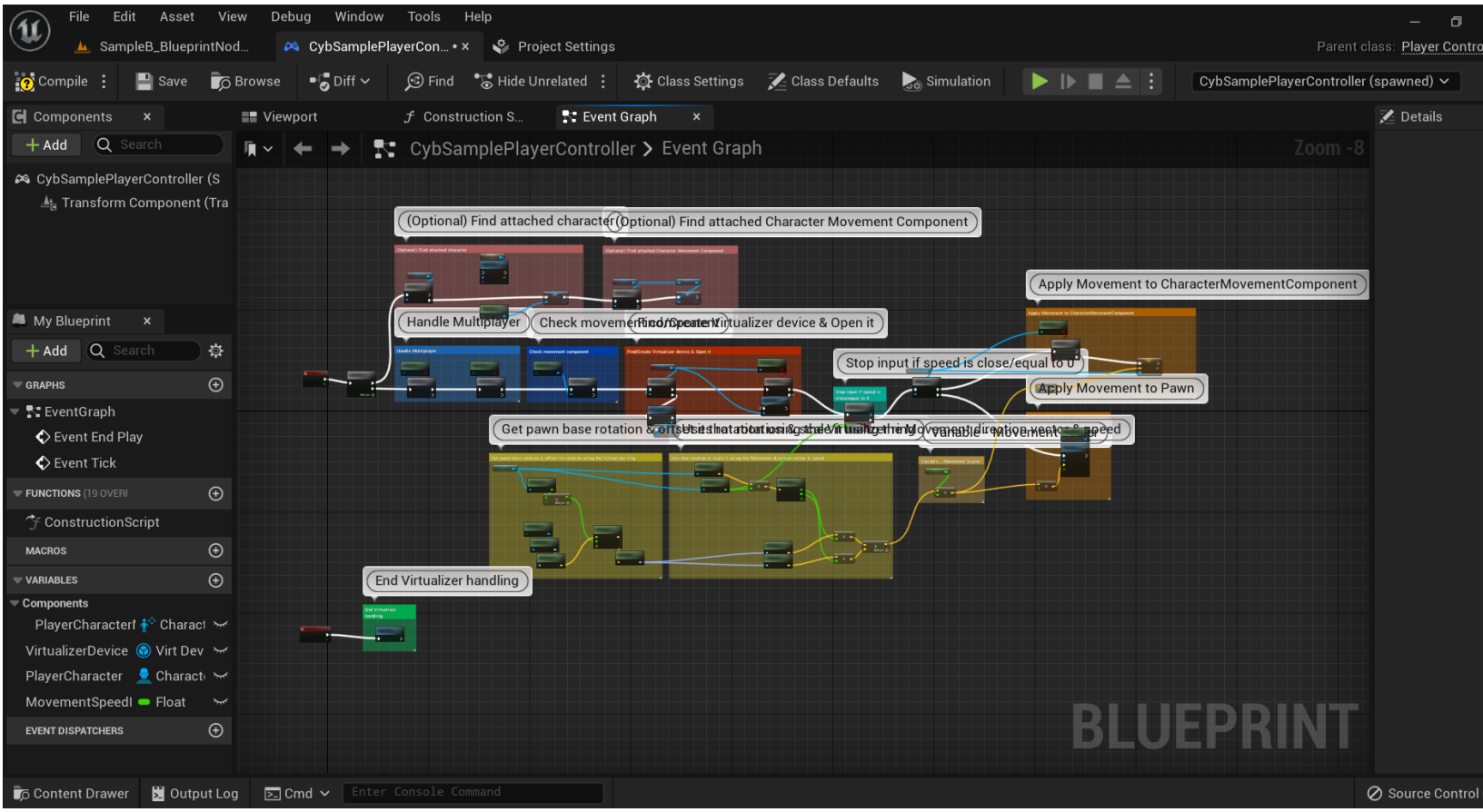




- Setup a node logic

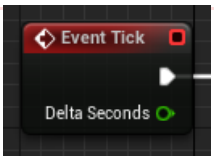
A particularly good starting point is to check the blueprint of *CybSamplePlayerController* from *SampleB_BlueprintNodes* and eventually to copy/paste its logic in your own blueprint.

This section explains how this blueprint is set up. It shall serve as an example for you to set up your own.



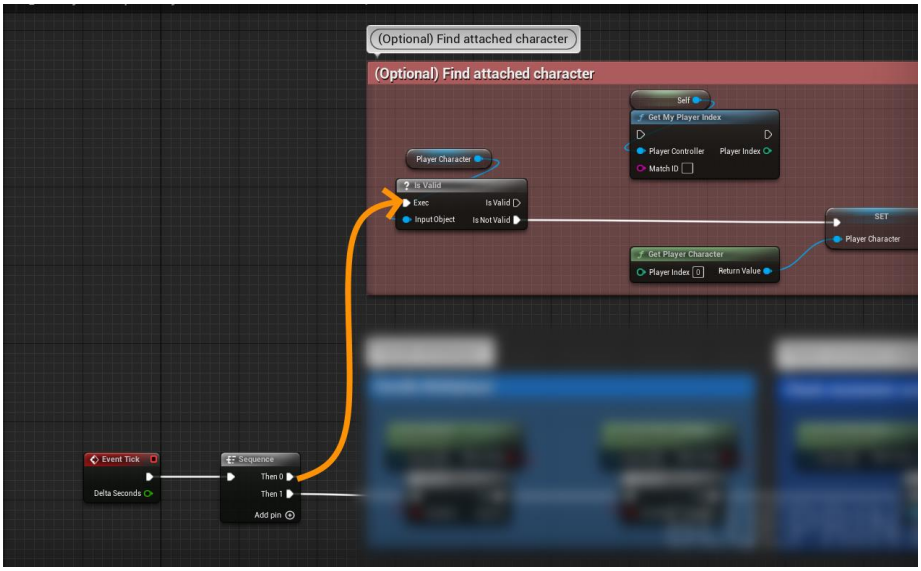
What does this blueprint do?

On the top left, you will find the “Tick” node.



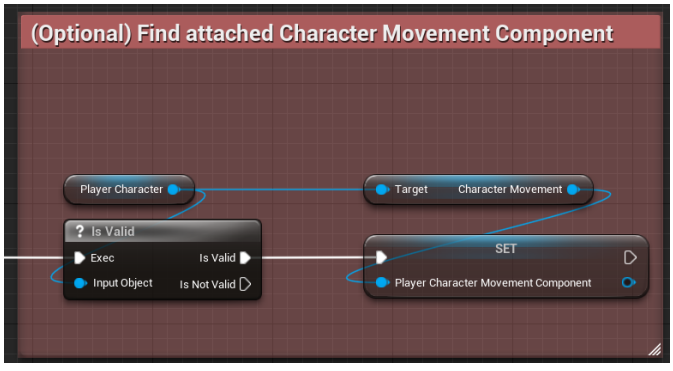
From the **Tick** which is the “every frame update” call:

- **First part of the sequence** is “finding components”

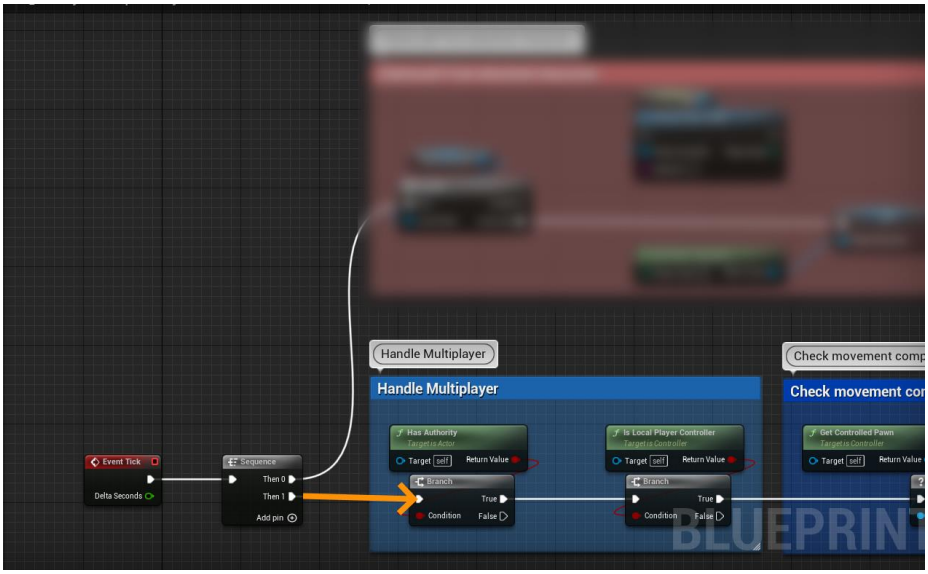


- We try to find an attached “Character” component & register it in the “PlayerCharacter” variable of this blueprint.

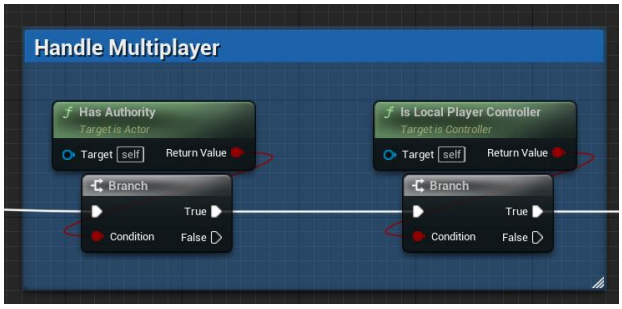
- We then try to find an attached “CharacterMovement” component & register it in the “PlayerCharacterMovement” variable of this blueprint.



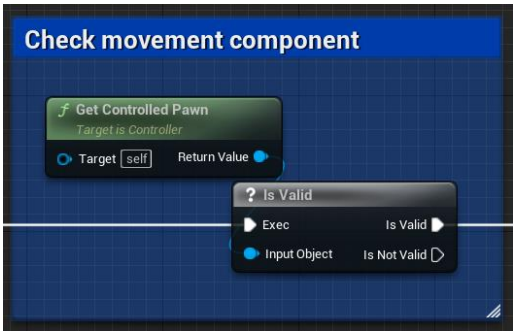
- **Second part of the sequence** is updating the player, rotating it & moving it.



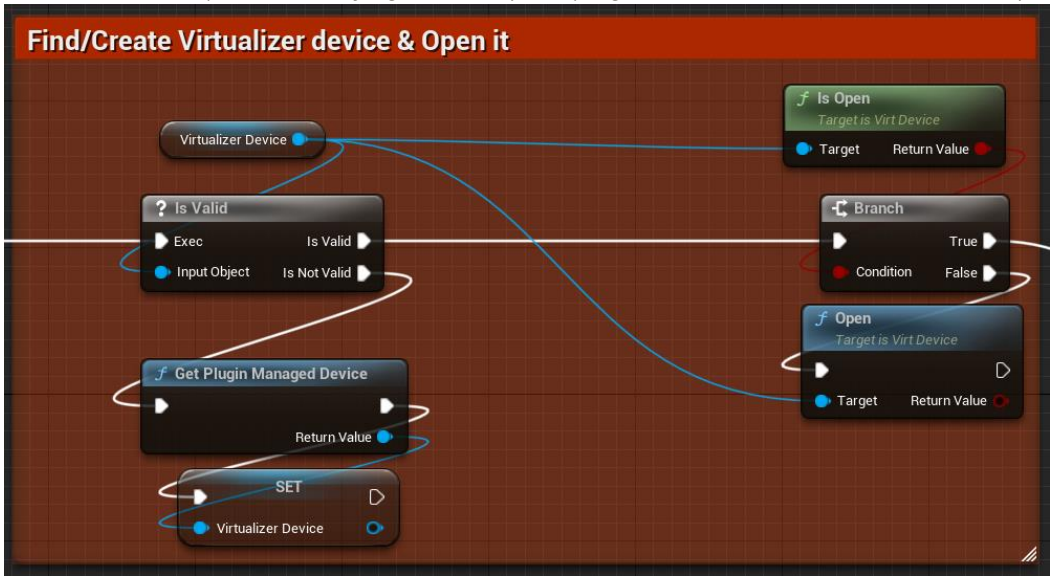
  - We first check if we are the owner of the PlayerController.



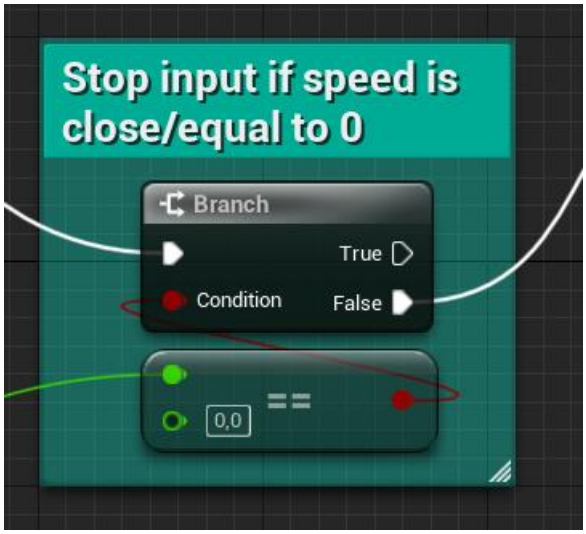
  - We then check the presence of a pawn possessed by our PlayerController



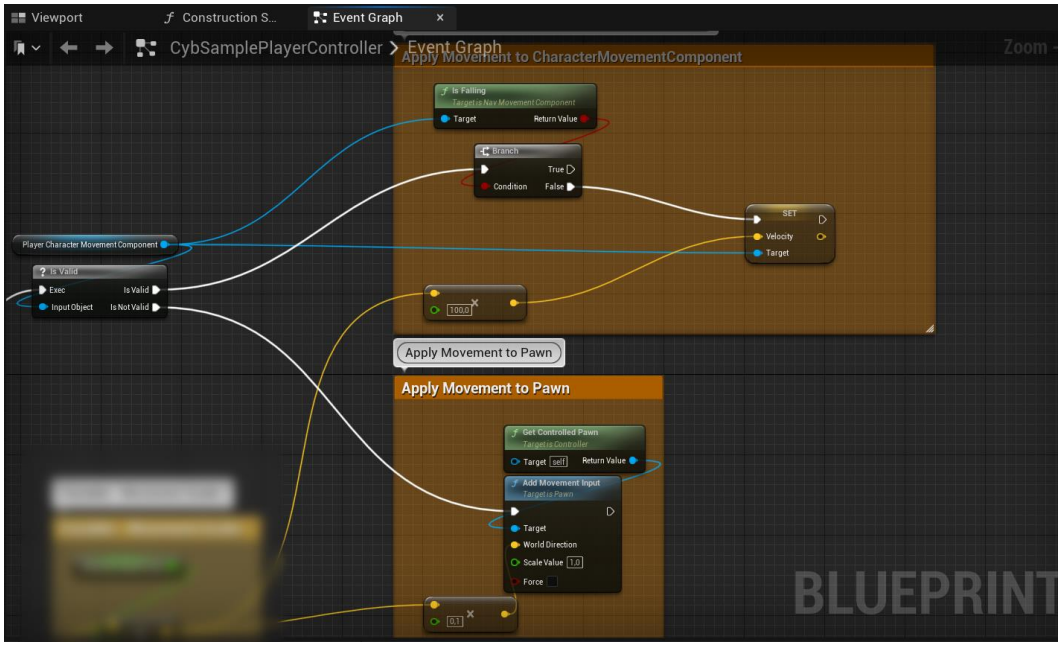
- Next in our logic, we get the plugin managed Virtualizer device & “open it” (ask data access to the Virtualizer) if it is not opened.
  As a reminder, *please do not forget that only one program can access Virtualizer data at any time!*



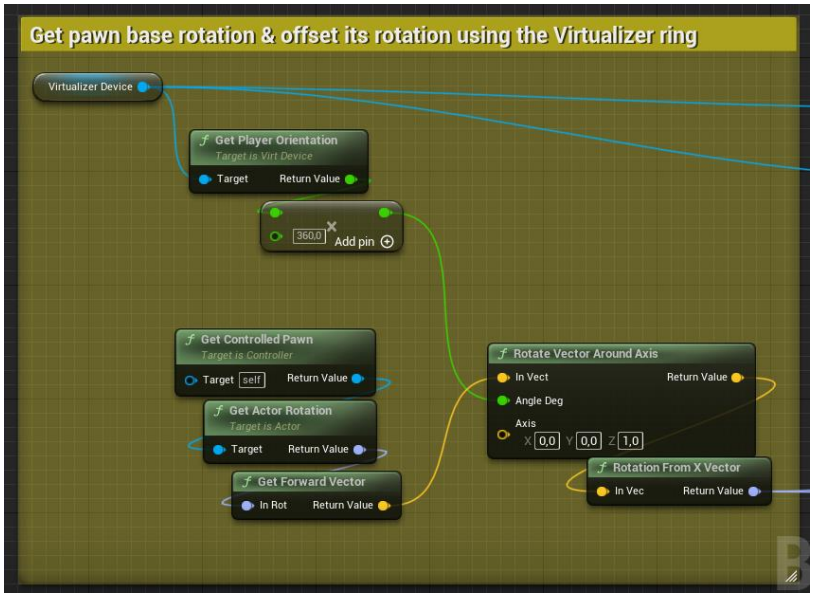
- We then prevent slow movement on the device to make us move while we shouldn’t, customize the behaviour as you see fit.



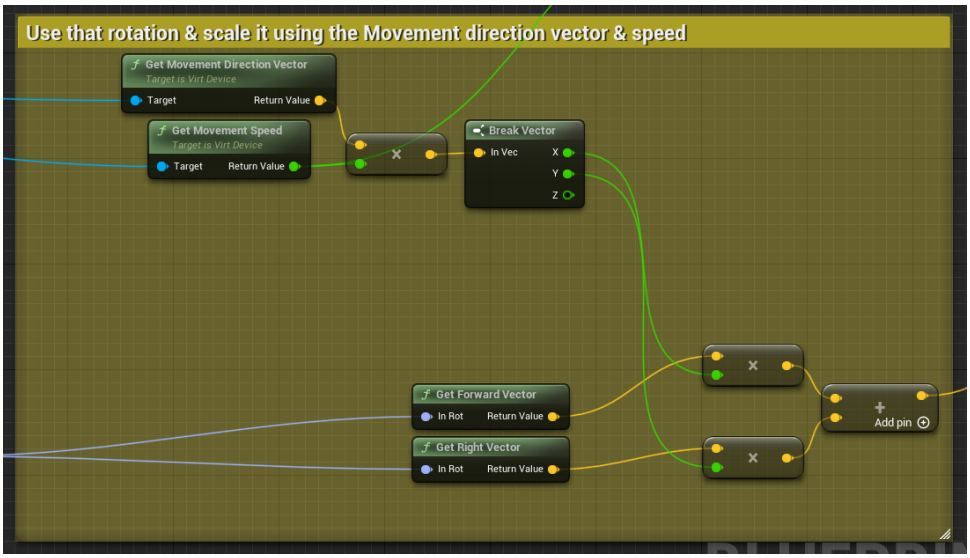
- And the node finishes by applying movement input to the CharacterMovement component or directly to the possessed Pawn depending on the possessed pawn components.



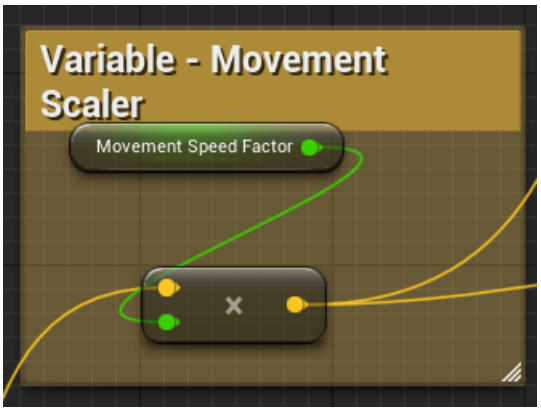
- This means that we have calculated the movement input before.
- We start by getting the Virtualizer Device variable we got earlier.
  - First, we get the Player Orientation data (which is between 0 & 1).
  - Let's multiply it by 360 to have an orientation ranging between 0° and 360°.
  - Then, get the Controlled Pawn's Forward direction vector.
  - This direction will then be rotated by the player rotation, to get the "Virtualizer" forward direction, relative to player's current rotation.



- We then continue by
  - Getting the Virtualizer's movement direction vector.
    > Is the player walking forward in the Virtualizer? Backward?
  - Scale it by the estimated Virtualizer's Movement Speed (meter / seconds).
    > Is the player walking fast? Slow?
  - Get the **forward** & **right** vector of the previously calculated rotation, scale it by the movement vector & make it thus a world oriented movement vector scaled with speed.

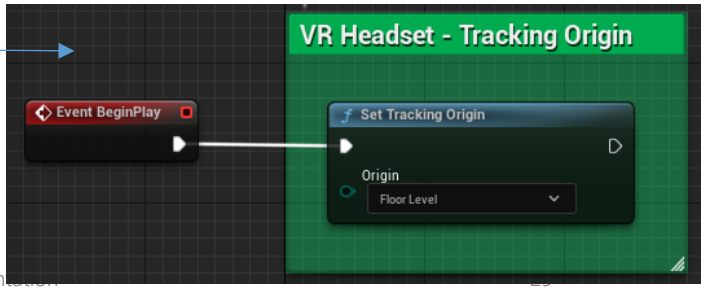


- **(Optional)** It's also possible to scale this movement by a custom factor through an easy to modify blueprint variable.



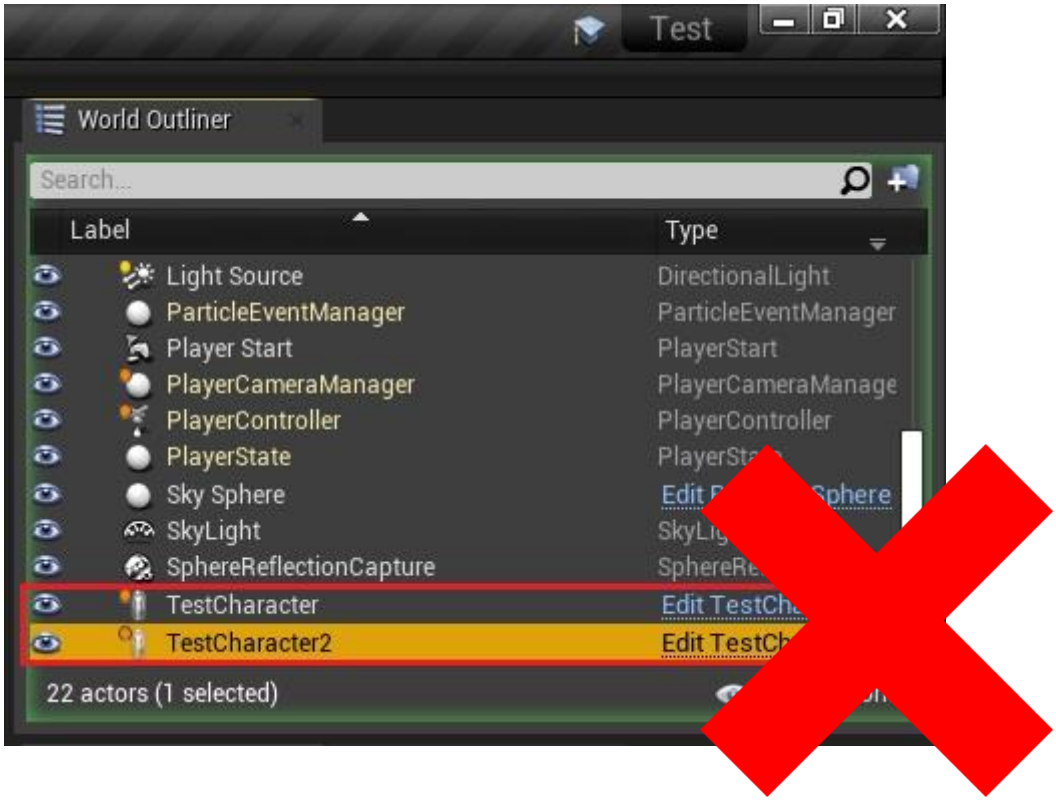
To finish this whole topic, we can also notice that in our CybSamplePlayer, we added a call to force the VR headset tracking "Origin" to be located on the floor (as normally you have setup your Virtualizer to be calibrated with your VR headset this way).

This is how you do it.



Note: In order to avoid potential problems caused by sending contradicting commands to the Virtualizer, **a Virtualizer can only have one open connection at a time!**
This means that it can also only connect to one Character Controller / Actor within Unreal Engine!

Tip: In case you can't open a connection to your Virtualizer, check if you have multiple Character Controllers activated during runtime of your application! You may spawn (an additional?) Character Controller when you are launching the application by pressing the "Play" Button.
In case there are two Character Controllers active during runtime, only one of them will connect to the Virtualizer.



If there are two Actors that both try to open a connection to the Virtualizer (at runtime), only one of them will be able to open the connection successfully.

No Virtualizer Hardware? – No Problem!

You may want to create applications for the Cyberith Virtualizer without having access to a real hardware device.
For this purpose, we added two kinds of virtual devices emulating a Virtualizer:

- Keyboard – WASD for movement and QE for rotation
- Controller (Xbox 360 & Xbox One Controller) – left joystick for movement and right joystick for rotation. Please plug the Xbox Controller to your PC per USB cable to ensure proper functionality.

This means:
If you don't have a Virtualizer available you can still test the functionality of your implementation with the help of an Xbox Controller or with the help of your keyboard!

In case no Virtualizer is plugged, the system will automatically use the Xbox Controller. If neither a Virtualizer nor an Xbox Controller are plugged, the system will automatically use the keyboard.
Alternatively to the automatic selection, these inputs methods can be configured in the project settings. – as described in the next chapter.

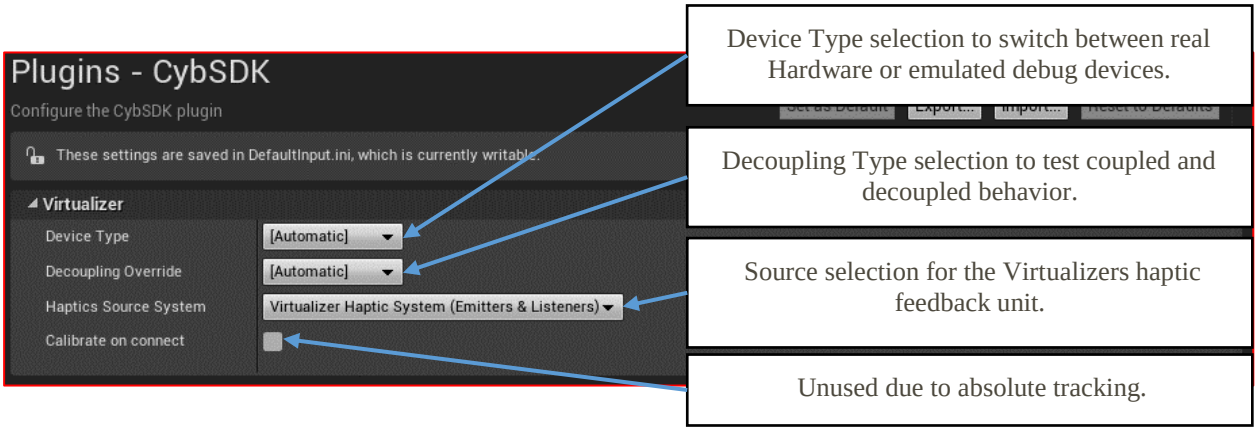
Note:

- Be aware, that these settings will still be active in a built executable and allow to test the finished (built) application without the need of real Virtualizer hardware.
- Currently, you can not walk backwards with an Xbox Controller. If you press the left joystick back, the avatar still walks forwards. That does not mean that you can't walk back with the Virtualizer! It is a problem caused by the Xbox controller implementation.
- The Xbox Controller allows you to basically check on the functionality of the haptic feedback (although it can only rumble with one frequency and not in many different ones like the Virtualizer.) The keyboard input does not allow you to check on the haptic feedback functionality.
- For development and testing purposes it can be practical to use an Xbox Controller even if you have a real Virtualizer next to you. Using such a controller, you don't need to stand up from your comfortable chair for every single test ;)

Project Settings Documentation

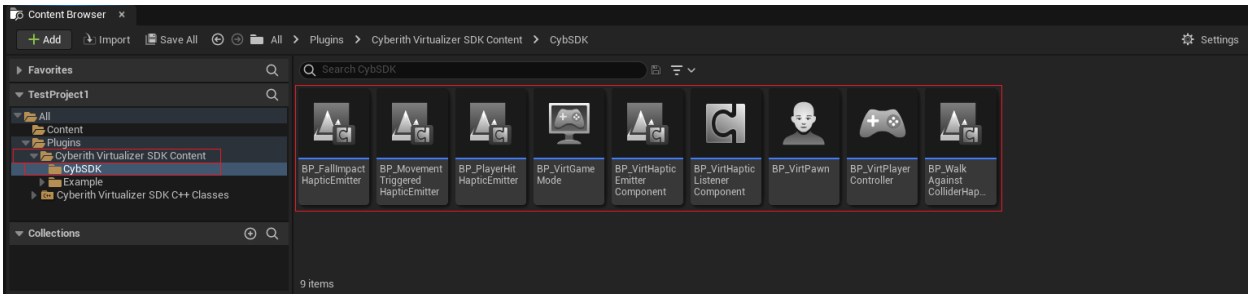
The main settings for the Virtualizer are available in your projects settings in the Plugins section.
Click *Settings* (in UE 4) *or Edit* (in UE 5), then click *Project Settings*. Scroll down to the section *Plugins* and click on *CybSDK* to find the following settings:

Virtualizer



Blueprint Settings Documentation

All of the following settings can be found through the Content Browser of Unreal Engine.

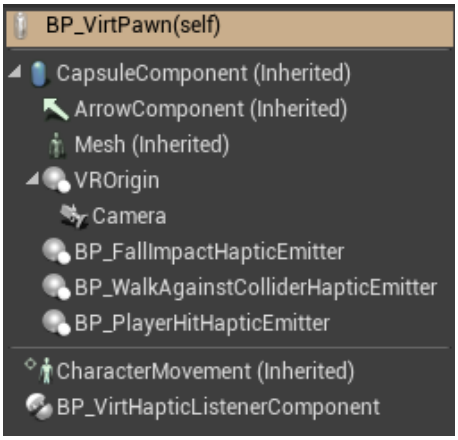


Click *Cyberith Virtualizer SDK Content* and then *CybSDK* in order to see the following components:

BP_VirtPawn

The BP_VirtPawn is a Pawn object, derived from the Unreal Character class. This pawn is moved around when the user walks in the Virtualizer. It also handles the haptic feedback of the Virtualizer. The BP_VirtPawn is used in the ExampleMap. The VirtPawn is selected in the BP_VirtGameMode described later and is spawned at the Player Start position when the ExampleMap is started.

All the inherited objects and the VROrigin (including the Camera) are Unreal components responsible for movement and visualization in Unreal.

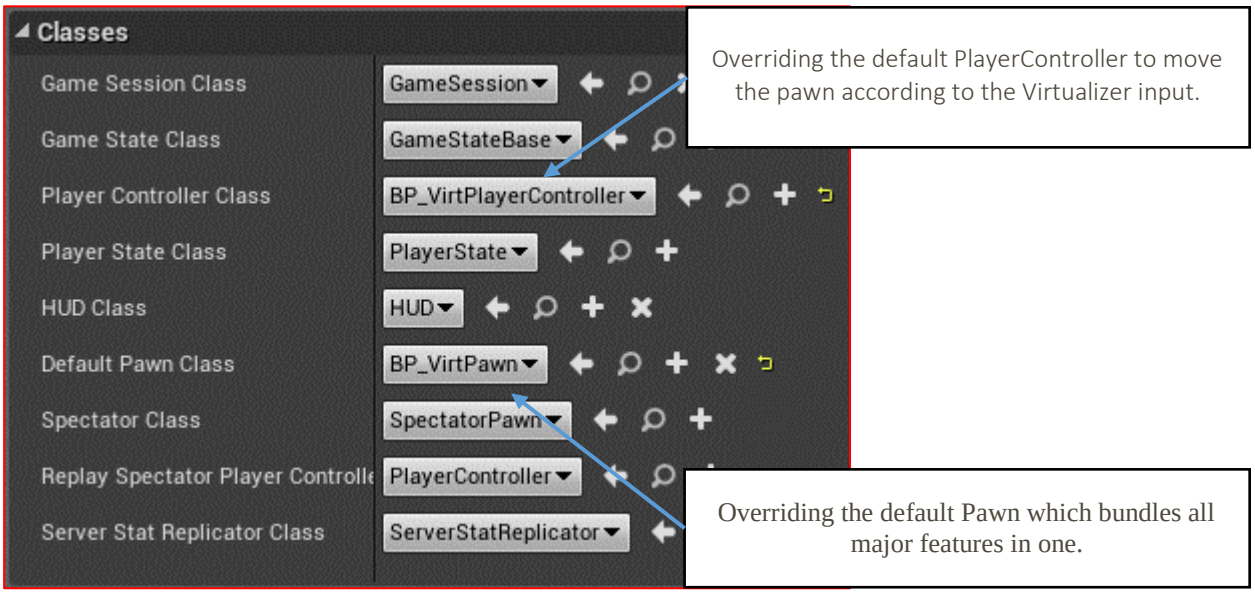


The other Blueprints are provided by Cyberith and handle the haptic of the Virtualizer.

If you want to deactivate certain HapticEmitter Blueprints, you have to select the specific HapticEmitter Blueprint and unselect the “Auto Activate” parameter in the Activation property in the details view.

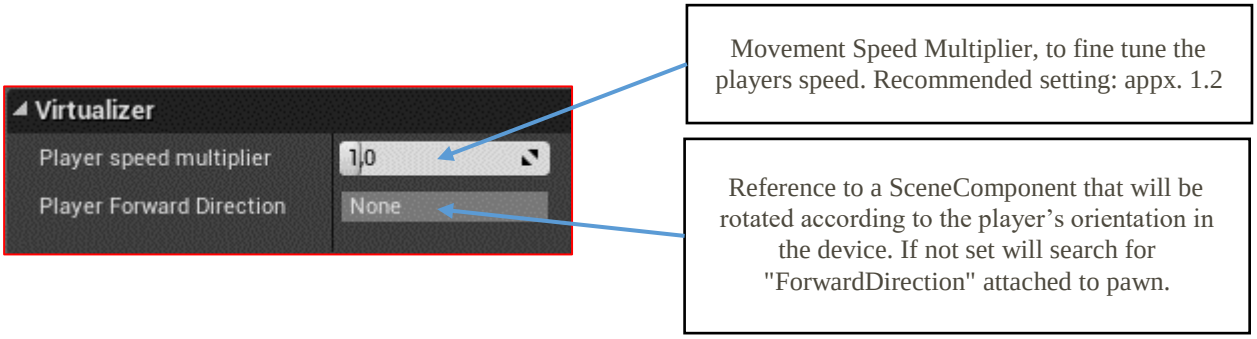
BP_VirtGameMode

The CybSDK defines its own GameMode to define the default Blueprint classes to be spawned.



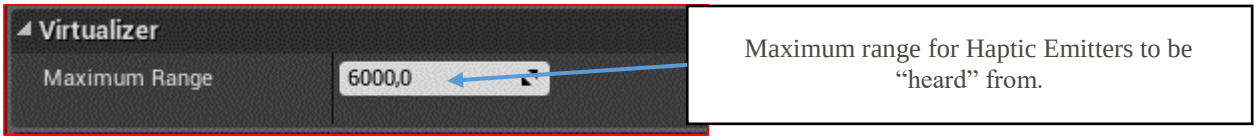
BP_VirtPlayerController

This script moves the pawn according to the Virtualizer input, as described in the Example Usage.



BP_VirtHapticListenerComponent

This script receives haptic feedback emitted by all active HapticEmitters in range and activates the Virtualizers haptic vibration unit.



The Haptic Listener sends two parameters for the haptic unit. One of these two parameters is the frequency, the haptic unit vibrates with. The other parameter is the volume (= "strength") the haptic unit vibrates with.

As the resonance frequency of implemented haptic unit is around 40 Hz, the Haptic Listener does not send out frequencies from 35 to 45 Hz. This avoids loud and uncomfortable vibrations. Instead of sending frequencies of 35-39 Hz it sends 34 Hz and instead of sending frequencies of 40-45 Hz it sends the frequency of 46 Hz. You might notice this behavior if you select one of these frequencies in a haptic emitter, which is explained in the next chapter.

In case multiple Haptic Emitters are active and within range, the HapticListener makes a weighted average of all the frequencies and sums up the volume caused by all Emitters. The volume of an emitter is defined by two curves ("Volume Over Time" & "Force over Distance") for each Emitter individually. Note, that the overall strength of the vibration can not exceed the maximum strength, independently of how many emitters you add.

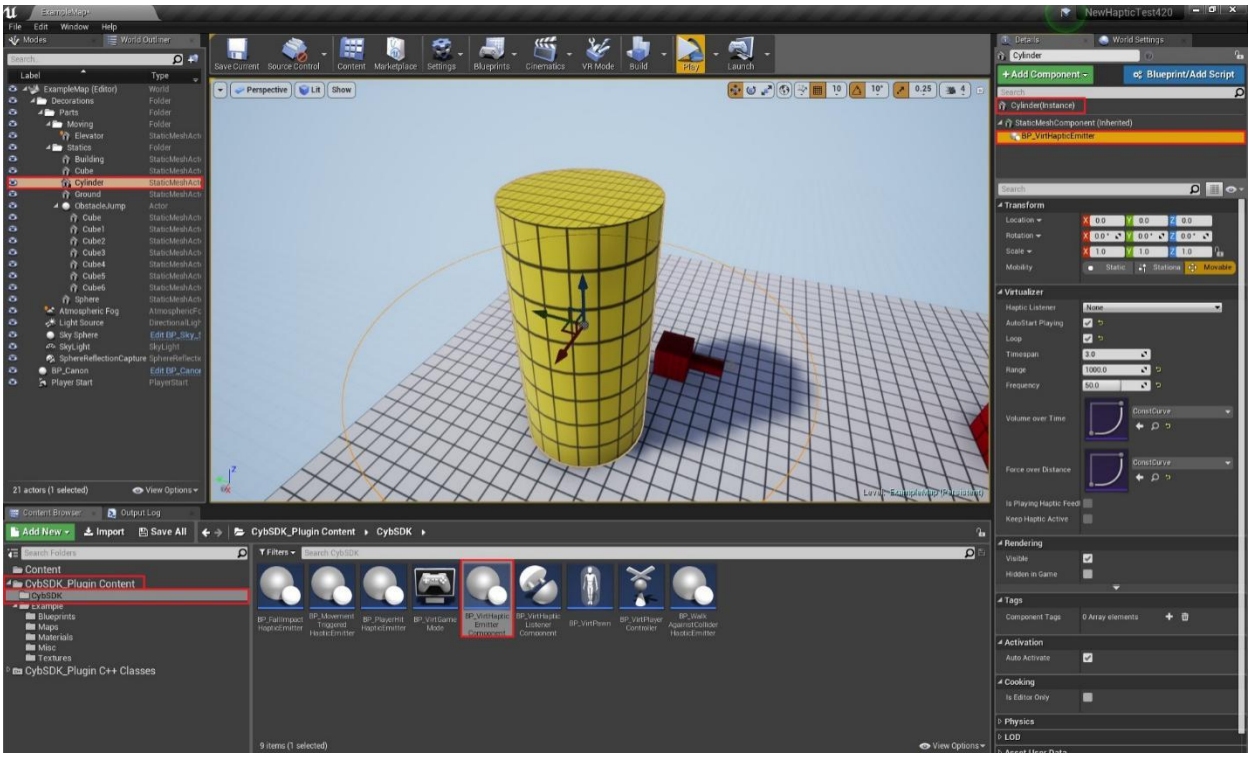
BP_VirtHapticEmitterComponent

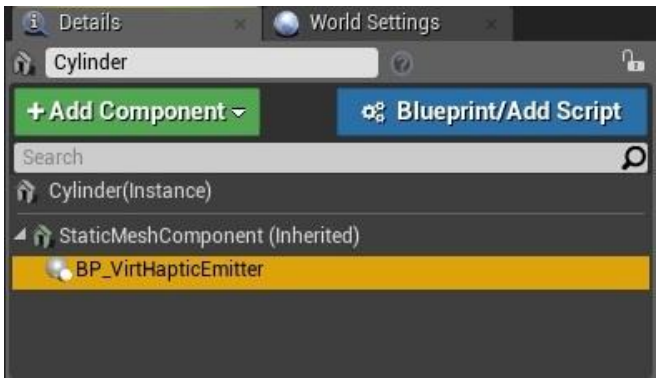
This script emits signals causing haptic feedback in a specific radius around it. These signals are received by a HapticListener, which in turn causes the Virtualizer baseplate to vibrate.

Attach such a Haptic Emitter to objects, that you want to cause haptic feedback!

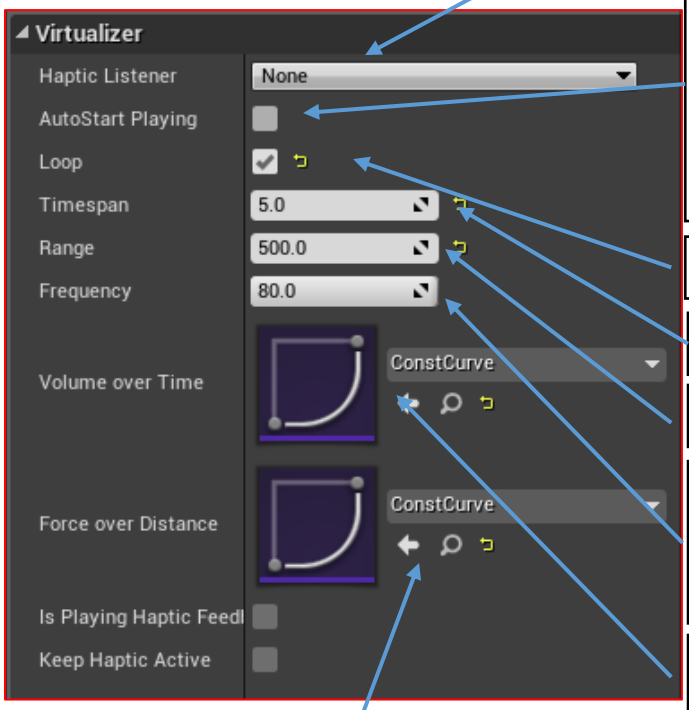
Add a Haptic Emitter by:

- Select an object you want to cause vibrations.
- Select the Haptic Emitter you want to use (f.ex. “BP_VirtHapticEmitterComponent”)
- Add the Haptic Emitter to the object by Drag & Drop.





Once added, you will see this Haptic Emitter Script:



Reference to the Haptic Listener receiving Haptic Feedback. If not set will find one in scene

Automatically start playing on application startup. startup. Check this box, if you want an object to cause vibrations whenever the player comes close to it. Otherwise, you need to activate it separately by calling the Play method of this class. To deactivate the Emitter, call the Stop method. The Play and the Stop method are explained below.

Loop the haptic feedback over time.

Duration in seconds for the feedback.

Max range of the haptic effects.

Frequency for the haptic unit. Range: 10-80Hz. The Haptic Listener filters frequencies between 35-45Hz and sets it to 34 or 46 Hz. Keep that in mind when choosing frequencies.

FloatCurve for the Haptic force over time. Normalized [y = force factor, x = time in s]

FloatCurve for the Haptic force over distance to the Haptic Listener.
Normalized [y = force factor, x = distance in m]

The generic Haptic Emitter "BP_VirtHapticEmitterComponent" Blueprint is the base Blueprint for causing haptic feedback. You can either activate the haptic functionality by ticking the "AutoStart Playing" checkbox or call the Play() and Stop() method of the Blueprint (the functionality is implemented in C++, but there are callable Blueprint methods) to manually activate and deactivate the haptic emitter whenever you like to do so.

Additionally to the generic Haptic Emitter Blueprint we added four specific Haptic Emitter Blueprints for special use cases.

In these specific Haptic Emitter Blueprints you can see the Play and Stop methods in action. Check out the Blueprints and you can see examples of how these methods are used.

Specific Haptic Emitter Scripts

The following specific Haptic Emitter Blueprints are part of Cyberith's Unreal Plugin to demonstrate exemplary use cases of haptic feedback. Feel free to add your own specific Haptic Emitter Blueprints for whatever purposes you like.

- BP_FallImpactHapticEmitter: Haptic is triggered when the player lands on the floor after falling/jumping from higher ground. The player has to fall for more than 0.1 seconds to activate the haptic feedback once reaching the ground. This Blueprint is attached to the BP_VirtPawn.

- BP_MovementTriggeredHapticEmitter: Haptic is triggered when the object, this Blueprint is attached to, moves. This Blueprint is attached to the Elevator object in the ExampleMap.

- BP_PlayerHitHapticEmitter: Haptic is triggered when the player is hit by an object. The object hitting the player has to trigger the PlayerHit Event of this Blueprint. This Blueprint is attached to the BP_VirtPawn. Furthermore, bullets (looking like big white balls) are spawned during runtime from the red canon (consisting of a cube and a cylinder). The bullets trigger the PlayerHit Event to cause haptic feedback upon hitting the BP_VirtPawn.

- BP_WalkAgainstColliderHapticEmitter: Haptic is triggered when the player walks against an obstacle. The haptic is activated when the player walks against the obstacle for more than 0.5 seconds. This Blueprint is attached to the BP_VirtPawn.

If one of these Blueprints is added to an object in a scene, the generic Haptic Emitter Blueprint ("BP_VirtHapticEmitterComponent") shall not be used. The described specific Haptic Emitters work autonomously.

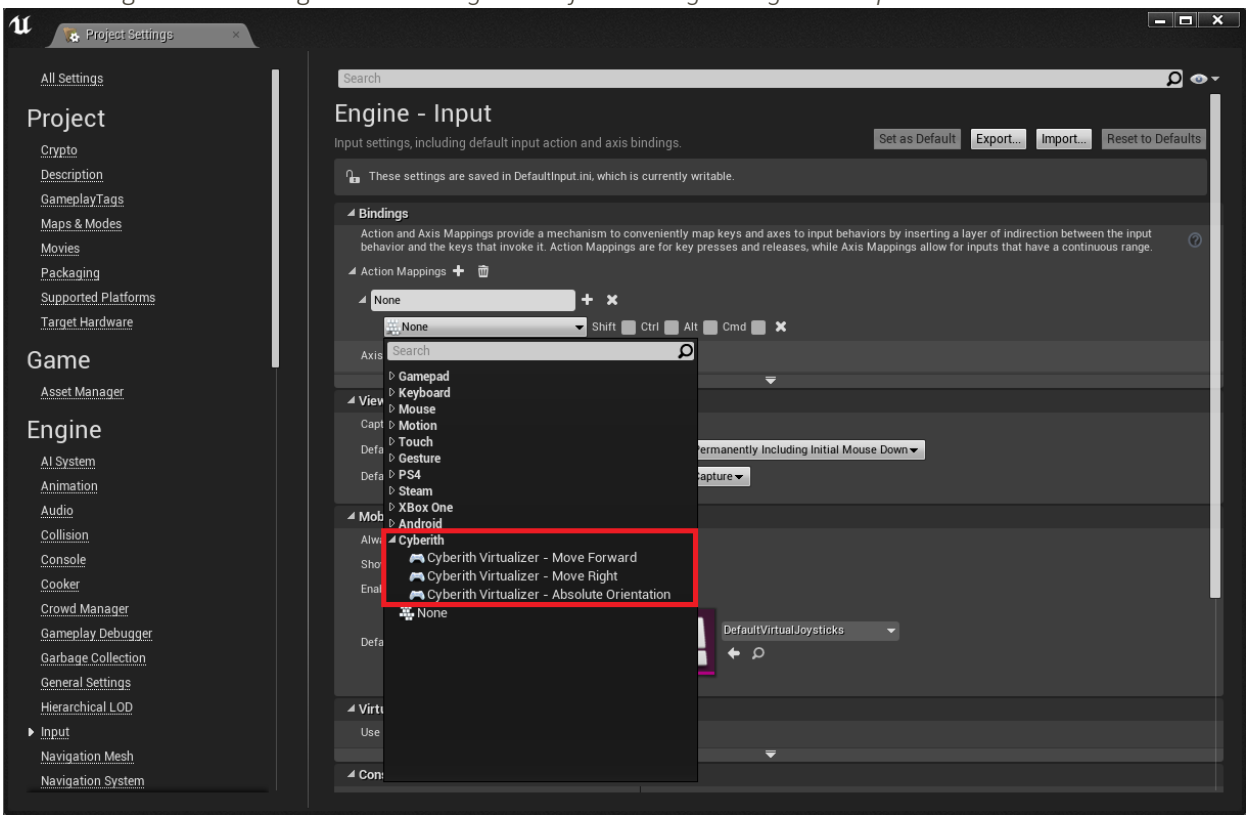
The "AutoStart Playing" checkbox is not available on any of these scripts. These emitters use the Play and Stop methods of the Haptic Emitter Blueprint.

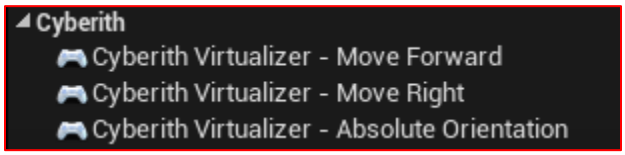
Compatibility with Standard Functions of Unreal Engine

The native CybSDK handles two functionalities differently compared to the standard functions of Unreal Engine.
In order to ease the process of implementing the Virtualizer into pre-existing systems, we added two “UE compatibility modes” in order to match the standard functions of Unreal Engine:

Gamepad Axis Emulation

Using this functionality, the Virtualizer device emulates standard GamePad Axis input for better compatibility with existing PlayerInput Systems.
This settings can be configured in *Settings -> Project Setting -> Engine -> Input*.



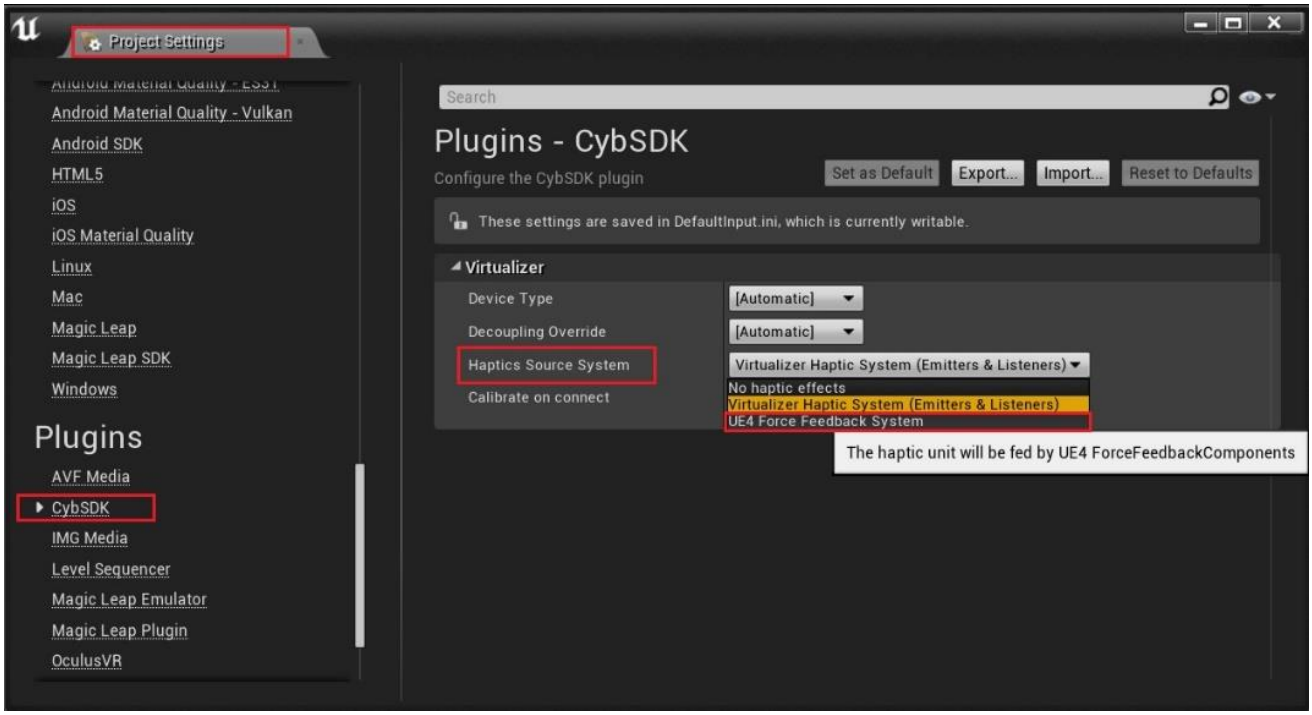


Move Forward	== Left Joystick Up
Move Right	== Left Joystick Right
Absolute Orientation	== Right Joystick result

Force Feedback Input

The Virtualizer is fully compatible with the UE4 Force Feedback System for haptic feedback. We created this mode for better compatibility with existing PlayerInput Systems.

This mode can be activated in *Settings -> Project Settings -> Plugins -> CybSDK*



Blueprints

All functionalities and classes provided by the CybSDK Plugin are available in Blueprint.

UVirt

This UBlueprintFunctionLibrary exports static functions into blueprint.

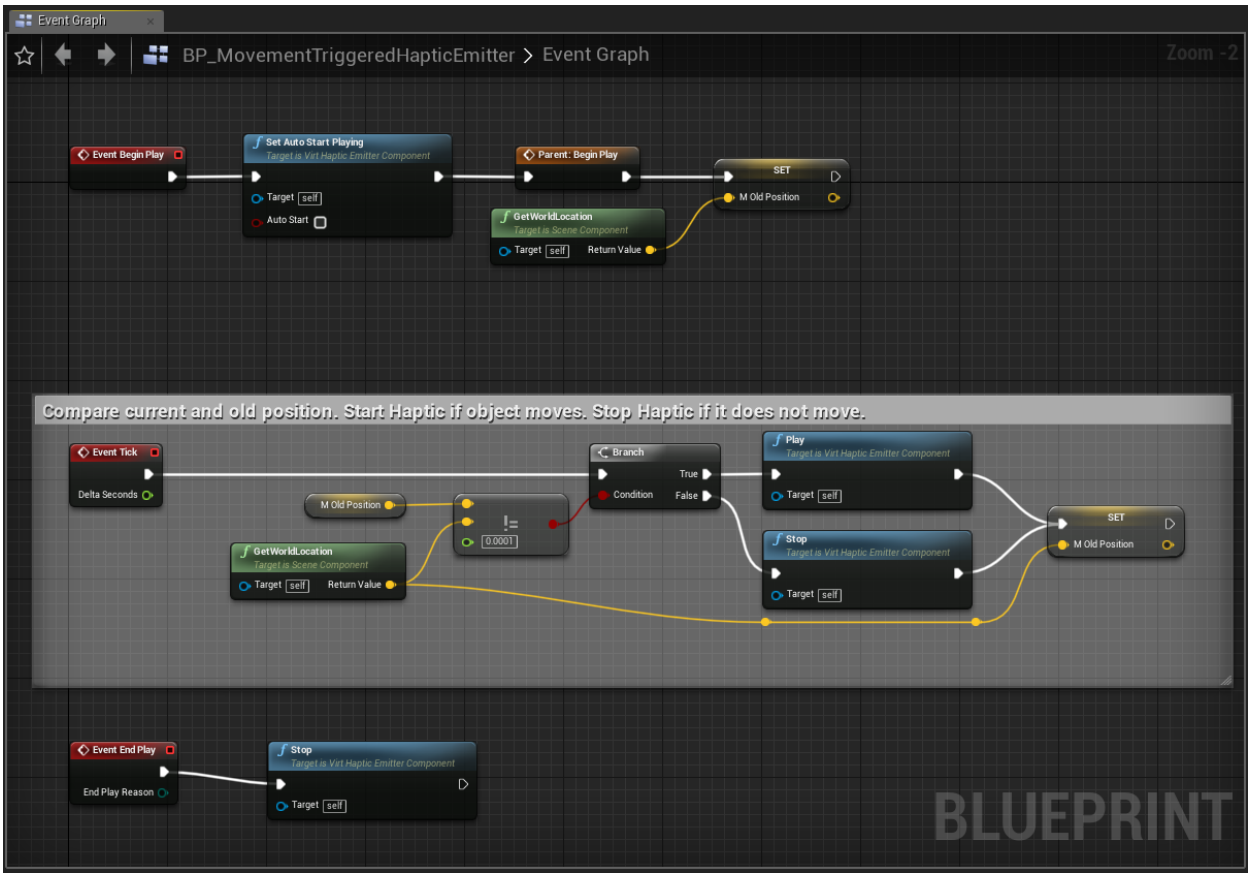
GetSDKVersion

Returns: int
Returns the version number of the Virtualizer SDK.

GetPluginManagedDevice

Returns: UVirtDevice*
Gets the Virtualizer device object currently managed by the CybSDK Plugin.

Sample – Haptic Emitter



SDK Documentation

C++ SDK Documentation

For full documentation of the C++ SDK take a look into the official online Documentation.
All classes and functions are documented in their respective header file located in the **Include** directory and should show up in your Visual Studio IntelliSense

FCybSDK_PluginModule

This class manages the main CybSDK plugin functionalities.

GetVirtualizerInputDevice

Returns: TSharedPtr<FVirtInputDevice>
Returns the UnrealEngine IInputDevice device managed by the FCybSDK_PluginModule.

FVirtInputDevice

This class has full authority over the Virtualizer device. Here the device is selected, a connection established and managed.

GetDevice

Returns: TWeakObjectPtr<UVirtDevice>
Returns the Virtualizer device managed by the FVirtInputDevice.

IsDecoupled

Returns: bool
Returns true if the UVirtDevice supports decoupled movement, otherwise false.

UVirtDevice

This blueprintable wrapper class holds the native Virtualizer Device and supports multiple Unreal specific methods.

GetMovementVector

Returns: FVector
Returns the movement direction as a speed scaled vector relative to the current player orientation.

GetMovementDirectionVector

Returns: FVector
Returns the movement direction as vector relative to the current player orientation.

GetPlayerOrientationVector

Returns: FVector
Returns the orientation of the player as vector.

GetPlayerOrientationQuaternion

Returns: FQuat
Returns the orientation of the player as quaternion.

UVirtHapticEmitterComponent

Play

Start playing the Haptic Emitter by adding it to the Haptic Listener

Stop

Stop playing the Haptic Emitter by removing it from the Haptic Listener

C++ Example usage

Locomotion

```
void AVirtPlayerController::Tick(float DeltaSeconds)
{
    /** Base call */
    Super::Tick(DeltaSeconds);

    /** Check components */
    // Ignore Virtualizer input if not local player
    if (!IsLocalPlayerController())
        return;

    // Ignore Virtualizer input if not possessed
    APawn* pawn = GetPawn();
    if (pawn == nullptr)
        return;

    // FCybSDK_PluginModule::GetVirtualizerInputDevice() will be initialized
    // by the Unreal Engine in the first player tick
    if (m_deviceController.IsValid() == false)
    {
        m_deviceController = FCybSDK_PluginModule::GetVirtualizerInputDevice();
        if (m_deviceController.IsValid() == false)
            return;
    }

    TWeakObjectPtr<UVirtDevice> device = m_deviceController->GetDevice();
    if (device == nullptr || !device->IsOpen())
        return;


    /** Calculate Movement & Apply Orientation */
    // Get movement speed
    FVector movement = device->GetMovementVector() * MovementSpeedMultiplier;

    // Get player orientation
    FQuat localOrientation = device->GetPlayerOrientationQuaternion();

    // Determine global orientation for characterController Movement
    FQuat globalOrientation;
```

```
    // For decoupled movement we do not rotate the pawn --> HMD does that
    if (m_deviceController->IsDecoupled())
    {
        if (m_forwardDirection != nullptr)
        {
            m_forwardDirection->SetRelativeRotation(localOrientation);
            globalOrientation = m_forwardDirection->GetComponentQuat();
        }
        else
        {
            // Quaternions are applied right to left
            globalOrientation = localOrientation * pawn->GetActorQuat();
        }
    }
    // For coupled movement we rotate the pawn and HMD
    else
    {
        pawn->SetActorRotation(localOrientation);
        globalOrientation = localOrientation;
    }

    FVector motionVector = globalOrientation * movement;
    m_motionVector = motionVector;

    if (motionVector.IsZero())
        return;

    /** Apply Movement */
    // If pawn is a character we use CharacterMovementComponent
    if (m_characterMovementComponent != nullptr)
    {
        // * 100 for correct Unreal Units --> MoveSmooth(m/s * 100) -->
MoveSmooth(cm)
        m_characterMovementComponent->MoveSmooth(motionVector * 100.0f,
DeltaSeconds);
    }
    // If pawn has explicit MovementComponent use it
    else if (m_movementComponent != nullptr)
    {
        // AddMovementInput wants normalized vectors --> 10 is the theoretical max
speed a user can achieve
        m_movementComponent->AddInputVector(motionVector / 10.0f);
    }
```

```
    // Otherwise use simple move
    else
    {
        // * 100 for correct Unreal Units --> Set( cm + m/s * 100 * s) --> Set(cm)
        pawn->SetActorLocation(pawn->GetActorLocation() + motionVector * 100.0f *
DeltaSeconds);
    }
}
```